Lecture 1: Introduction to Databases

A historical perspective

From the earliest days of computers, storing and manipulating data have been a major application focus. The first general-purpose DBMS was designed by Charles Bachman at General Electric in the early 1960s and was called the Integrated Data Store. It formed the basis for the network data model, which was standardized by the Conference on Data Systems Languages (CODASYL) and strongly influenced database systems through the 1960s. Bachman was the first recipient of ACM's Turing Award (the computer science equivalent of a Nobel Prize) for work in the database area; he received the award in 1973.

In the late 1960s, IBM developed the Information Management System (IMS) DBMS, used even today in many major installations. IMS formed the basis for an alternative data representation framework called the hierarchical data model. The SABRE system for making airline reservations was jointly developed by American Airlines and IBM around the same time, and it allowed several people to access the same data through a computer network. Interestingly, today the same SABRE system is used to power popular Web-based travel services such as Travelocity!

In 1970, Edgar Codd, at IBM's San Jose Research Laboratory, proposed a new data representation framework called the relational data model. This proved to be a watershed in the development of database systems: it sparked rapid development of several DBMSs based on the relational model, along with a rich body of theoretical results that placed the field on a firm foundation. Codd won the 1981 Turing Award for his seminal work. Database systems matured as an academic discipline, and the popularity of relational DBMSs changed the commercial landscape. Their benefits were widely recognized, and the use of DBMSs for managing corporate data became standard practice.

In the 1980s, the relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use. The SQL query language for relational databases, developed as part of IBM's System R project, is now the standard query language. SQL was standardized in the late 1980s, and the current standard, SQL-92, was adopted by the American National Standards Institute (ANSI) and International Standards Organization (ISO). Arguably, the most widely used form of concurrent programming is the concurrent execution of database programs (called transactions). Users write programs as if they are to be run by themselves, and the responsibility for running them concurrently is given to the DBMS. James Gray won the 1999 Turing award for his contributions to the field of transaction management in a DBMS.

In the late 1980s and the 1990s, advances have been made in many areas of database systems. Considerable research has been carried out into more powerful query languages and richer data models, and there has been a big emphasis on supporting complex analysis of data from all parts of an enterprise. Several vendors (e.g., IBM's DB2, Oracle 8, Informix UDS) have extended their systems with the ability to store new data types such as images and text, and with the ability to ask more complex queries. Specialized systems have been developed by numerous vendors for creating data warehouses, consolidating data from several databases, and for carrying out specialized analysis.

An interesting phenomenon is the emergence of several enterprise resource planning (ERP) and management resource planning (MRP) packages, which add a substantial layer of application-oriented features on top of a DBMS. Widely used packages include systems from Baan, Oracle, PeopleSoft, SAP, and Siebel. These packages identify a set of common tasks (e.g., inventory management, human resources planning, financial

analysis) encountered by a large number of organizations and provide a general application layer to carry out these tasks. The data is stored in a relational DBMS, and the application layer can be customized to different companies, leading to lower overall costs for the companies, compared to the cost of building the application layer from scratch.

Most significantly, perhaps, DBMSs have entered the Internet Age. While the first generation of Web sites stored their data exclusively in operating systems files, the use of a DBMS to store data that is accessed through a Web browser is becoming widespread. Queries are generated through Web-accessible forms and answers are formatted using a markup language such as HTML, in order to be easily displayed in a browser. All the database vendors are adding features to their DBMS aimed at making it more suitable for deployment over the Internet.

Database management continues to gain importance as more and more data is brought on-line, and made ever more accessible through computer networking. Today the field is being driven by exciting visions such as multimedia databases, interactive video, digital libraries, a host of scientific projects such as the human genome mapping effort and NASA's Earth Observation System project, and the desire of companies to consolidate their decision-making processes and mine their data repositories for useful information about their businesses. Commercially, database management systems represent one of the largest and most vigorous market segments. Thus the study of database systems could prove to be richly rewarding in more ways than one!

Database Management System (DBMS)

A DBMS is a piece of software that is designed to make the preceding tasks easier. By storing data in a DBMS, rather than as a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner. As the volume of data and the number of users grow hundreds of gigabytes of data and thousands of users are common in current corporate databases DBMS support becomes indispensable.

Advantages of DBMS

Using a DBMS to manage data has many advantages:

- **Data independence:** Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.
- **Efficient data access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data integrity and security:** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce access controls that govern what data is visible to different classes of users.
- **Data administration:** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals, who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.
- **Concurrent access and crash recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

- **Reduced application development time:** Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application.

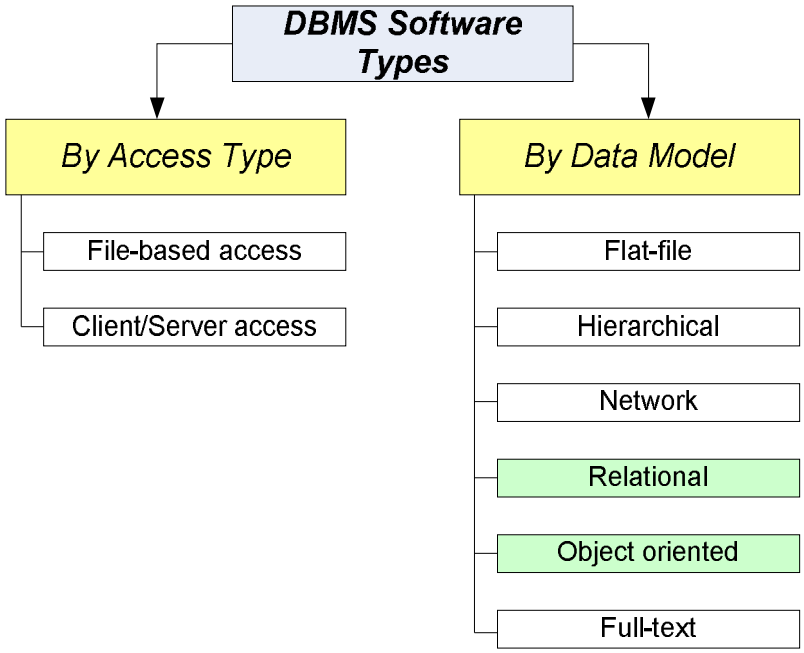
Describing and Storing Data in DBMS

The user of a DBMS is ultimately concerned with some real-world enterprise, and the data to be stored describes various aspects of this enterprise. For example, there are students, faculty, and courses in a university, and the data in a university database describes these entities and their relationships.

A data model is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the relational data model.

While the data model of the DBMS hides many details, it is nonetheless closer to how the DBMS stores data than to how a user thinks about the underlying enterprise. A semantic data model is a more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise. These models contain a wide variety of constructs that help describe a real application scenario. A DBMS is not intended to support all these constructs directly; it is typically built around a data model with just a few basic constructs, such as the relational model. A database design in terms of a semantic model serves as a useful starting point and is subsequently translated into a database design in terms of the data model the DBMS actually supports.

DBMS Types

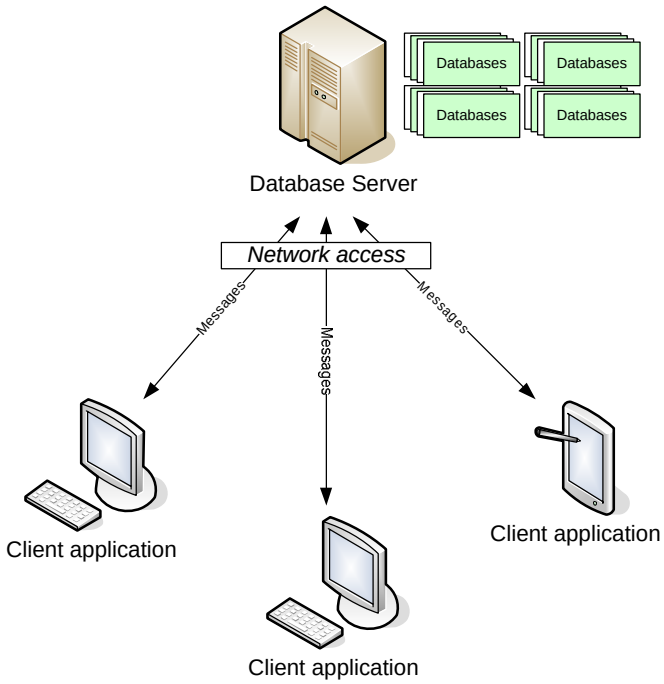


By Access-Type

DBMS using **file-based access** are using files stored locally or on a predefined network location.

The term **client/server** was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. The client/server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing.

A client is defined as a requester of services and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration.



By Data Model

A **flat-file database** is described by a very simple database model, where all the information is stored in a plain text file, one database record per line. Each record is divided into fields using delimiters or at fixed column positions. The data is "flat", as in a sheet of paper, as compared to a more complex model such as a relational database.

Hierarchical database stores related information in terms of pre-defined categorical relationships in a "tree-like" fashion. Information is traced from a major group, to a subgroup, and to further subgroups. Much like tracing a family tree, data can be traced through parents along paths through the hierarchy. Users must keep track of the hierarchical structure in order to make use of the data. The relational database provides an alternative means of organizing datasets.

A **network model database** management system has a more flexible structure than the hierarchical model or relational model, but pays for it in processing time and specialization of types. Some object-oriented database systems use a general network model, but most have some hierarchical limitations.

Relational database has information structure that stores data in tables that can be linked to each other for cross-referencing - for instance, a table that presents vocabulary items and their grammatical features that is linked to a table that presents grammatical features and their definitions. This format prevents the duplication of data and is the preferred method of storing complex sets of information. The data is stored in such a way that it can be added to, and used independently of, all other data stored in the database. Users can query a relational database without knowing how the information has been

organized. Although relational databases have the advantages of ease-of-use and analytical flexibility, their weakness can be slower retrieval speed.

Object-oriented database has structure that organizes, manipulates, and retrieves classes of objects, such as sound, video, text, and graphic files. In these databases, algorithms for processing data are integrated with the data, so that data related to each object of importance have their own associated object-oriented programs.

Full-text database is a bibliographic database which contains the complete text of the bibliographic record (such as a journal article) which is referenced in the database.

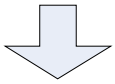
The Relational Model

The central data description construct in this model is a relation, which can be thought of as a set of records.

A description of data in terms of a data model is called a schema. In the relational model, the schema for a relation specifies its name, the name of each field (or attribute or column), and the type of each field. As an example, student information in a university database may be stored in a relation with the following schema:

Schema

Students(*sid*:**string**; *name*:**string**; *login*:**string**; *age*: **byte**: *gpa*: **real**)



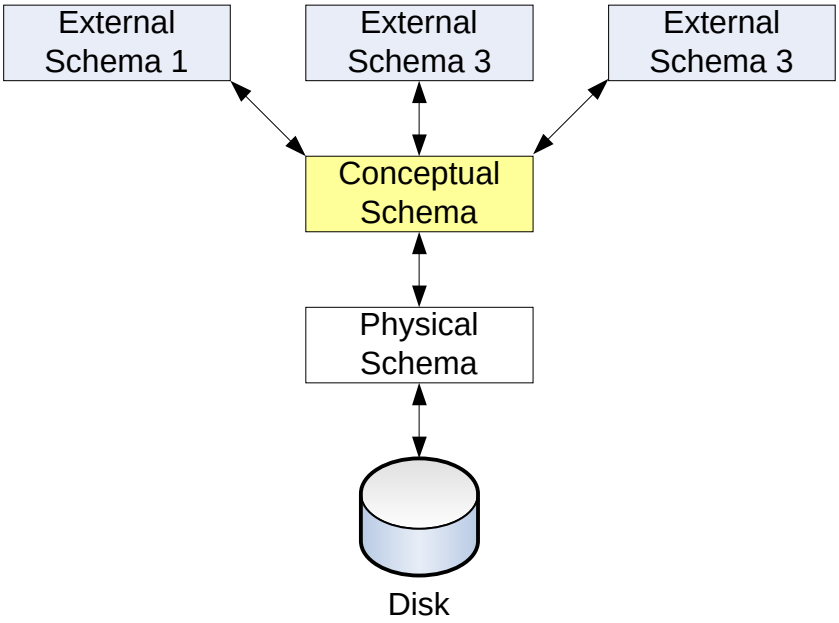
Instance

SID	Name	Login	Age	gpa
string	string	string	byte	real
1	John Hobsons	jhobs	22	4.5
2	Mary Timberly	mary	24	6.0
3	Smith Jackson	sj12	22	4.4
...	...	...	...	...

