

Lecture 2. - Database System Concepts and Architecture

November 18, 2010

1 Data models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by most database users. A *data model* — a collection of concepts that can be used to describe the structure of a database — provides the necessary means to achieve this abstraction. By *structure of a database* we mean the data types, relationships, and constraints that should hold on the data. Most data models also include a set of basic operations for specifying retrievals and updates on the database.

1.1 Categories of Data Models

Many data models have been proposed, and we can categorize them according to the types of concepts they use to describe the database structure. *Conceptual (high-level) data models* provide concepts that are close to the way many end users perceive data, whereas *physical (low-level) data models* provide concepts that describe the details of how data is stored in the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users. Between these two extremes is a class of *representational (implementation) data models*, which provide concepts that may be understood by end users but that are not too far from the way data is organized within the computer. Representational data models hide some details of data storage but can be implemented on a computer system in a direct way.

Physical Data Models describes how data is stored in the computer by representing information such as stored record formats, record orderings, and access paths. An *access path* is a structure that makes the search for particular database records efficient.

Representational Data Models are the models used most frequently in traditional commercial DBMSs, and they include the widely-used relational data model, object data model and some old models like network data model or hierarchical data model.

Conceptual Data Models use concepts such as entities, attributes, and relationships. An entity represents a real-world object or concept, such as an employee or a project that is described in the database. An attribute - represents some property of interest that further describes an entity, such as the employee's name or salary. A relationship — among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project.

1.2 Schemas, Instances, and Database State

In any data model it is important to distinguish between the description of the database and the database itself. The description of a database is called the *database schema*, which is specified during database design and is not expected to change frequently. Most data models have certain conventions for displaying the schemas as diagrams. A displayed schema is called a *schema diagram*. Figure 1 shows a sample schema diagram. Diagram displays only the structure of each record type but not the actual instances of records. Schema diagram displays only some aspect of a schema, such as the names of record types and data items, and some types of constraints. Figure 1 shows neither the data type of each data item nor the relationships among the various files.

Figure 1: Sample schema diagram

SUPPLIER

SuppNumber	SuppName	Status	City
------------	----------	--------	------

PART

PartNumber	PartName	Color	Weight	City
------------	----------	-------	--------	------

DELIVERY

SuppNumber	PartNumber	Quantity
------------	------------	----------

The actual data in a database may change quite frequently; for example, the database shown in Figure 1 changes every time we add a supplier or enter a new color for a part. The data in the database in a particular moment in time is called *database state or snapshot*. It is also called the current set of occurrences or instances in the database. The actual data in a database may change quite frequently. Every time we insert or delete a record, or change the value of a data item in a record, we change a one state of the database into another state.

The distinction between database schema and database state is very important. When we define a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first loaded with the initial data. From then, on every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*. The DBMS is partly responsible for ensuring that every state of the database is a *valid state* — that is, a state that satisfies the structure and constraints specified in the schema. Example below shows current state of the database.

2 DBMS Architecture

Three important characteristics of the database approach are insulation of programs and data, support of multiple user views and use of a catalog to store the database description. Let's specify architecture for database systems, called the three-schema architecture, which was proposed to help achieve and visualize these characteristics.

Figure 2: Sample current state of database

SUPPLIER

SuppNumber	SuppName	Status	City
S1	Jones	20	New York
S2	Black	30	Paris
S3	Smith	10	London

PART

PartNumber	PartName	Color	Weight	City
P1	Desk	Blue	20	London
P2	Monitor	Red	10	Paris

DELIVERY

SuppNumber	PartNumber	Quantity
S1	P1	200

2.1 The Three-Schema Architecture

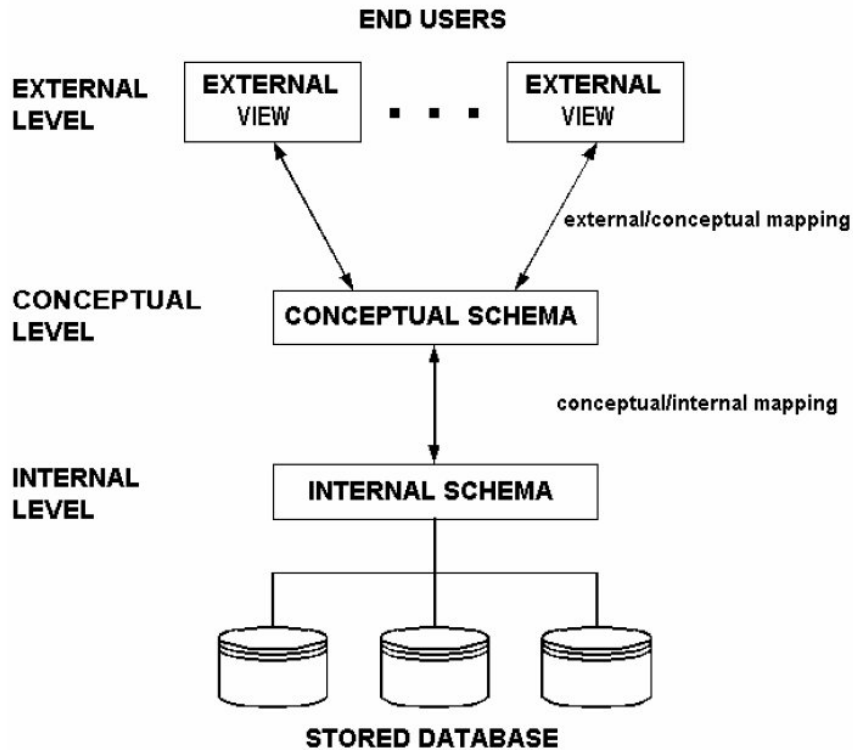
The goal of the three-schema architecture is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

