CPS352 Lecture - Course Introduction; Fundamental DBMS Concepts

Last revised December 1, 2022

Objectives:

1. To introduce the syllabus and key issues of the course - showing how they are related.
2. To contrast file-processing systems and DBMS-based systems.
3. To introduce key DBMS concepts and terminology

Materials:

1. Course syllabus to hand out
2. Example database demo setup (intro)
3. On Canvas copy of SQL used for it [test run examples on it]
4. File containing projectable versions of:
   a. Demo database creation code
   b. DBMS between program and data
   c. Above with user interface added
   d. Figure 1.2 from book
5. Design project posted on Canvas and printed to hand out

Setup for demos on personal laptop

1. Create local account aardvark
2. As db2inst1, create database intro
3. connect to intro and then create tables and content using intro.sql

I. **Course Introduction**

Go over syllabus. Mention that we will be relating key ideas to the course schedule later in this lecture.

II. **File Processing Systems vs Database Management Systems**

A. At the outset of our course on database management systems, we want to review two different approaches to meeting the information processing needs of an organization: a file-processing approach versus a database management system approach.

1. Historically, these two approaches evolved successively.

   a) Early computer applications were always developed using the file processing approach, because that was the only approach known.

   b) The DBMS approach was first developed in the 1960's

      (1)In the 1970‘s and 1980‘s it came to be widely used in a variety of application areas - many falling into the broad category of "business data processing", but for other areas as well.

      (2)Today, DBMSs are the backbone of a host of computer applications across the entire spectrum and in some cases are embedded with applications (e.g. several web browsers embed a DBMS known as SQLLite to manage configuration data internally)

2. Though we will distinguish these approaches, we don't want to treat them as totally mutually exclusive. In fact, for many applications, the file processing approach is the best way to go. (Word processing and spreadsheets, for example, work this way) Both approaches can be used at various points in the information processing activities of an organization; each is legitimate in its own proper domain.

B. Regardless of which approach is used, an organization's needs are going to be met by a collection of files containing its operational data and programs that manipulate this data.

1. We distinguish between operational data that is relatively permanent and transient input/output data.

   Example: In a payroll application, the operational data includes the basic employee data such as name, SSN, rate of pay etc., plus year-to-date totals for income, taxes etc. This data is retained in storage relatively permanently.

(This is distinct from transient data like weekly time card information, which is discarded once processed.)

2. In either case, we call the collection of files that together contain the organization's operational data its DATABASE.

3. What distinguishes the file processing approach from the DBMS approach is how the programs relate to the operational data. (A file processing approach is often still used with transient data.)

C. We might say that a file-processing approach is characterized by a close relationship between programs and data.

1. In a file-processing approach, each program is written to process a certain file or group of files and must embody detailed knowledge about the structure of each file it uses.

   Example: Consider a program that prints mailing labels to send a mailing to each customer on an organization's customer list, written using a file-processing approach. This program presumably works with the organization's customer file, and must, therefore, incorporate knowledge about how the customer file is structured.

2. As a corollary, any change in the structure of the file will necessitate a change in the program. If the same file is used by several different programs, each program must be changed when the file structure is changed.

   Example: Suppose the company decides to adopt a policy of giving different discounts to different customers. This might call for adding a discount field to the customer file. Of course, this change does not affect the mailing list directly; nonetheless, the mailing list program must be modified to reflect the changed file structure.

3. To avoid this unintended coupling between unrelated programs, historically, it was common to design file-processing type systems so that each application area “owned” its own files.

4. File-processing based systems, then, tended to be characterized by a proliferation of application-specific files, each with its own format. Certain data items would stored redundantly - i.e. in more than one place in the database. This, however, created new problems:

   ASK CLASS

   a) Wasted storage (becoming less of a problem as storage costs go down, but still a concern, especially when one thinks of backup using a network.)

   b) Update problems: when an item of information has to be changed, it may need to be changed in several different places in the database. This means extra work each time an update has to be done.

   c) Inconsistency problems: over time, it became likely that the database would contain two different values for the same data item in two different places, because some update operation did not catch all of the places that need to be changed. This caused confusion.