Each row in the Students relation is a record that describes a student.

Levels of Abstraction in a DBMS

The data in a DBMS is described at three levels of abstraction, as illustrated. The database description consists of a schema at each of these three levels of abstraction: the conceptual, physical, and external schemas.



A data definition language (DDL) is used to define the external and conceptual schemas. We will discuss the DDL facilities of the most widely used database language, SQL later. All DBMS vendors also support SQL commands to describe aspects of the physical schema, but these commands are not part of the SQL-92 language standard. Information about the conceptual, external, and physical schemas is stored in the system catalogs.

The Conceptual Schema

The conceptual schema (sometimes called the logical schema) describes the stored data in terms of the data model of the DBMS. In a relational DBMS, the conceptual schema describes all relations that are stored in the database. In our sample university database, these relations contain information about entities, such as students and faculty, and about relationships, such as students' enrollment in courses. All student entities can be described using records in a Students relation, as we saw earlier. In fact, each collection of entities and each collection of relationships can be described as a relation, leading to the following conceptual schema:

```
      Students(sid: string, name: string, login: string, age: integer, gpa:
real)
      Faculty(fid: string, fname: string, sal: real)
      Courses(cid: string, cname: string, credits: integer)
      Rooms(rno: integer, address: string, capacity: integer)
      Enrolled(sid: string, cid: string, grade: string)
      Teaches(fid: string, cid: string)
      Meets_In(cid: string, rno: integer, time: string)
```

The choice of relations, and the choice of fields for each relation, is not always obvious, and the process of arriving at a good conceptual schema is called **conceptual database design**.

The Physical Schema

The physical schema specifies additional storage details. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.

We must decide what file organizations to use to store the relations, and create auxiliary data structures called indexes to speed up data retrieval operations. A sample physical schema for the university database follows:

1. Store all relations as unsorted files of records. (A file in a DBMS is either a collection of records or a collection of pages, rather than a string of characters as in an operating system.)
2. Create indexes on the first column of the Students, Faculty, and Courses relations, the *sal* column of Faculty, and the capacity column of Rooms.

Decisions about the physical schema are based on an understanding of how the data is typically accessed. The process of arriving at a good physical schema is called **physical database design**.

The External Schema

External schemas, which usually are also in terms of the data model of the DBMS, allow data access to be customized (and authorized) at the level of individual users or groups of users. Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more views and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS. Rather, they are computed using a definition for the view, in terms of relations stored in the DBMS.

The external schema design is guided by end user requirements. For example, we might want to allow students to find out the names of faculty members teaching courses, as well as course enrollments. This can be done by defining the following view:

```
Courseinfo(cid: string, fname: string, enrollment: integer)
```

A user can treat a view just like a relation and ask questions about the records in the view. Even though the records in the view are not stored explicitly, they are computed as needed. We did not include *Courseinfo* in the conceptual schema because we can compute *Courseinfo* from the relations in the conceptual schema, and to store it in addition would be redundant. Such redundancy, in addition to the wasted space, could lead to inconsistencies. For example, a tuple may be inserted into the Enrolled relation, indicating that a particular student has enrolled in some course, without incrementing the value in the enrollment field of the corresponding record of *Courseinfo* (if the latter also is part of the conceptual schema and its tuples are stored in the DBMS).

Data Independence

A very important advantage of using a DBMS is that it offers data independence. That is, application programs are insulated from changes in the way the data is structured and stored. Data independence is achieved through use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.

Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema. If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. For example, suppose that the Faculty relation in our university database is replaced by the following two relations:

```
Faculty public(fid: string, fname: string, office: integer)
Faculty private(fid: string, sal: real)
```

Intuitively, some confidential information about faculty has been placed in a separate relation and information about offices has been added. The *Courseinfo* view relation can be redefined in terms of Faculty public and Faculty private, which together contain all the information in Faculty, so that a user who queries *Courseinfo* will get the same answers as before.

Thus users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called logical data independence.

In turn, the conceptual schema insulates users from changes in the physical storage of the data. This property is referred to as physical data independence. The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes. As long as the conceptual schema remains the same, we can change these storage details without altering applications. (Of course, performance might be affected by such changes.)

Queries in a DBMS

The ease with which information can be obtained from a database often determines its value to a user. In contrast to older database systems, relational database systems allow a rich class of questions to be posed easily; this feature has contributed greatly to their popularity. Consider the sample university. Here are examples of questions that a user might ask:

1. What is the name of the student with student id 123456?
2. What is the average salary of professors who teach the course with cid CS564?
3. How many students are enrolled in course CS564?
4. What fraction of students in course CS564 received a grade better than B?
5. Is any student with a GPA less than 3.0 enrolled in course CS564?

Such questions involving the data stored in a DBMS are called queries. A DBMS provides a specialized language, called the query language, in which queries can be posed. A very attractive feature of the relational model is that it supports powerful query languages. Relational calculus is a formal query language based on mathematical, logic, and queries in this language have an intuitive, precise meaning. Relational algebra is another formal query language, based on a collection of operators for manipulating relations, which is equivalent in power to the calculus.

A DBMS takes great care to evaluate queries as efficiently as possible. We discuss query optimization and evaluation later. Of course, the efficiency of query evaluation is determined to a large extent by how the data is stored physically.

Indexes can be used to speed up many queries in fact, a good choice of indexes for the underlying relations can speed up each query in the preceding list. We discuss data storage and indexing later.

A DBMS enables users to create, modify, and query data through a data manipulation language (DML). Thus, the query language is only one part of the DML, which also provides constructs to insert, delete, and modify data. We will discuss the DML features of SQL later. The DML and DDL are collectively referred to as the data sublanguage when embedded within a host language (e.g., C++ or C#).

Transaction Management

Consider a database that holds information about airline reservations. At any given instant, it is possible (and likely) that several travel agents are looking up information about available seats on various flights and making new seat reservations. When several users access (and possibly modify) a database concurrently, the DBMS must order their

requests carefully to avoid conflicts. For example, when one travel agent looks up Flight 100 on some given day and finds an empty seat, another travel agent may simultaneously be making a reservation for that seat, thereby making the information seen by the first agent obsolete.

Another example of concurrent use is a bank's database. While one user's application program is computing the total deposits, another application may transfer money from an account that the first application has just 'seen' to an account that has not yet been seen, thereby causing the total to appear larger than it should be. Clearly, such anomalies should not be allowed to occur. However, disallowing concurrent access can degrade performance.

Further, the DBMS must protect users from the effects of system failures by ensuring that all data (and the status of active applications) is restored to a consistent state when the system is restarted after a crash. For example, if a travel agent asks for a reservation to be made, and the DBMS responds saying that the reservation has been made, the reservation should not be lost if the system crashes. On the other hand, if the DBMS has not yet responded to the request, but is in the process of making the necessary changes to the data while the crash occurs, the partial changes should be undone when the system comes back up.

A transaction is any one execution of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS: Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions.

Concurrent Execution of Transactions

An important task of a DBMS is to schedule concurrent accesses to data so that each user can safely ignore the fact that others are accessing the data concurrently. The importance of this task cannot be underestimated because a database is typically shared by a large number of users, who submit their requests to the DBMS independently and simply cannot be expected to deal with arbitrary changes being made concurrently by other users. A DBMS allows users to think of their programs as if they were executing in isolation, one after the other in some order chosen by the DBMS. For example, if a program that deposits cash into an account is submitted to the DBMS at the same time as another program that debits money from the same account, either of these programs could be run first by the DBMS, but their steps will not be interleaved in such a way that they interfere with each other.

A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. A lock is a mechanism used to control access to database objects. Two kinds of locks are commonly supported by a DBMS: shared locks on an object can be held by two different transactions at the same time, but an exclusive lock on an object ensures that no other transactions hold any lock on this object.

Suppose that the following locking protocol is followed: Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data object that it needs to modify, and then releases all its locks after completing all actions. Consider two transactions T1 and T2 such that T1 wants to modify a data object and T2 wants to read the same object. Intuitively, if T1's request for an exclusive lock on the object is granted first, T2 cannot proceed until T1 releases this lock, because T2's request for a shared lock will not be granted by the DBMS until then. Thus, all of T1's actions will be completed before any of T2's actions are initiated.

Incomplete Transactions and System Crashes

Transactions can be interrupted before running to completion for a variety of reasons, e.g., a system crash. A DBMS must ensure that the changes made by such incomplete transactions are removed from the database. For example, if the DBMS is in the middle of transferring money from account A to account B, and has debited the first account but not yet credited the second when the crash occurs, the money debited from account A must be restored when the system comes back up after the crash.

To do so, the DBMS maintains a log of all writes to the database. A crucial property of the log is that each write action must be recorded in the log (on disk) before the corresponding change is reflected in the database itself otherwise, if the system crashes just after making the change in the database but before the change is recorded in the log, the DBMS would be unable to detect and undo this change. This property is called Write-Ahead Log or WAL. To ensure this property, the DBMS must be able to selectively force a page in memory to disk.