- *The internal level* has an internal schema, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
- *The conceptual level* has a conceptual schema, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
- *The external or view level* includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support, the threeschema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Notice that the three schemas are only descriptions of data. The only data that actually exists is at the physical level. In a DBMS based on the three-schema architecture, each user group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is database retrieval, the data extracted from the stored database

Figure 3: Illustrating the three-schema architecture



must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called *mappings*. These mappings may be time-consuming, so some DBMSs — especially those that are meant to support small databases — do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2 Data Independence

The three-schema architecture can be used to explain the concept of *data independence*, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

- *Logical data independence* is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization.

Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

- *Physical data independence* is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized — for example, by creating additional access structures — to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

3 Database Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first order of the day is to specify conceptual and internal schemas for the database and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the *data definition language* (DDL), is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the *storage definition language* (SDL), is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. For true three-schema architecture, we would need a third language, the *view definition language* (VDL), to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a *data manipulation language* (DML) for these purposes.

In current DBMSs, the preceding types of languages are usually not considered distinct languages, rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, and it is usually utilized by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language, which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification and schema evolution. The SDL was a component in earlier versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A *high-level* or *nonprocedural DML* can be used on its own to specify complex database operations in a concise manner. Many DBMSs allow high-level DML statements either to be entered interactively from a terminal (or monitor) or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a pre-compiler and processed by the DBMS. A *low-level* or *procedural DML* must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Hence, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. High-level

DMLs, such as SQL, can specify and retrieve many records in a single DML statement. A query in a high-level DML often specifies which data to retrieve rather than how to retrieve it.

Whenever DML commands, whether high-level or low-level, are embedded in a general-purpose programming language, that language is called the *host language* and the DML is called the *data sublanguage*. On the other hand, a high-level DML used in a stand-alone interactive manner is called a *query language*. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are user-friendly interfaces for interacting with the database. These can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

4 Database Interfaces

User-friendly interfaces provided by a DBMS may include the following.

4.1 Menu-Based Interfaces for Browsing

These interfaces present the user with lists of options, called *menus*, that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step by step by picking options from a menu that is displayed by the system. Pull-down menus are becoming a very popular technique in window-based user interfaces. They are often used in browsing interfaces, which allow a user to look through the contents of a database in an exploratory and unstructured manner.

4.2 Forms-Based Interfaces

A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have forms specification languages, special languages that help programmers specify such forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

4.3 Graphical User Interfaces

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a pointing device, such as a mouse, to pick certain parts of the displayed schema diagram.

4.4 Natural Language Interfaces

These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema", which is similar to the database conceptual schema. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing. Otherwise, a dialogue is started with the user to clarify the request.

4.5 Interfaces for Parametric Users

Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

4.6 Interfaces for the DBA

Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures.