      Example: Gordon's first computerized registration system (1980's) maintained a separate student file for each academic term. Each file contained various personal data on the student, the name of their advisor, and a list of the courses he/she was enrolled in that term. The file also contained space to record the grades for each course taken, though of course these slots would not be filled in until after the end of the term.

      (1) As registration time for a new term approached, the computer center would copy data from the current term's file into a file for the new term, blanking out the list of courses registered for but leaving all else intact.

      (2) At some point in time, the registrar's office could have three different files active:

         (a) The term just completed, awaiting the posting of grades and printing of grade reports, plus the possibility of grade changes by the professor.

         (b) The current term.

(c)The upcoming term, since registration for a new term is held toward the end of the preceding term.

(3)Any change in basic student information would have to be posted to ALL the active files. Sometimes, this would not be done.

(a)In one case, a student changed into the computer science major in mid-term, and I was assigned as her advisor. This was duly recorded in the current term's file; however, the file for the new term had already been created, containing her old advisor's name, and this was not changed.

(b)As a result, I got her grade report for one term, but the next term the grade report was sent to her old advisor. (In pre-web days, grade reports were mailed to the student and the advisor). We caught this, and the file was updated; but not before the outdated information had been propagated into yet another term's file.

(c)It took multiple terms before all the records agreed that I was this student's advisor. In one case, she was sent back from registration because my signature was on her card and the computer said someone else was her advisor (a year after she had changed majors)! (Students registered using a paper form at this time.)

d) Data isolation problems: it is not easy in such a system to pull together a report containing all the information stored on one particular entity, since it is scattered over many files, each with a distinctive format.

e) Atomicity problems: Suppose a bank customer uses an ATM to transfer $100 from his/her checking account to savings. This involves a program debiting the checking account for $100 and crediting the savings account with $100. Now suppose the bank's computer crashes in the middle of the operation.

(1)If the debit occurs first, then the computer crashes, the customer might be out $100.

(2)OTOH, if the credit is done first, the customer might gain $100 at the bank's expense.

(3)To prevent this kind of problem, we must somehow ensure that the transfer transaction is ATOMIC - either the entire transaction occurs, or nothing occurs. Each program that changes the database must deal with these issues.

f) Concurrency problems: if the data is contained on a multi-user system, it may be that two different users might access the same data item simultaneously.

(1)If both are trying to update it, inconsistencies could result.

Example: in a bank account system, if a customer is depositing money at an ATM at the same time that an EFT program is transferring money from the checking account to savings, the following scenarios may occur. (Assume that the customer has $1000 in their checking account and deposits $200 more at the same moment the EFT program is transferring $100 from checking to savings):

Scenario 1:
ATM program reads $1000 balance
EFT program reads $1000 balance
ATM program adds $200 and writes back the sum ($1200)
EFT program subtracts $100 and writes back the difference ($900)

Final balance is $900, not $1100 as it should be. (Customer is mad!)

Scenario 2:
ATM program reads $1000 balance
EFT program reads $1000 balance
EFT program subtracts $100 and writes back the difference ($900)
ATM program adds $200 and writes back the um ($1200)

Final balance is \$1200, not \$1100 as it should be.
(Customer is happy, but bank is mad!)

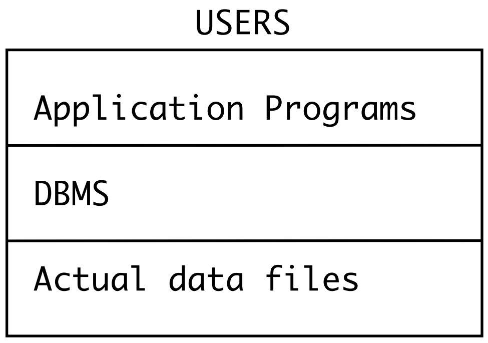
(2)To prevent such inconsistencies, each program modifying shareable data must be aware of other programs that might access the same item.