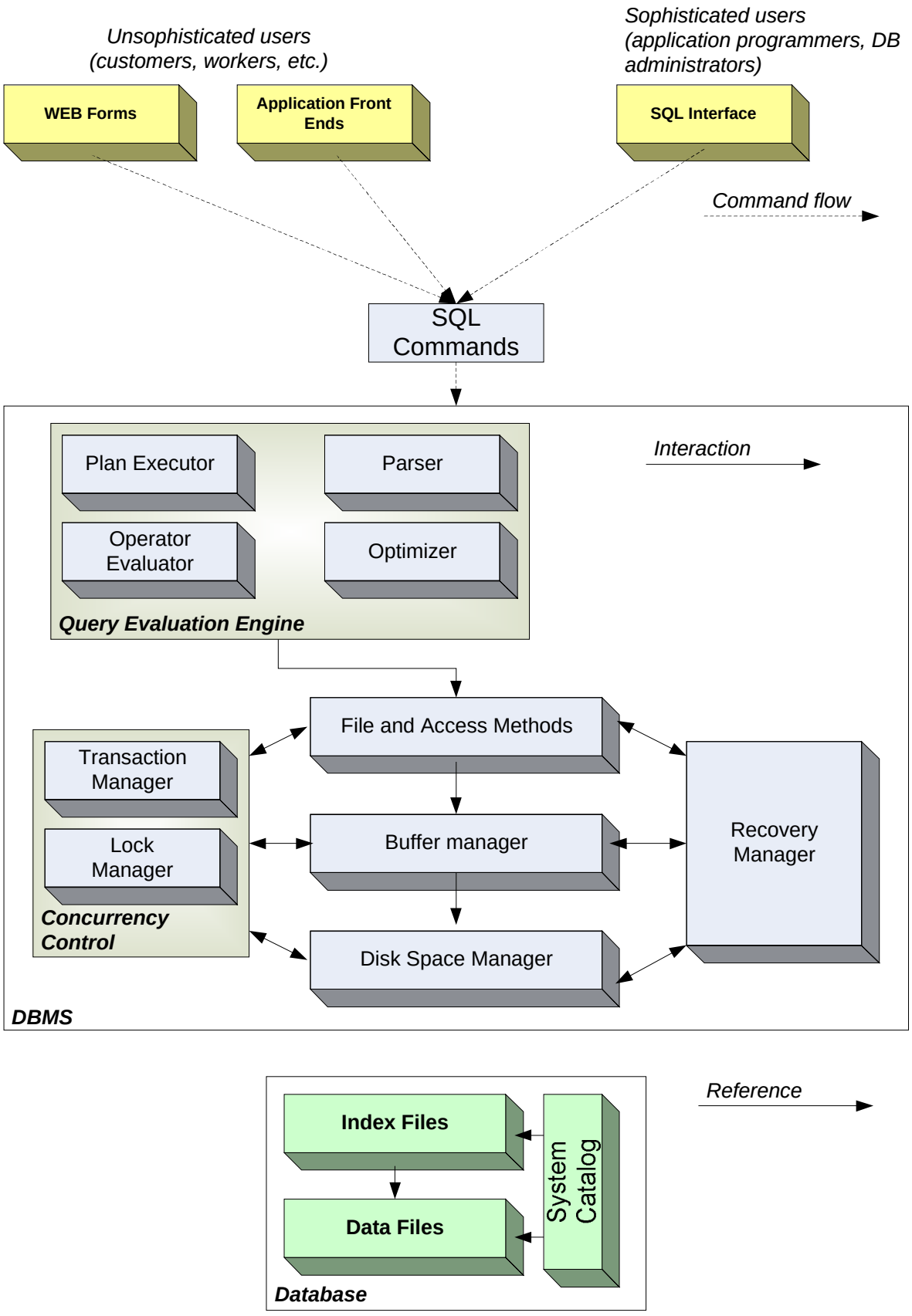
The log is also used to ensure that the changes made by a successfully completed transaction are not lost due to a system crash. Bringing the database to a consistent state after a system crash can be a slow process, since the DBMS must ensure that the effects of all transactions that completed prior to the crash are restored, and that the effects of incomplete transactions are undone. The time required to recover from a crash can be reduced by periodically forcing some information to disk; this periodic operation is called a checkpoint.

Transaction Summary

In summary, there are three points to remember with respect to DBMS support for concurrency control and recovery:

1. Every object that is read or written by a transaction is first locked in shared or exclusive mode, respectively. Placing a lock on an object restricts its availability to other transactions and thereby affects performance.
2. For efficient log maintenance, the DBMS must be able to selectively force a collection of pages in main memory to disk. Operating system support for this operation is not always satisfactory.
3. Periodic checkpointing can reduce the time needed to recover from a crash. Of course, this must be balanced against the fact that checkpointing too often slows down normal execution.

Structure of DBMS



The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in hostlanguage

application programs, e.g., Java or C# programs. We ignore these issues to concentrate on the core DBMS functionality.)

When a user issues a query, the parsed query is presented to a query optimizer, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An execution plan is a blueprint for evaluating a query, and is usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.). Relational operators serve as the building blocks for evaluating queries posed against the data.

The code that implements relational operators sits on top of the file and access methods layer. This layer includes a variety of software for supporting the concept of a file, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a heap file, or file of unordered pages, as well as indexes. In addition to keeping track of the pages in a file, this layer organizes the information within a page

The files and access methods layer code sits on top of the buffer manager, which bring pages in from disk to main memory as needed in response to read requests The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer, called the disk space manager The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the transaction manager, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the lock manager, which keeps track of requests for locks and grants locks on database objects when they become available; and the recovery manager, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components.

People Who Deal with Databases

Quite a variety of people are associated with the creation and use of databases. Obviously, there are database implementors, who build DBMS software, and end users who wish to store and use data in a DBMS. Database implementors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields. As data grows in complexity and volume, and is increasingly recognized as a major asset, the importance of maintaining it professionally in a DBMS is being widely accepted. Many end users simply use applications written by database application programmers (see below), and so require little technical knowledge about DBMS software. Of course, sophisticated users who make more extensive use of a DBMS, such as writing their own queries, require a deeper understanding of its features. In addition to end users and implementors, two other classes of people are associated with a DBMS: application programmers and database administrators (DBAs).

Database application programmers develop packages that facilitate data access for end users, who are usually not computer professionals, using the host or data languages and software tools that DBMS vendors provide. (Such tools include report writers, spreadsheets, statistical packages, etc.). Application programs should ideally access data through the external schema. It is possible to write applications that access data at a lower level, but such applications would compromise data independence.

A personal database is typically maintained by the individual who owns it and uses it. However, corporate or enterprise-wide databases are typically important enough and complex enough that the task of designing and maintaining the database is entrusted to a professional called the database administrator. The DBA is responsible for many critical tasks:

- **Design of the conceptual and physical schemas:** The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. Based on this knowledge, the DBA must design the conceptual schema (decide what relations to store) and the physical schema (decide how to store them). The DBA may also design widely used portions of the external schema, although users will probably augment this schema by creating additional views.
- **Security and authorization:** The DBA is responsible for ensuring that unauthorized data access is not permitted. In general, not everyone should be able to access all the data. In a relational DBMS, users can be granted permission to access only certain views and relations. For example, although you might allow students to find out course enrollments and who teaches a given course, you would not want students to see faculty salaries or each others' grade information. The DBA can enforce this policy by giving students permission to read only the Courseinfo view.
- **Data availability and recovery from failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible. The DBA must also work to restore the data to a consistent state. The DBMS provides software support for these functions, but the DBA is responsible for implementing procedures to back up the data periodically and to maintain logs of system activity (to facilitate recovery from a crash).
- **Database tuning:** The needs of users are likely to evolve with time. The DBA is responsible for modifying the database, in particular the conceptual and physical schemas, to ensure adequate performance as user requirements change.

Lecture 2: The Entity-Relationship Model. The Relational Model.

Entity-relationship Model

The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. In this lecture, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.

The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed and precise, description that can be implemented in a DBMS.

Overview of Database Design

The database design process can be divided into six steps. The ER model is most relevant to the first three steps:

1. **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on. Several methodologies have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.
2. **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.
3. **Logical Database Design:** We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.
4. **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.
5. **Physical Database Design:** In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.
6. **Security Design:** In this step, we identify different user groups and different roles played by various users (e.g., the development team for a product, the customer support representatives, the product manager). For each role and user group, we

must identify the parts of the database that they must be able to access and the parts of the database that they should not be allowed to access, and take steps to ensure that they can access only the necessary parts.

In general, our division of the design process into steps should be seen as a classification of the kinds of steps involved in design. Realistically, although we might begin with the six step process outlined here, a complete database design will probably require a subsequent tuning phase in which all six kinds of design steps are interleaved and repeated until the design is satisfactory. Further, we have omitted the important steps of implementing the database design, and designing and implementing the application layers that run on top of the DBMS. In practice, of course, these additional steps can lead to a rethinking of the basic database design.

The concepts and techniques that underlie a relational DBMS are clearly useful to someone who wants to implement or maintain the internals of a database system. However, it is important to recognize that serious users and DBAs must also know how a DBMS works. A good understanding of database system internals is essential for a user who wishes to take full advantage of a DBMS and design a good database; this is especially true of physical design and database tuning.

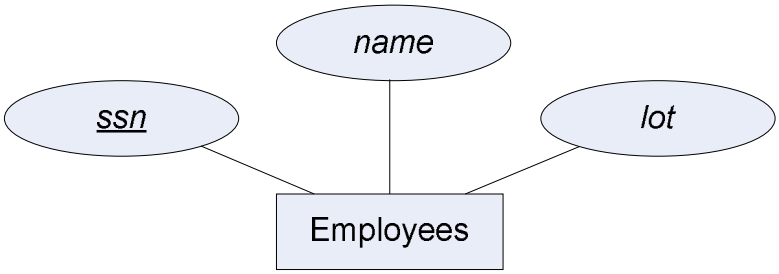
Entities, Attributes and Entities Sets

An entity is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy department. It is often useful to identify a collection of similar entities. Such a collection is called an entity set. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.

An entity is described using a set of attributes. All entities in a given entity set have the same attributes; this is essentially what we mean by similar. Our choice of attributes reflects the level of detail at which we wish to represent information about entities. For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee's address (or gender or age).

For each attribute associated with an entity set, we must identify a domain of possible values. For example, the domain associated with the attribute name of Employees might be the set of 20-character strings. As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called rating, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a key. A key is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key; if so, we designate one of them as the primary key. For now we will assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key.

An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is *ssn*.



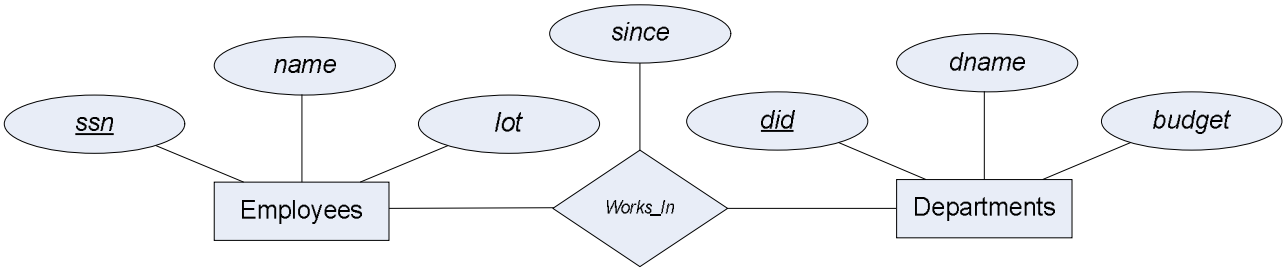
Relationships and Relationships Sets

A relationship is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department. As with entities, we may wish to collect a set of similar relationships into a relationship set.