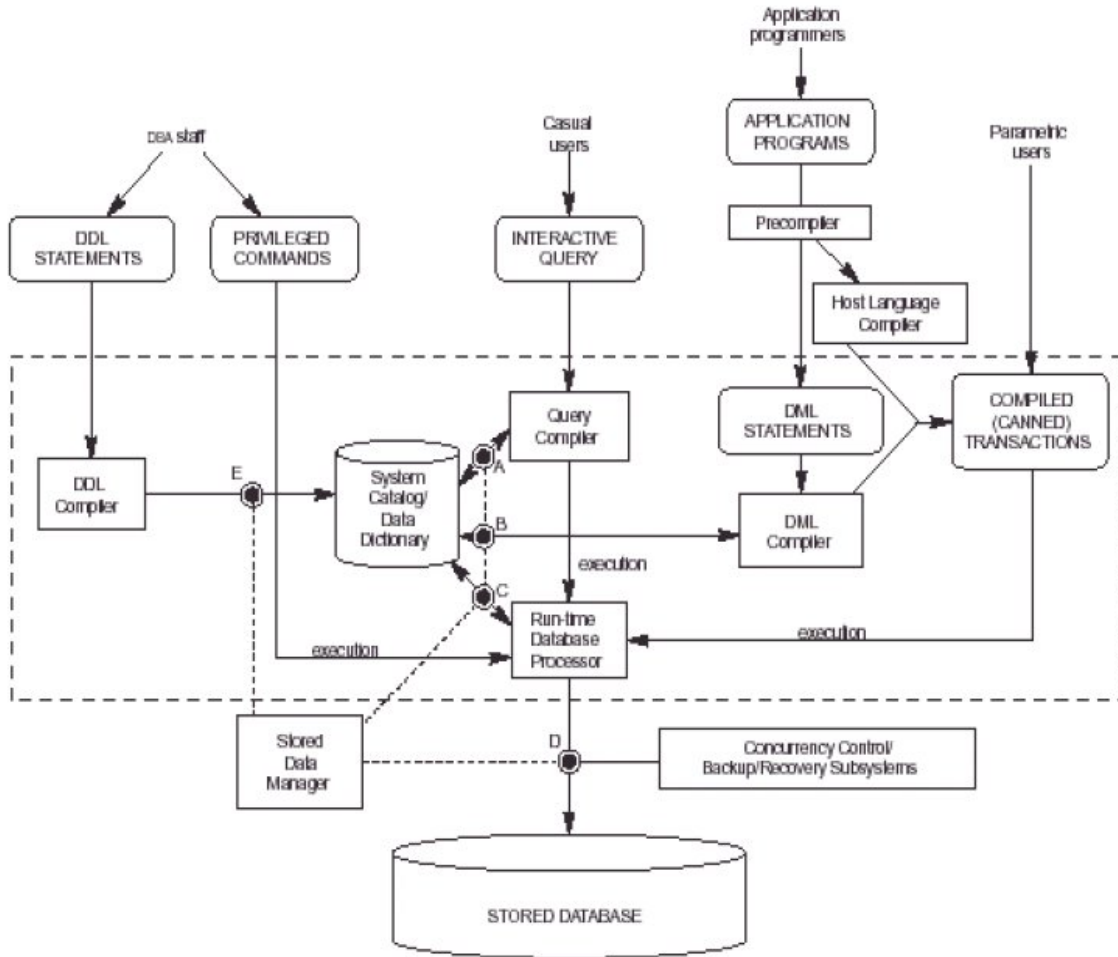
5 DBMS Component Modules

Figure 4 illustrates in a simplified form, the typical DBMS components. The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the operating system, which schedules disk input/output. A higher-level *stored data manager* module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The dotted lines and circles marked ABCD and E illustrate accesses that are under the control of this stored data manager. The stored data manager may use basic operating system services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs.

The *DDL compiler* processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up the catalog information as needed.

The *run-time database processor* handles database accesses at run time; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager. The *query compiler* handles high-level queries that are entered interactively. It parses, analyzes, and

Figure 4: Typical component modules of DBMS



compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The *pre-compiler* extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.

Figure 4 is not meant to describe a specific DBMS, rather it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses to the database or to the catalog — are needed. If the computer system is shared by many users, the operating system will schedule DBMS disk access requests and DBMS processing along with other processes. The DBMS also interfaces with compilers for general-purpose host programming languages. User-friendly interfaces to the DBMS can be provided to help any of the user types shown in Figure 4 to specify their requests.

6 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have database utilities that help the DBA in managing the database system. Common utilities have the following types of functions:

- *Loading*: A loading utility is used to load existing data files — such as text files or sequential files — into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. Transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called *conversion tools*.
- *Backup*: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but it saves space.
- *File reorganization*: This utility can be used to reorganize a database file into a different file organization to improve performance.
- *Performance monitoring*: Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, and performing other functions.

7 Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the *data model* on which the DBMS is based. The two types of data models used in many current commercial DBMSs are the *relational data model* and the *object data model*. Many legacy applications still run on database systems based on the *hierarchical* and *network data models*. The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed, in object databases. This has led to a new class of DBMSs that are being called *object-relational DBMSs*. We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

The second criterion used to classify DBMSs is the *number of users* supported by the system. *Single-user systems* support only one user at a time and are mostly used with personal computers. *Multi-user systems*, which include the majority of DBMSs, support multiple users concurrently.

A third criterion is the *number of sites* over which the database is distributed. A DBMS is *centralized* if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A *distributed DBMS* (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer

network. *Homogeneous DDBMSs* use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under heterogeneous DBMSs. This leads to a *federated DBMS* (or *multi-database system*), where the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use client-server architecture.

A fourth criterion is the *cost* of the DBMS. The majority of DBMS packages cost between \$10,000 and \$100,000. Single-user low-end systems that work with microcomputers cost between \$100 and \$3000. At the other end, a few elaborate packages cost more than \$100,000.

We can also classify a DBMS on the basis of the *types of access path* options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be *general-purpose* or *special-purpose*. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of *on-line transaction processing* (OLTP) systems, which must support a large number of concurrent transactions without imposing excessive delays.