5. Actually, there is one place today where the idea of an application "owning" its files is very obvious. What is it?

   ASK

   Mobile devices such as cell phones!
   (Of course, the problems we noted above are generally not issues in this context, and some files such as pictures can be shared by multiple applications.)

D. In contrast, a database management system approach breaks the tight coupling between application programs and data, by putting a software layer in between:



PROJECT

1. Application programs that need data do not get it directly from the files where it is stored, but rather from the DBMS, which in turn gets it from the file. Application programs are not allowed to access the data directly.

2. In addition to the data itself, the database (collection of files) also contains META-DATA: data about the data.

a) This takes the form of a data dictionary, which contains at least two things for each data item in the database:

(1)A standard name for the data item which application programs use when they want to access it - e.g.

Customer.name, Customer.address, Customer.street, Customer.city, Customer.state, Customer.ZIP, Customer.YTD-Sales, Customer.discount-percent, Customer.mail-code.

(2)Where the data item is stored (what file it is in, and what field in the file), so that the DBMS can locate it.

(3): DEMO

```
docker exec -itu db2inst1 db2 bash
db2 -t
connect to intro   (connects as db2inst1_
list tables;
describe table accounts;
```

b) The data dictionary often contains other information as well. For example, it may contain:

(1)Security constraints: rules as to who is allowed to examine or update a given data item.

(a)In a file processing system, security must be done on a file by file basis: any user having read/write access to a file has read/write access to all the fields in it

(b)In a DBMS system, security can be applied item by item. For example, the mailing list application might have access to a customer's name, address, and mailing code, but not his discount percentage - even though all these items may be stored in the same file. The DBMS can enforce these security constraints, since all accesses to the data go through it.

(c) DEMO: In original window from above

```
select * from accounts;
```

Open another terminal window

Start db2 with `-t`

```
connect to intro user aardvark;
set schema db2inst1
select * from accounts;
select * from own_accounts;
```

(a view - we will discuss his a bit more later, but notice how we restrict access to the accounts one owns)

```
select * from account_owners;
```

(another view - this time we restrict access to "public" columns)

```
update account_owners
 set address = 'KOSC 242'
 where owner = 'Aardvark';
select * from account_owners;
```

Repeat in original window: `select * from accounts;`

(Views can allow controlled updating)

(2) Integrity constraints: often, the values of certain items in a database are logically constrained to only certain possibilities.

Example: a grade field in the Gordon registration system may only contain A, A-, B+ ... D-, NC, I or W. Any other value (e.g. Z) is meaningless. It is important for software that modifies such an item to ensure that the new value obeys the appropriate constraints.

(a) Under a file-processing approach, this is difficult since each program that accesses the data must know and apply the constraints. The problem becomes especially severe if a new constraint must be added or an existing one altered: every program accessing the data must be modified to the new rules.

(b)Under a DBMS approach, the data dictionary entry for the item can contain constraint information which the DBMS software can check whenever the item is changed, since all changes to the item are done through the DBMS.

(c)DEMO: In the window connected as db2inst1

```
update accounts
 set checking_balance = checking_balance + 100
 where owner = 'Aardvark';
select * from accounts;
update accounts
 set checking_balance = checking_balance - 200
 where owner = 'Aardvark';
```

(d)PROJECT/REFER TO ON CANVAS - SQL used to create this database - discuss. (In appendix to these notes)

3. The DBMS can ensure atomicity of transactions by using one of several approaches we will consider later in the course.

4. The DBMS can allow multi-user access to the data by managing accesses in such a way as to prevent inconsistencies. For example, an application that reads a data item may be required to tell the DBMS that it intends to update the item, in which case the DBMS will lock that value until the update is complete.