A relationship set can be thought of as a set of *n*-tuples:

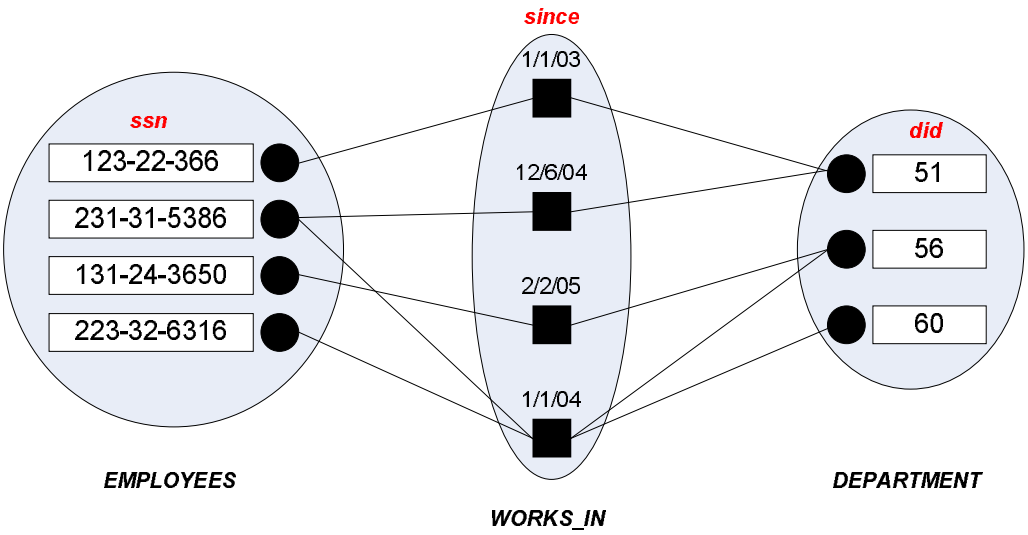
$$\{(e_1,...,e_n)|e_1 \in E_1,...,e_n \in E_n\}$$

Each *n*-tuple denotes a relationship involving n entities e_1 through en, where entity e_i is in entity set E_i. The following figure shows the relationship set *Works_In*, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

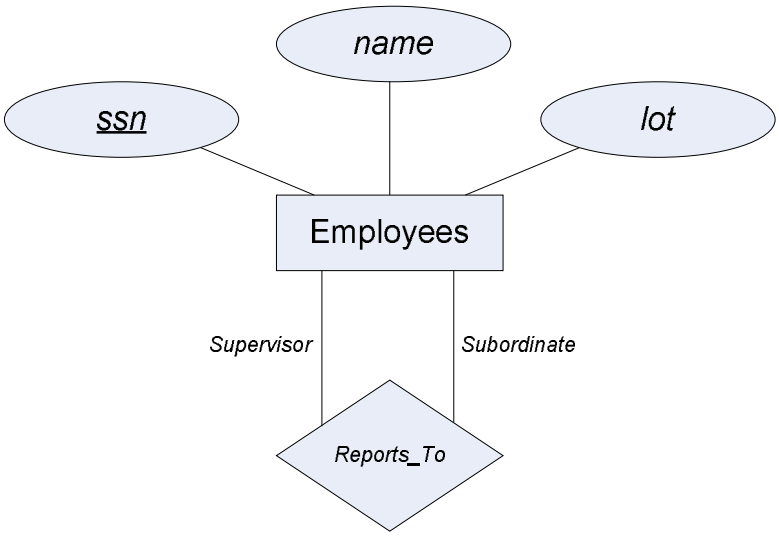


A relationship can also have descriptive attributes. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured by adding an attribute, since, to *Works_In*. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the *Works_In* relationship set, for example, each *Works_In* relationship must be uniquely identified by the combination of employee *ssn* and department did. Thus, for a given employee-department pair, we cannot have more than one associated since value.

An instance of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant in time. Each Employees entity is denoted by its *ssn*, and each Departments entity is denoted by its did, for simplicity. The since value is shown beside each relationship.



The entity sets that participate in a relationship set need not be distinct; sometimes a relationship might involve two entities in the same entity set. Since employees report to other employees, every relationship in *Reports_To* is of the form (*emp1; emp2*), where both *emp*1 and *emp*2 are entities in Employees. However, they play different **roles**: *emp*1 reports to the managing employee *emp*2, which is reflected in the **role indicators** *superisor* and *subordinate*. If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports To relationship set has attributes corresponding to the *ssn* of the supervisor and the *ssn* of the subordinate, and the names of these attributes are *supervisor ssn* and *subordinate ssn*.

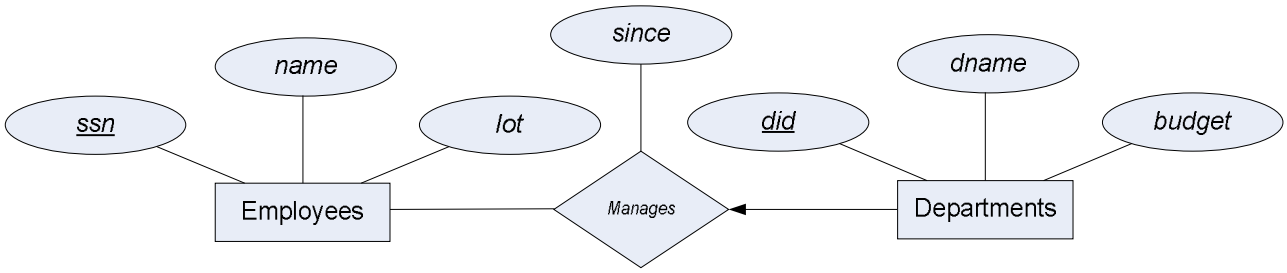


Key Constraints

An employee can work in several departments, and a department can have several employees, as illustrated in the Works In. Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees.

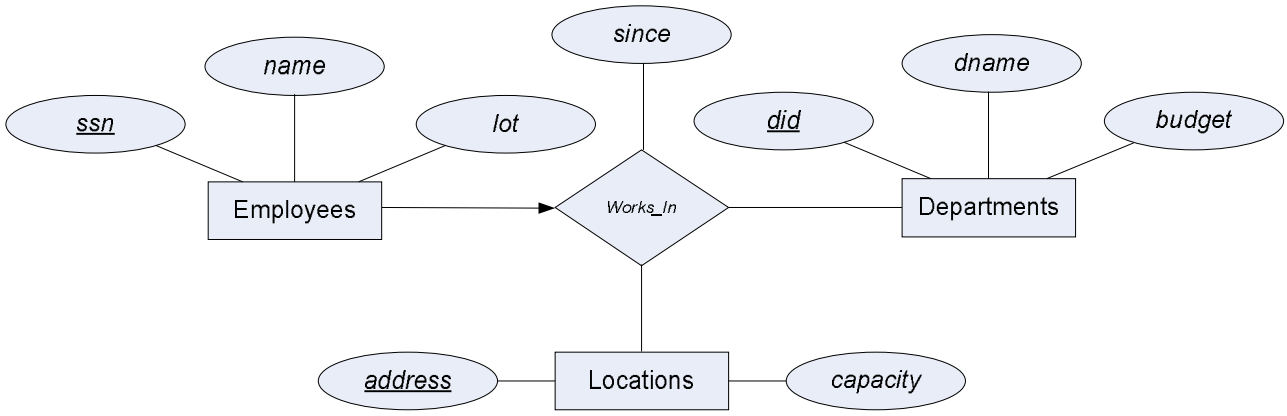
Now consider another relationship set called *Manages* between the *Employees* and *Departments* entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is an example of a **key constraint**, and it implies that each *Departments* entity appears in at most one *Manages* relationship in any

allowable instance of *Manages*. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.

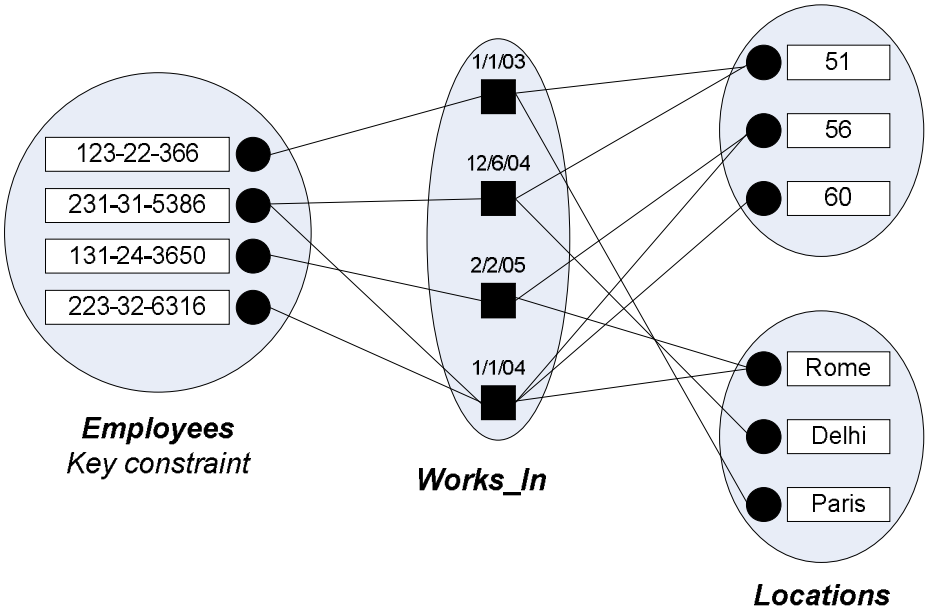


A relationship set like Manages is sometimes said to be **one-to-many**, to indicate that *one* employee can be associated with *many* departments (in the capacity of a manager), whereas each department can be associated with at most one employee as its manager. In contrast, the Works In relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be **many-to-many**.

In following figure, we show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location. An instance of the *Works_In* relationship set. Notice that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

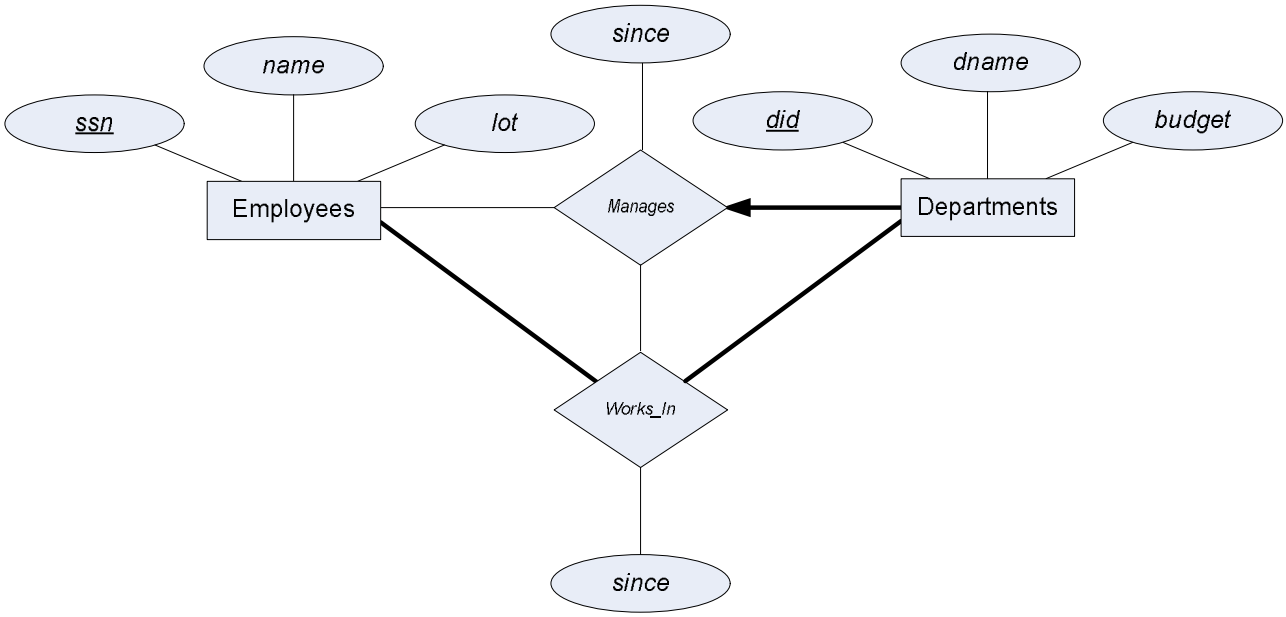


The instance of the following relations is:



The key constraint on *Manages* tells us that a department has at most one manager. A natural question to ask is whether every department has a manager. Let us say that every department is required to have a manager. This requirement is an example of a **participation constraint** ; the participation of the entity set *Departments* in the relationship set *Manages* is said to be **total**. A participation that is not total is said to be **partial**. As an example, the participation of the entity set *Employees* in *Manages* is partial, since not every employee gets to manage a department.

Revisiting the Works In relationship set, it is natural to expect that each employee works in at least one department and that each department has at least one employee. This means that the participation of both *Employees* and *Departments* in *Works_In* is total. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint.



Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

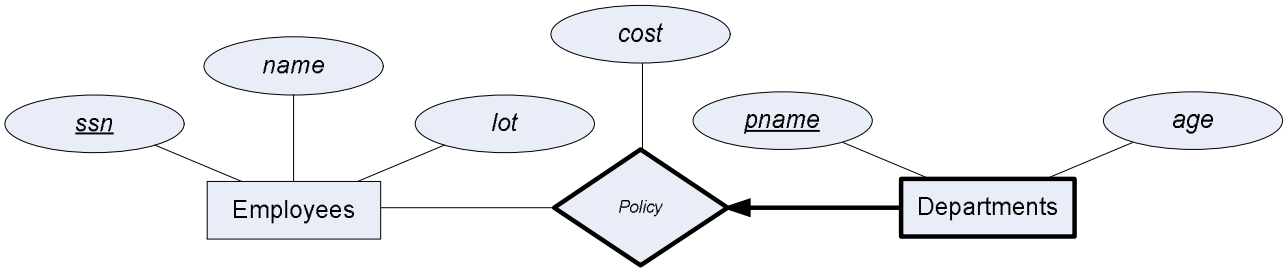
We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age*. The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is *ssn*; thus we might have two employees called Smethurst, and each might have a son called Joe.

Dependents is an example of a **weak entity set** . A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.

The total participation of Dependents in Policy is indicated by linking them with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same *pname* value.

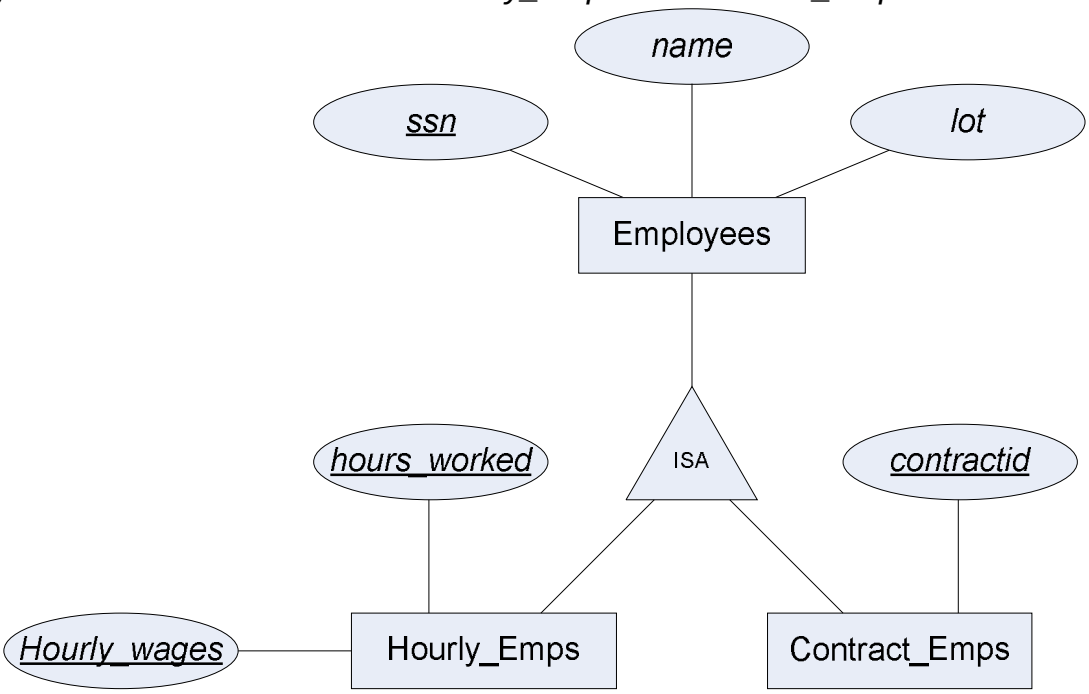


Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an *Hourly_Emps* entity set and a *Contract_Emps* entity set to distinguish the basis on which they are paid. We might have attributes *hours_worked* and *hourly_wage* defined for *Hourly_Emps* and an attribute *contracted* defined for *Contract_Emps*.

We want the semantics that every entity in one of these sets is also an Employees entity, and as such must have all of the attributes of Employees defined. Thus, the attributes defined for an *Hourly_Emps* entity are the attributes for Employees plus *Hourly_Emps*. We say that the attributes for the entity set Employees are **inherited** by the entity set *Hourly_Emps*, and that *Hourly_Emps* **ISA** (read *is a*) Employees. In addition| and in contrast to class hierarchies in programming languages such as C++ there is a

constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all *Hourly_Emps* and *Contract_Emps* entities as well.



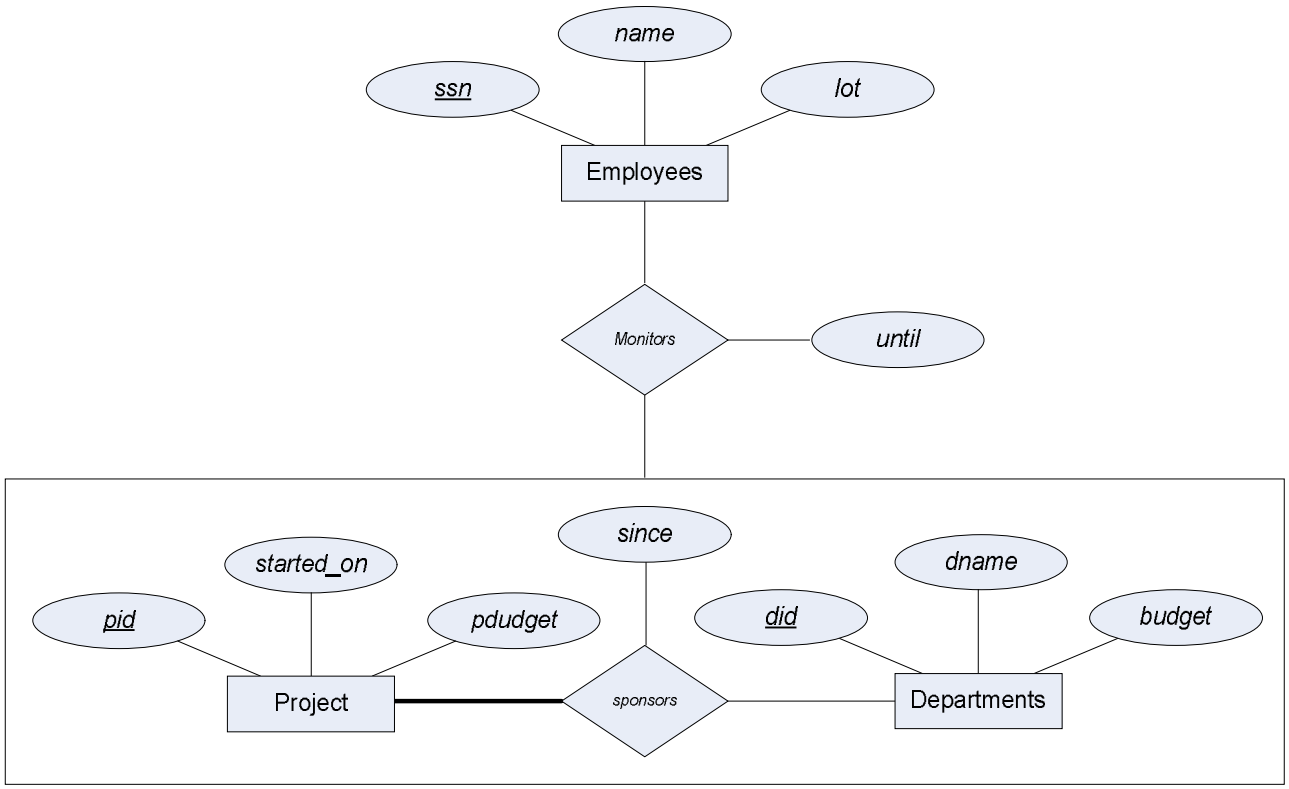
A class hierarchy can be viewed in one of two ways:

- *Employees* is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.
- *Hourly_Emps* and *Contract_Emps* are **generalized** by Employees. As another example, two entities sets Motorboats and Cars may be generalized into an entity set Motor Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

Aggregation

As we have defined it thus far, a relationship set is an association between entity sets. Sometimes we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

In order to define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set.



When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can't we express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the Monitors relationship has an attribute until that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute since of Sponsors, which is the date when the sponsorship took effect.

Conceptual Database Design with the ER Model

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

Conceptual Design for Large Enterprises

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups.

Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user

requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

An alternative approach is to develop separate conceptual schemas for different user groups and to then *integrate* these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, and differences in measurement units). This task is difficult in its own right. In some situations schema integration cannot be avoided for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to *heterogeneous* data sources, often maintained by different organizations.

The Relational Model

Codd proposed the relational data model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database field and largely supplanted these earlier models. Prototype relational database management systems were developed in pioneering research projects at IBM and UC-Berkeley by the mid-70s, and several vendors were offering relational database products shortly thereafter. Today, the relational model is by far the dominant data model and is the foundation for the leading DBMS products, including IBM's DB2 family, Informix, Oracle, Sybase, Microsoft's Access and SQLServer, FoxBase, and Paradox. Relational database systems are ubiquitous in the marketplace and represent a multibillion dollar industry.

The relational model is very simple and elegant; a database is a collection of one or more *relations*, where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a database, and it permits the use of simple, high-level languages to query the data. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

SQL: It was the query language of the pioneering System-R relational DBMS developed at IBM. Over the years, SQL has become the most widely used language for creating, manipulating, and querying relational DBMSs. Since many vendors offer SQL products, there is a need for a standard that defines 'official SQL.' The existence of a standard allows users to measure a given vendor's version of SQL for completeness. It also allows users to distinguish SQL features that are specific to one product from those that are standard; an application that relies on non-standard features is less portable. The first SQL standard was developed in 1986 by the American National Standards Institute (ANSI), and was called SQL-86. There was a minor revision in1989 called SQL-89, and a major revision in 1992 called SQL-92. The International Standards Organization (ISO) collaborated with ANSI to develop SQL-92. Most commercial DBMSs currently support SQL-92. An exciting development is the imminent approval of SQL:1999, a major extension of SQL-92.

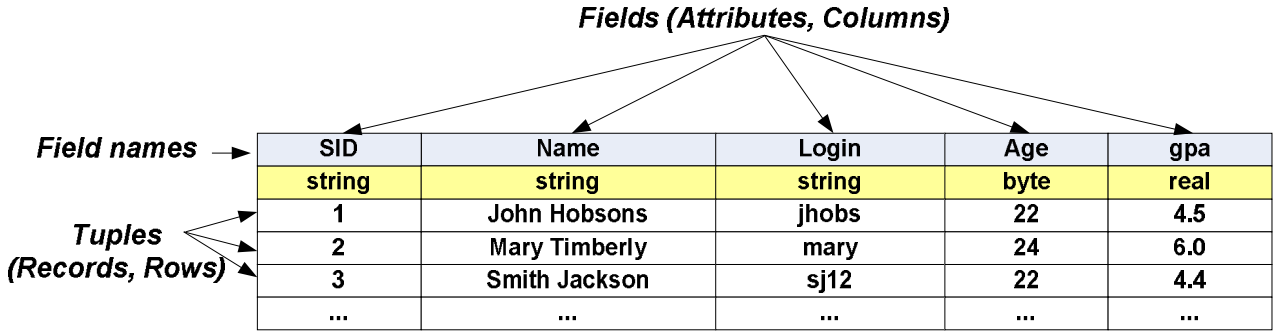
Introduction to Relational Model

The main construct for representing data in the relational model is a **relation**. A relation consists of a **relation schema** and a **relation instance**. The relation instance is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation's

name, the name of each **field** (or **column**, or **attribute**), and the **domain** of each field. A domain is referred to in a relation schema by the **domain name** and has a set of associated **values**.

```
Students(sid: string, name: string, login: string, age: integer, gpa:
real)
```

This says, for instance, that the field named *sid* has a domain named string. The set of values associated with domain string is the set of all character strings. We now turn to the instances of a relation. An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields. (The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)



SID	Name	Login	Age	gpa
string	string	string	byte	real
1	John Hobsons	jhobs	22	4.5
2	Mary Timberly	mary	24	6.0
3	Smith Jackson	sj12	22	4.4
...	...	...	...	...