   Thus, in our ATM/EFT scenario, once the ATM application read the checking account balance, that item would be locked until the ATM update was complete, and the EFT application would be made to wait until the lock is released.

   a) In a file-processing system, every program that accesses a shared file needs to be aware of all the other possible accesses to that file. (Or - and more typically - files are locked so that only one program can be modifying a given file at a time.)

   b) A DBMS can manage concurrent access to files.

      DEMO: start two db2inst1 connections to intro using

      ```
      db2 -t +c  -- +c important for concurrency demo
      ```

Consider the following series of operations, which might be used to effect a transfer of money from one account to another. Clearly, we don't want someone else to be able to see the balances between these two operations, lest he/she mistakenly believe that 'Aardvark' has $100 more than he really does

```
update accounts
 set checking_balance = checking_balance + 100
 where owner = 'Aardvark';
update accounts
 set savings_balance = savings_balance - 100
 where owner = 'Aardvark';
```

Issue the first `update` from one window, then try `select * from accounts;` from the other.

Note how it sees the old values. Now finish the transaction and `commit` it - note how the access attempt can now "see" the updated balances

(Note: normally db2 treats each statement as a transaction; issuing `+c` at startup caused it to require an explicit `commit` to end a transaction.

5. DBMS's also often make it easier for users to get at the data in an ad hoc way.

   a) Under a file processing approach, any access to data requires a program to be written for that purpose.

      For example, the get a report of total YTD sales for all customers receiving the "F" mailings, a program would have to be written containing:

      - The definition of the record layout.
      - Code to open and close the file.
      - A loop like the following:

```
sum = 0.0;

for each record in the file
  if (mail_code == 'F')
   sum += ytd_sales;
```

If no program has been written to generate a given type of report, then someone who needs that type of report must either do without or be willing to have a programmer paid to write it (and be willing to wait until he/she can finish the program!)

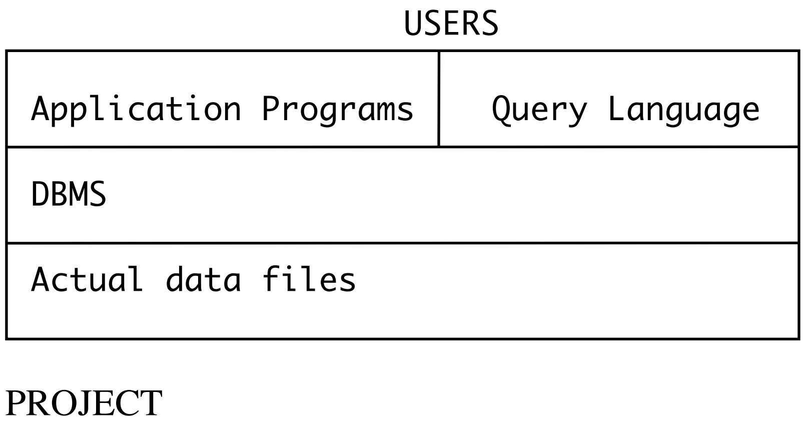
b) Most DBMS's also include a QUERY LANGUAGE which allows a moderately sophisticated user to get at information in the data base directly, without going through an application program.

Example: A DBMS that supports the SQL query language would allow an interactive user to get an answer to the above question by typing:

```
select sum(ytd_sales)
    from customer
    where mail_code = 'F';
```

(1)Such queries are possible because the data dictionary is able to provide a translation between item names such as ytd_sales and mail_code and actual physical locations in the database.

(2)Thus, our picture becomes:



Thus, our DBMS has two interfaces: one for application programs (which may call the DBMS using the regular procedure call mechanism of the language they are written in), and one for direct access by end users, using a query language.

(In fact, a personal DBMS may have only the latter interface.)

6. We have seen that putting a DBMS software layer between the data and users of the data has many advantages in terms of eliminating redundancy and inconsistency while facilitating security, integrity, multi-user access and end-user queries. However, there is a price tag on this: the additional layer of software can result in a performance penalty:

   a) At least, there is the additional processing overhead each application incurs by going through a software layer to get at the data it needs, rather than getting it directly.