The instance in previous example contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model each relation is defined to be a *set* of unique tuples or rows. The order in which the rows are listed is not important. If the fields are named, as in our schema definitions and figures depicting relation instances, the order of fields does not matter either. However, an alternative convention is to list fields in a specific order and to refer to a field by its position. Thus *sid* is field 1 of Students, *login* is field 3, and so on. If this convention is used, the order of fields is significant. Most database systems use a combination of these conventions. For example, in SQL the named fields convention is used in statements that retrieve tuples, and the ordered fields convention is commonly used when inserting tuples.

A relation schema specifies the domain of each field or column in the relation instance. These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the *type* of that field, in programming language terms, and restricts the values that can appear in the field.

More formally, let R(f_1:D_1, *: : :*, f_n:D_n) be a relation schema, and for each f_i, $1 \le i \le n$, let Dom_i be the set of values associated with the domain named D. An instance of R that satisfies the domain constraints in the schema is a set of tuples with *n* fields:

$$\left\{ \left\langle f_i : d_1, ..., f_n : d_n \right\rangle \middle| d_1 \in Dom_1, ... d_n \in Dom_n \right\}$$

The angular brackets identify the fields of a tuple.

Domain constraints are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, *relation instance* means *relation instance that satisfies the domain constraints in the relation schema.*

The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality** of a relation instance is the number of tuples in it.

A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, in previous lection, we discussed a university database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets In. An **instance** of a relational database is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

Creating and Modifying Relations Using SQL-92

The SQL-92 language standard uses the word *table* to denote *relation*, and we will often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the **Data Definition Language (DDL)**.

The CREATE TABLE statement is used to define a new table. To create the Students relation, we can use the following statement:

```
CREATE TABLE Students  (sid CHAR(20),
                        name CHAR(30),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL)
```

Tuples are inserted using the INSERT command. We can insert a single tuple into the Students table as follows:

```
INSERT
INTO Students (sid, name, login, age, gpa)
VALUES (53688, ‘Smith’, ‘smith@ee’, 18, 3.2)
```

We can optionally omit the list of column names in the INTO clause and list the values in the appropriate order, but it is good style to be explicit about column names.

We can delete tuples using the DELETE command. We can delete all Students tuples with *name* equal to Smith using the command:

```
DELETE
FROM Students S
WHERE S.name = ‘Smith’
```

We can modify the column values in an existing row using the UPDATE command. For example, we can increment the age and decrement the gpa of the student with *sid* 53688:

```
UPDATE Students S
SET S.age = S.age + 1, S.gpa = S.gpa - 1
WHERE S.sid = 53688
```

These examples illustrate some important points. The WHERE clause is applied first and determines which rows are to be modified. The SET clause then determines how these rows are to be modified. If the column that is being modified is also used to determine the new value, the value used in the expression on the right side of equals (=) is the *old* value, that is, before the modification. To illustrate these points further, consider the following variation of the previous query:

```
UPDATE Students S
SET S.gpa = S.gpa - 0.1
WHERE S.gpa >= 3.3
```

Integrity Constraints over Relations

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An **integrity constraint (IC)** is a condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database. Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.)

In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value. In this section we discuss the integrity constraints, other than domain constraints, that a DBA or user can specify in the relational model.

Key Constraints

Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*. In the case of the Students relation, the (set of fields containing just the) *sid* field is a candidate key.

Let us take a closer look at the above definition of a (candidate) key. There are two parts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

The first part of the definition means that in *any* legal instance, the values in the key fields uniquely identify a tuple in the instance. When specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a ‘correct’ set of tuples. (A similar comment applies to the specification of other kinds of ICs as well.) The notion of ‘correctness’ here depends upon the nature of the data being stored. For example, several students may have the same name, although each student has a unique student id. If the *name* field is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!

The second part of the definition means, for example, that the set of fields *{sid, name}* is not a key for Students, because this set properly contains the key *{sid}*. The set *{sid, name}* is an example of a **superkey**, which is a set of fields that contains a key.

A relation may have several candidate keys. For example, the *login* and *age* fields of the Students relation may, taken together, also identify students uniquely. That is, *{login, age}* is also a key. It may seem that *login* is a key, since no two rows in the example instance have the same *login* value. However, the key must identify tuples uniquely in all possible legal instances of the relation. By stating that { *login, age}* is a key, the user is declaring that two students may have the same login or age, but not both.

Out of all the available candidate keys, a database designer can identify a **primary** key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its *sid* value. As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their *sid* value. In principle, we can use any key, not just the primary key, to refer to a tuple. However, using the primary key is preferable because it is what the DBMS expects this is the significance of designating a particular candidate key as a primary key and optimizes for. For example, the DBMS may create an index with the primary key fields as the search key, to make the retrieval of a tuple given its primary key value efficient.

Specifying Key Constraints in SQL-92

In SQL we can declare that a subset of the columns of a table constitute a key by using the UNIQUE constraint. At most one of these 'candidate' keys can be declared to be a *primary key* , using the PRIMARY KEY constraint. (SQL does not require that such constraints be declared for a table.)

Let us revisit our example table definition and specify key information:

```
CREATE TABLE Students  (sid CHAR(20),
                        name CHAR(30),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid))
```

This definition says that *sid* is the primary key and that the combination of *name* and *age* is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with CONSTRAINT *constraint-name*. If the constraint is violated, the constraint name is returned and can be used to identify the error.

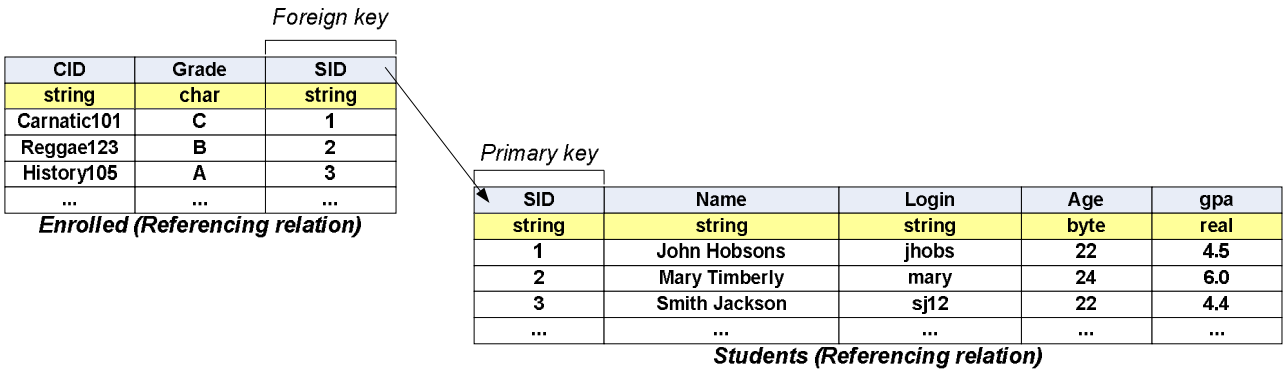
Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a *foreign key* constraint.

Suppose that in addition to Students, we have a second relation:

```
Enrolled(sid: string, cid: string, grade: string)
```

To ensure that only bona fide students can enroll in courses, any value that appears in the *sid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *sid* field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.



CID	Grade	SID
string	char	string
Carnatic101	C	1
Reggae123	B	2
History105	A	3
...	...	...

Enrolled (Referencing relation)

SID	Name	Login	Age	gpa
string	string	string	byte	real
1	John Hobsons	jhobs	22	4.5
2	Mary Timberly	mary	24	6.0
3	Smith Jackson	sj12	22	4.4
...	...	...	...	...

Students (Referencing relation)

However, every *sid* value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

If we try to insert the tuple *h55555, Art104, Ai* into *E*1, the IC is violated because there is no tuple in *S*1 with the id 55555; the database system should reject such an insertion. Similarly, if we delete the tuple *h53666, Jones, jones@cs, 18, 3.4i* from *S*1, we violate the foreign key constraint because the tuple *h53666, History105, Bi* in *E*1 contains *sid* value 53666, the *sid* of the deleted Students tuple. The DBMS should disallow the deletion or, perhaps, also delete the Enrolled tuple that refers to the deleted Students tuple.

Finally, we note that a foreign key could refer to the same relation. For example, we could extend the Students relation with a column called *partner* and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the *partner* field contains the partner's *sid*. The observant reader will no doubt ask, "What if a student does not (yet) have a partner?" This situation is handled in SQL by using a special value called **null**. The use of *null* in a field of a tuple means that value in that field is either unknown or not applicable (e.g., we do not know the partner yet, or there is no partner). The appearance of *null* in a foreign key field does not violate the foreign key constraint. However, *null* values are not allowed to appear in a primary key field (because the primary key fields are used to identify a tuple uniquely).

Specifying Foreign Key Constraints in SQL-92

Let us define:

```
Enrolled(sid: string, cid: string, grade: string)

CREATE TABLE Enrolled  sid CHAR(20),
                       cid CHAR(20),
                       grade CHAR(10),
                       PRIMARY KEY (sid, cid),
                       FOREIGN KEY (sid) REFERENCES Students )
```

The foreign key constraint states that every *sid* value in Enrolled must also appear in Students, that is, *sid* in Enrolled is a foreign key referencing Students. Incidentally, the primary key constraint states that a student has exactly one grade for each course that he or she is enrolled in. If we want to record more than one grade per student per course, we should change the primary key constraint.

General Constraints

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

For example, we may require that student ages be within a certain range of values; given such an IC specifying, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of Students shown in following figure is illegal because two students are underage.

SID	Name	Login	Age	gpa
string	string	string	byte	real
1	John Hobsons	jhobs	22	4.5
2	Mary Timberly	mary	24	6.0
3	Smith Jackson	sj12	22	4.4
...	...	...	...	...

The IC that students must be older than 16 can be thought of as an extended domain constraint, since we are essentially defining the set of permissible *age* values more stringently than is possible by simply using a standard domain such as integer. In general, however, constraints that go well beyond domain, key, or foreign key constraints can be specified example, we could require that every student whose age is greater than 18 must have a gpa greater than 3.

Current relational database systems support such general constraints in the form of *table constraints* and *assertions*. Table constraints are associated with a single table and are checked whenever that table is modified. In contrast, assertions involve several tables and are checked whenever any of these tables is modified. Both table constraints and assertions can use the full power of SQL queries to specify the desired restriction.

Enforcing Integrity Constraints

As we observed earlier, ICs are specified when a relation is created and enforced when a relation is modified. The impact of domain, PRIMARY KEY, and UNIQUE constraints is straightforward: if an insert, delete, or update command causes a violation, it is rejected. Potential IC violation is generally checked at the end of each SQL statement execution, although it can be *deferred* until the end of the transaction executing the statement.

SQL-92 provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. *What should we do if an Enrolled row is inserted, with a* sid *column value that does not appear in any row of the Students table?*

   In this case the INSERT command is simply rejected.

2. *What should we do if a Students row is deleted?*

   The options are:

   - Delete all Enrolled rows that refer to the deleted Students row.
   - Disallow the deletion of the Students row if an Enrolled row refers to it.
   - Set the *sid* column to the *sid* of some (existing) 'default' student, for every Enrolled row that refers to the deleted Students row.
   - For every Enrolled row that refers to it, set the *sid* column to *null*. In our example, this option conflicts with the fact that *sid* is part of the primary key of Enrolled and therefore cannot be set to *null*. Thus, we are limited to the first three options in our example, although this fourth option (setting the foreign key to *null*) is available in the general case.

3. *What should we do if the primary key value of a Students row is updated?*

   The options here are similar to the previous case.

SQL-92 allows us to choose any of the four options on DELETE and UPDATE. For example, we can specify that when a Students row is *deleted*, all Enrolled rows that refer to it are to be deleted as well, but that when the *sid* column of a Students row is *modified*, this update is to be rejected if an Enrolled row refers to the modified Students row:

```
CREATE TABLE Enrolled  ( sid CHAR(20),
                         cid CHAR(20),
                         grade CHAR(10),
                         PRIMARY KEY (sid, cid),
                         FOREIGN KEY (sid) REFERENCES Students
                         ON DELETE CASCADE
                         ON UPDATE NO ACTION )
```

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that if a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the *sid* column of a Students row is updated, this update is also carried out in each Enrolled row that refers to the updated Students row.

If a Students row is deleted, we can switch the enrollment to a 'default' student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the *sid* field in Enrolled; for example, *sid* CHAR(20) DEFAULT *'53666'*. Although the specification of a default value is appropriate in some situations (e.g., a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE), or to reject the update.

SQL also allows the use of *null* as the default value by specifying ON DELETE SET NULL.

Querying Relational Data

A **relational database query** (query, for short) is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students younger than 18 or all students enrolled in Reggae203. A **query language** is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried. Consider the instance of the Students relation. We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT *
FROM Students S
WHERE S.age < 18
```

The symbol * means that we retain all fields of selected tuples in the result. To understand this query, think of S as a variable that takes on the value of each tuple in Students, one tuple after the other. The condition *S.age < 18* in the WHERE clause specifies that we want to select only tuples in which the *age* field has a value less than 18.

This example illustrates that the domain of a field restricts the operations that are permitted on field values, in addition to restricting the values that can appear in the field. The condition *S.age < 18* involves an arithmetic comparison of an *age* value with an integer and is permissible because the domain of *age* is the set of integers. On the other hand, a condition such as *S.age = S.sid* does not make sense because it compares an

integer value with a string value, and this comparison is defined to fail in SQL; a query containing this condition will produce no answer tuples.

In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. We can compute the names and logins of students who are younger than 18 with the following query:

```
SELECT S.name, S.login
FROM Students S
WHERE S.age < 18
```

We can also combine information in the Students and Enrolled relations. If we want to obtain the names of all students who obtained an A and the id of the course in which they got an A, we could write the following query:

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = 'A'
```

DISTINCT **types in SQL:** A comparison of two values drawn from different domains should fail, even if the values are 'compatible' in the sense that both are numeric or both are string values etc. For example, if *salary* and *age* are two different domains whose values are represented as integers, a comparison of a salary value with an age value should fail. Unfortunately, SQL-92's support for the concept of domains does not go this far: We are forced to define *salary* and *age* as integer types and the comparison *S* < *A* will succeed when *S* is bound to the salary value 25 and *A* is bound to the age value 50. The latest version of the SQL standard, called SQL:1999, addresses this problem, and allows us to define *salary* and *age* as DISTINCT types even though their values are *represented* as integers. Many systems, e.g., Informix UDS and IBM DB2, already support this feature.

Introduction to Views

A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition** . Consider the Students and Enrolled relations. Suppose that we are often interested in finding the names and student identifiers of students who got a grade of B in some course, together with the cid for the course. We can define a view for this purpose. Using SQL-92 notation:

```
CREATE VIEW B-Students (name, sid, course)
AS SELECT S.sname, S.sid, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = 'B'
```

The view B-Students has three fields called *name*, *sid*, and *course* with the same domains as the fields *sname* and *sid* in *Students* and *cid* in Enrolled. (If the optional arguments *name*, *sid*, and *course* are omitted from the CREATE VIEW statement, the column names *sname*, *sid*, and *cid* are inherited.)

The *physical* schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The *conceptual* schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, i.e., be part of the *external* schema of the database, additional relations in the *external* schema can be defined using the view mechanism. The view mechanism thus provides the support for *logical data independence* in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we

can define a view with the old schema, and applications that expect to see the old schema can now use this view.

Views are also valuable in the context of *security*: We can define views that give a group of users access to just the information they are allowed to see. For example, we can define a view that allows students to see other students' name and age but not their *gpa*, and allow all students to access this view, but not the underlying Students table.

Updates on Views

The motivation behind the view mechanism is to tailor how users see the data. Users should not have to worry about the view versus base table distinction. This goal is indeed achieved in the case of queries on views; a view can be used just like any other relation in defining a query. However, it is natural to want to specify updates on views as well. Here, unfortunately, the distinction between a view and a base table must be kept in mind.

The SQL-92 standard allows updates to be specified only on views that are defined on a single base table using just selection and projection, with no use of aggregate operations. Such views are called **updatable views**. This definition is oversimplified, but it captures the spirit of the restrictions. An update on such a restricted view can always be implemented by updating the underlying base table in an unambiguous way.

Consider the following view:

```
CREATE VIEW GoodStudents (sid, gpa)
AS SELECT S.sid, S.gpa
FROM Students S
WHERE S.gpa > 3.0
```

We can implement a command to modify the gpa of a GoodStudents row by modifying the corresponding row in Students. We can delete a GoodStudents row by deleting the corresponding row from Students. (In general, if the view did not include a key for the underlying table, several rows in the table could 'correspond' to a single row in the view. This would be the case, for example, if we used *S.sname* instead of *S.sid* in the definition of GoodStudents. A command that affects a row in the view would then affect all corresponding rows in the underlying table.)

We can insert a GoodStudents row by inserting a row into Students, using *null* values in columns of Students that do not appear in GoodStudents (e.g., *sname*, *login*). Note that primary key columns are not allowed to contain *null* values. Therefore, if we attempt to insert rows through a view that does not contain the primary key of the underlying table, the insertions will be rejected. For example, if GoodStudents contained *sname* but not *sid*, we could not insert rows into Students through insertions to GoodStudents.

An important observation is that an INSERT or UPDATE may change the underlying base table so that the resulting (i.e., inserted or modified) row is not in the view! For example, if we try to insert a row *h51234, 2.8i* into the view, this row can be (padded with *null* values in the other fields of Students and then) added to the underlying Students table, but it will not appear in the GoodStudents view because it does not satisfy the view condition *gpa* > 3:0. The SQL-92 default action is to allow this insertion, but we can disallow it by adding the clause WITH CHECK OPTION to the definition of the view.

We caution the reader that when a view is defined in terms of another view, the interaction between these view definitions with respect to updates and the CHECK OPTION clause can be complex; we will not go into the details.

Destroying/Altering Tables and Views

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table definition information), we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

ALTER TABLE modifies the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:

```
ALTER TABLE Students
ADD COLUMN maiden-name CHAR(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

Lecture 3: SQL – Queries, Programming, Triggers

We will present a number of sample queries using the following table definitions:

```
Sailors(sid: integer, sname: string, rating: integer, age: real)
Boats(bid: integer, bname: string, color: string)
Reserves(sid: integer, bid: integer, day: date)
```

Sailors:

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Reserves:

sid	Bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Boats:

bid	bname	color
101	Interlake	Blue
102	Interlake	Red
103	Clipper	Green
104	Marine	Red

The Form of a Basic SQL Query

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy* . A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

```
SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification
```

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a

cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause. Let us consider a simple query.

Find the names and ages of all sailors.

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

Answer to query:

sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Answer to query without DISTINCT:

sname	age
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

The answer is a *set* of rows, each of which is a pair ⟨*sname, age*⟩. If two or more sailors have the same name and age, the answer still contains just one pair with that name and age. This query is equivalent to applying the projection operator of relational algebra.

If we omit the keyword DISTINCT, we would get a copy of the row ⟨*s,a*⟩ for each sailor with name *s* and age *a*; the answer would be a *multiset* of rows. A **multiset** is similar to a set in that it is an unordered collection of elements, but there could be several copies of each element, and the number of copies is significant two multisets could have the same elements and yet be different because the number of copies is different for some elements. For example, {a, b, b*}* and {b, a, b} denote the same multiset, and differ from the multiset *{*a, a, b}.

Find all sailors with a rating above 7.

```
SELECT S.sid, S.sname, S.rating, S.age
FROM Sailors AS S
WHERE S.rating > 7
```

This query uses the optional keyword AS to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides convenient

shorthand: We can simply write SELECT *. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained.

As these two examples illustrate, the SELECT clause is actually used to do *projection*, whereas *selections* in the relational algebra sense are expressed using the WHERE clause! This mismatch between the naming of the selection and projection operators in relational algebra and the syntax of SQL is an unfortunate historical accident.

We now consider the syntax of a basic SQL query in more detail.

- The **from-list** in the FROM clause is a list of table names. A table name can be followed by a **range variable**; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The **select-list** is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The **qualification** in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form *expression* op *expression*, where op is one of the comparison operators *{<;<=;=; <>;>=;>}.* An *expression* is a *column* name, a *constant*, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates*, that is, two copies of the same row. The default is that duplicates are not eliminated.

Although the preceding rules describe (informally) the syntax of a basic SQL query, they don't tell us the *meaning* of a query. The answer to a query is itself a relation which is a *multiset* of rows in SQL - whose contents can be understood by considering the following conceptual evaluation strategy:

1. Compute the cross-product of the tables in the **from-list**.
2. Delete those rows in the cross-product that fail the **qualification** conditions.
3. Delete all columns that do not appear in the **select-list**.
4. If DISTINCT is specified, eliminate duplicate rows.

Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid=103
```

Let us compute the answer to this query on the instances *R*3 of Reserves and *S*4:

Instance R3 of Reserves:

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Instance S4 of Sailors:

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

The first step is to construct the cross-product *S*4*xR*3, which is shown:

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96

31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

The second step is to apply the qualification *S.sid = R.sid* AND *R.bid=103*. (Note that the first part of this qualification requires a join operation.) This step eliminates all but the last row from the instance shown in previous figure. The third step is to eliminate unwanted columns; only *sname* appears in the SELECT clause. This step leaves us with the result shown in next figure, which is a table with a single column and, as it happens, just one row.

sname
Rusty

Examples

```
SELECT sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND bid=103
```

Only the occurrences of *sid* have to be qualified, since this column appears in both the Sailors and Reserves tables. An equivalent way to write this query is:

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid AND bid=103
```

Find the sids of sailors who have reserved a red boat.

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid = R.bid AND B.color = ‘red’
```

Find the names of sailors who have reserved a red boat.

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
```

Find the colors of boats reserved by Lubber.