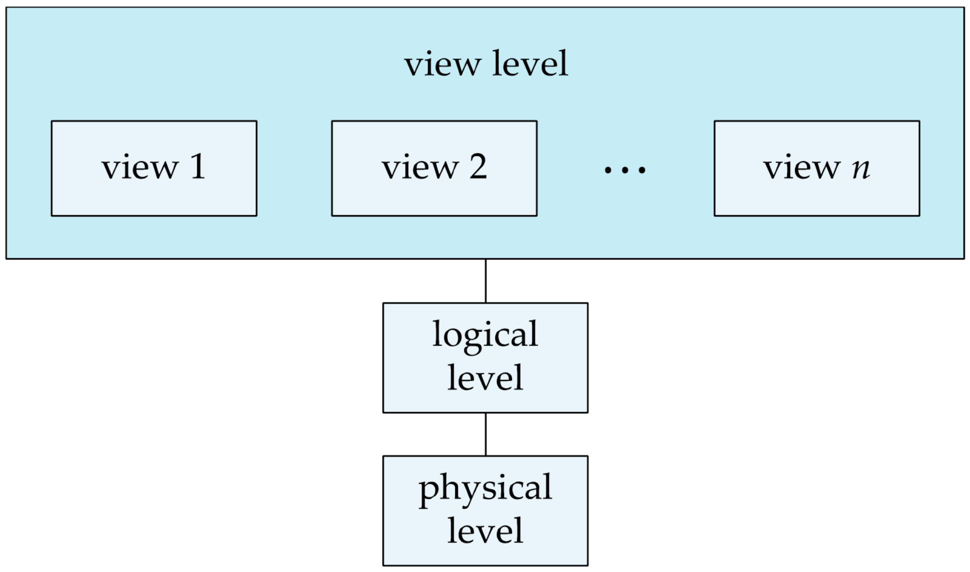
   b) If the application software "knows" how the data is stored physically, it may be able to arrange its accesses to the data in an optimum way in terms of processing efficiency. The DBMS level deprives the application software of this knowledge.

II. The Structure of Database Management Systems

A. Levels of Abstraction.

A DBMS provides three different ways of looking at the data - three levels of abstraction, each of which builds on the level below it:

An architecture for a database system



PROJECT Figure 1.2

1. At the physical level, data is stored in one or more files, and may be stored on the system or elsewhere such as in "the cloud"

2. The logical level represents the "big picture" of the overall database. At this level, data is thought of in terms of objects and relationships between those objects. Details of the physical storage of the data are abstracted out.

3. At the view level, each user of the data (application program or interactive user using a query language) can be presented with the relevant portion of the database in an appropriate way. There is one logical (conceptual) description of the database, but there may be many views.

   a) Often, a view is a subset of the information on one particular object.

      Example: in the Gordon database, information about students includes basic information such as name and id, academic information, chapel information, financial information etc. But different offices on campus see only subsets of this - e.g. most offices do not have access to the financial information, while the finance office does not need access to academic and chapel information.

   b) A view may not just be strictly a subset of the logical data on an object. It may also include virtual fields.

      Example: The total of current outstanding bills owed by all students is an item of information that is of interest to upper management in the college, as is information about the number of students whose payments are overdue, etc. This kind of information should appear in one of their views of the database. At any time, this figure is computable by combining information (typically multiple items) in all the student records; thus, it need not be represented by a data item at the physical level, but may be computed as needed. However, the view presented to users needing this data may make it look as if such a field exists; but when they try to access it the DBMS converts the access request into the appropriate computation in a way that is transparent to the user.

4. One very important goal of a DBMS is DATA INDEPENDENCE: changes at one level of the database should be possible without in any way affecting higher levels.

   a) Physical data independence: changes at the physical level should not affect the logical or view levels.

      Example: a decision is made to change to incorporate some index structure to improve performance. This change should not be visible at the logical or view levels.

   b) Logical data independence: changes at the logical level should not affect the view level.

      Example: a decision is made to add a new item of information to data stored about an object at the logical level. This must, of course, be reflected at the physical level, but should not affect any views other than those for which the new item of information was added.

B. Data Models

1. Historically, there have been five major kinds of logical model used in actual DBMS's - listed here historically.

   a) The hierarchical model (largely obsolete, but still used in some legacy systems -we will not look at)

   b) The network model (largely obsolete, but still used in some legacy systems -we will not look at)

   c) The relational model (the dominant model, and the focus of much of the course).

(1)A relational database stores data in tables as discussed in the book chapter. We will discuss this more extensively in subsequent lectures.

(2)In the context of this model we will spend significant time on Structured Query Language (variously called Seqel or SQL) which is the dominant language for defining and accessing relational databases and is also used in modified form with later non-relational models.

d) Various object-based models (we will look at briefly)

e) Various semi-structured models used in situations where performance is critical (we will look at briefly)

2. There is also a logical model known as the entity-relationship model which is useful for database design but is not directly used as the basis of an actual system.

3. Note relationship to the course syllabus: the first half of the course will primarily discuss a logical model (the relational model) and the use of the entity-relationship model as a tool for designing a database. We will also say a bit about object-based and semi-structured models later in the course.

C. Organization of a DBMS - A DBMS has three major components, each of which we will spend some time discussing in the second half of the course.

1. A storage manager that is responsible for physically storing the data on whatever storage medium is being used.
2. A query processor that handles requests to access data coming from users either directly or through an application program.

3. A transaction manager whose purpose we will discuss next.

D. The Transaction Concept

1. A key concept in DBMS design is the notion of a TRANSACTION - a unit of work that has four key properties (sometimes called the ACID properties) These were discussed in CPS221, but we will discuss them more thoroughly later in this course.

   What are the ACID properties of a transaction?

   ASK

   a) It is <u>A</u>tomic - either it happens in its entirely, or nothing of it happens - e.g. if a customer is doing a funds transfer at an ATM and something crashes mid-transaction, both accounts will reflect the "before" balances - it won't be the case that one has been debited but the other has not been credited or vice-versa.

   b) It is <u>C</u>onsistent - if the database is consistent before the transaction is started, it is consistent when the transaction is complete. (Though the database may become momentarily inconsistent during the execution of a transaction - e.g. during a transfer of funds transaction at a bank, there will be a moment when the sum total of balances in the customer's account is either too high or too low by the amount of the transfer.)

   c) It is <u>I</u>solated - other transactions do not "see" any momentary inconsistencies introduced the the transaction.

   d) It is <u>D</u>urable - once it has been done, its effect is guaranteed to remain, even in the case of hardware failures.

2. One thing that distinguishes "industrial strength" DBMS's from less capable DBMS's is the extent to which they support ACID transactions. (That's one reason why we're using db2 in this course.)

3. We will spend a some time in the second half of the course discussing how ACID transactions can be supported.

4. As we shall see, supporting ACID transactions entails considerable complexity, which in turn has an impact on overall performance. For this reason, some systems (e.g. many-structured ones and sometimes others) sacrifice ensuring ACID transactions for the sake of performance. We will discuss this toward the end of the course.

E. We will also look at some issues related to other data models and the use of databases to support business decisions

III. Distribute/Go Over Design Project

Appendix - SQL for demonstration database creation

```
create table accounts(
     owner char(20),
     address char(30),
     checking_balance decimal
          constraint positive_checking
               check (checking_balance > 0),
     savings_balance decimal
          constraint non_negative_savings
               check (savings_balance >= 0));

create view account_owners as
     select owner, address
          from accounts;

create view own_accounts as
     select *
          from accounts
          where upper(owner) = user;

grant select on account_owners to public;

grant update(address) on account_owners to public;

grant select on own_accounts to public;

insert into accounts values(
     'Aardvark',
     'Jenks Sub-basement',
     100,
     1000);
insert into accounts values(
     'Zebra',
     'Stoneham Zoo',
     50,
     0);
```