```
SELECT B.color
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = ‘Lubber’
```

Find the names of sailors who have reserved at least one boat.

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```

Expressions and Strings in the SELECT Command

SQL supports a more general version of the **select-list** than just a list of columns. Each item in a **select-list** can be of the form *expression* AS *column name* , where *expression* is any arithmetic or string expression over column names (possibly prefixed by range variables) and constants. It can also contain *aggregates* such as *sum* and *count.* The SQL-92 standard also includes expressions over date and time values, which we will not discuss. Although not part of the SQL-92 standard, many implementations also support the use of built-in functions such as *sqrt*, *sin*, and *mod*.

Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
SELECT S.sname, S.rating+1 AS rating
FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid
AND R1.day = R2.day AND R1.bid <> R2.bid
```

Also, each item in a *qualification* can be as general as *expression1 = expression2*.

```
SELECT S1.sname AS name1, S2.sname AS name2
FROM Sailors S1, Sailors S2
WHERE 2*S1.rating = S2.rating-1
```

For string comparisons, we can use the comparison operators (= *;<;>;* etc.) with the ordering of strings determined alphabetically as usual. If we need to sort strings by an order other than alphabetical (e.g., sort strings denoting month names in the calendar order January, February, March, etc.), SQL-92 supports a general concept of a **collation**, or sort order, for a character set. A collation allows the user to specify which characters are 'less than' which others, and provides great flexibility in string manipulation.

In addition, SQL provides support for pattern matching through the LIKE operator, along with the use of the wild-card symbols % (which stands for zero or more arbitrary characters) and _ (which stands for exactly one, arbitrary, character). Thus, '_AB%' denotes a pattern that will match every string that contains at least three characters, with the second and third characters being A and B respectively. Note that unlike the other comparison operators, blanks can be significant for the LIKE operator (depending on the collation for the underlying character set). Thus, '*Jeff*' = '*Jeff* ' could be true while '*Jeff*' LIKE '*Jeff* ' is false. An example of the use of LIKE in a query is given below.

Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SELECT S.age
FROM Sailors S
WHERE S.sname LIKE 'B %B'
```

UNION, INTERSECT, and EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT. SQL also provides other set operations: IN (to check if an element is in a given set), op ANY, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning.

Consider the following query:

Find the names of sailors who have reserved a red or a green boat.

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
AND (B.color = 'red' OR B.color = 'green')
```

This query is easily expressed using the OR connective in the WHERE clause. However, the following query, which is identical except for the use of 'and' rather than 'or' in the English version, turns out to be much more difficult:

If we were to just replace the use of OR in the previous query by AND, in analogy to the English statements of the two queries, we would retrieve the names of sailors who have reserved a boat that is both red and green. The integrity constraint that *bid* is a key for Boats tells us that the same boat cannot have two colors, and so the variant of the previous query with AND in place of OR will always return an empty answer set.

Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname
FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2
WHERE S.sid = R1.sid AND R1.bid = B1.bid
AND S.sid = R2.sid AND R2.bid = B2.bid
AND B1.color='red' AND B2.color = 'green'
```

We can think of R1 and B1 as rows that prove that sailor *S.sid* has reserved a red boat. R2 and B2 similarly prove that the same sailor has reserved a green boat. *S.sname* is not included in the result unless five such rows S, R1, B1, R2, and B2 are found.

The previous query is difficult to understand (and also quite inefficient to execute, as it turns out). In particular, the similarity to the previous OR query is completely lost. A better solution for these two queries is to use UNION and INTERSECT.

The OR query (Query Q5) can be rewritten as follows:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats. In complete symmetry, the AND query can be rewritten as follows:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

This query actually contains a subtle bug if there are two sailors such as Horatio in our example instances *B*1, *R*2, and *S*3, one of whom has reserved a red boat and the other has reserved a green boat, the name Horatio is returned even though no one individual called Horatio has reserved both a red and a green boat. Thus, the query actually computes sailor names such that some sailor with this name has reserved a red boat and some sailor with the same name (perhaps a different sailor) has reserved a green boat.

If we select *sid* instead of *sname* in the previous query, we would compute the set of *sid*s of sailors who have reserved both red and green boats.

Our next query illustrates the set-difference operation in SQL.

Find the sids *of all sailors who have reserved red boats but not green boats.*

```
SELECT S.sid
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
```

```
SELECT S2.sid
FROM Sailors S2, Reserves R2, Boats B2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’
```

Sailors 22, 64, and 31 have reserved red boats. Sailors 22, 74, and 31 have reserved green boats. Thus, the answer contains just the *sid* 64.

Indeed, since the Reserves relation contains *sid* information, there is no need to look at the Sailors relation, and we can use the following simpler query:

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = ‘red’
EXCEPT
SELECT R2.sid
FROM Boats B2, Reserves R2
WHERE R2.bid = B2.bid AND B2.color = ‘green’
```

Note that UNION, INTERSECT, and EXCEPT can be used on *any* two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types. For example, we can write the following query:

Find all sids of sailors who have a rating of 10 or have reserved boat 104.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating = 10
UNION
SELECT R.sid
FROM Reserves R
WHERE R.bid = 104
```

The first part of the union returns the *sid*s 58 and 71. The second part returns 22 and 31. The answer is, therefore, the set of *sid*s 22, 31, 58, and 71. A final point to note about UNION, INTERSECT, and EXCEPT follows. In contrast to the default that duplicates are not eliminated unless DISTINCT is specified in the basic query form, the default for UNION queries is that duplicates *are* eliminated! To retain duplicates, UNION ALL must be used; if so, the number of copies of a row in the result is $m + n$, where m and n are the numbers of times that the row appears in the two parts of the union. Similarly, one version of INTERSECT retains duplicates the number of copies of a row in the result is $min(m; n)$|and one version of EXCEPT also retains duplicates the number of copies of a row in the result is $m - n$, where m corresponds to the first relation.

Nested Queries

One of the most powerful features of SQL is nested queries. A **nested query** is a query that has another query embedded within it; the embedded query is called a **subquery**. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause. This section discusses only subqueries that appear in the WHERE clause. The treatment of subqueries appearing elsewhere is quite similar.

Find the names of sailors who have reserved boat 103.

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
```

```
FROM Reserves R
WHERE R.bid = 103 )
```

The nested subquery computes the (multi)set of *sid*s for sailors who have reserved boat 103 (the set contains 22, 31, and 74 on instances *R*2 and *S*3), and the top-level query retrieves the names of sailors whose *sid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. Notice that it is very easy to modify this query to find all sailors who have *not* reserved boat 103 we can just replace IN by NOT IN!

The best way to understand a nested query is to think of it in terms of a conceptual evaluation strategy. In our example, the strategy consists of examining rows in Sailors, and for each such row, evaluating the subquery over Reserves. In general, the conceptual evaluation strategy that we presented for defining the semantics of a query can be extended to cover nested queries as follows: Construct the cross-product of the tables in the FROM clause of the top-level query as before. For each row in the cross-product, while testing the qualification in the WHERE clause, (re)compute the subquery.5 Of course, the subquery might itself contain another nested subquery, in which case we apply the same idea one more time, leading to an evaluation strategy with several levels of nested loops.

As an example of a multiply-nested query, let us rewrite the following query.

Find the names of sailors who have reserved a red boat.

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
FROM Reserves R
WHERE R.bid IN ( SELECT B.bid
FROM Boats B
WHERE B.color = ‘red’ )
```

The innermost subquery finds the set of *bid*s of red boats (102 and 104 on instance *B*1). The subquery one level above finds the set of *sid*s of sailors who have reserved one of these boats. On instances *B*1, *R*2, and *S*3, this set of *sid*s ontains 22, 31, and 64. The top-level query finds the names of sailors whose *sid* is in this set of *sid*s. For the example instances, we get Dustin, Lubber, and Horatio.

To find the names of sailors who have not reserved a red boat, we replace the outermost occurrence of IN by NOT IN:

Find the names of sailors who have not *reserved a red boat.*

```
SELECT S.sname
FROM Sailors S
WHERE S.sid NOT IN ( SELECT R.sid
FROM Reserves R
WHERE R.bid IN ( SELECT B.bid
FROM Boats B
WHERE B.color = ‘red’ )
```

This query computes the names of sailors whose *sid* is *not* in the set 22, 31, and 64.

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query. In general the inner subquery could depend on the row that is currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more:

Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS ( SELECT *
FROM Reserves R
WHERE R.bid = 103
AND R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty. Thus, for each Sailor row *S*, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty. If so, sailor *S* has reserved boat 103, and we retrieve the name. The subquery clearly depends on the current row *S* and must be re-evaluated for each row in Sailors. The occurrence of *S* in the subquery (in the form of the literal *S.sid*) is called a *correlation*, and such queries are called *correlated queries*.

This query also illustrates the use of the special symbol * in situations where all we want to do is to check that a qualifying row exists, and don't really want to retrieve any columns from the row. This is one of the two uses of * in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation, which we will describe shortly.

As a further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate. When we apply UNIQUE to a subquery, it returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates; in particular, it returns true if the answer is empty. (And there is also a NOT UNIQUE version.)

Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators *{<;<=;=; <>;>=;>}.* . (SOME is also available, but it is just a synonym for ANY.)

Find sailors whose rating is better than some sailor called Horatio.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating
FROM Sailors S2
WHERE S2.sname = 'Horatio' )
```

If there are several sailors called Horatio, this query finds all sailors whose rating is better than that of *some* sailor called Horatio. On instance *S*3, this computes the *sid*s 31, 32, 58, 71, and 74. What if there were *no* sailor called Horatio? In this case the comparison *S.rating* > ANY is defined to return false, and the above query returns an empty answer set. To understand comparisons involving ANY, it is useful to think of the comparison being carried out repeatedly. In the example above, *S.rating* is successively compared with each rating value that is an answer to the nested query. Intuitively, the subquery must return a row that makes the comparison true, in order for *S.rating* > ANY to return true.

Find sailors whose rating is better than every sailor called Horatio.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating
```

```
FROM Sailors S2 )
```

The subquery computes the set of all rating values in Sailors. The outer WHERE condition is satisfied only when *S.rating* is greater than or equal to each of these rating values, i.e., when it is the largest rating value. In the instance *S*3, the condition is only satisfied for *rating* 10, and the answer includes the *sid*s of sailors with this rating, i.e., 58 and 71.

Note that IN and NOT IN are equivalent to = ANY and <> ALL, respectively.

More Examples of Nested Queries

Let us revisit a query that we considered earlier using the INTERSECT operator.

Find the names of sailors who have reserved both a red and a green boat.

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’
AND S.sid IN ( SELECT S2.sid
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid
AND B2.color = ‘green’ )
```

This query can be understood as follows: \Find all sailors who have reserved a red boat and, further, have *sid*s that are included in the set of *sid*s of sailors who have reserved a green boat." This formulation of the query illustrates how queries involving INTERSECT can be rewritten using IN, which is useful to know if your system does not support INTERSECT. Queries using EXCEPT can be similarly rewritten by using NOT IN. To find the *sid*s of sailors who have reserved red boats but not green boats, we can simply replace the keyword IN in the previous query by NOT IN.

As it turns out, writing this query using INTERSECT is more complicated because we have to use *sid*s to identify sailors (while intersecting) and have to return sailor names:

```
SELECT S3.sname
FROM Sailors S3
WHERE S3.sid IN (( SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = ‘red’ )
INTERSECT
(SELECT R2.sid
FROM Boats B2, Reserves R2
WHERE R2.bid = B2.bid AND B2.color = ‘green’ ))
```

Our next example illustrates how the *division* operation in relational algebra can be expressed in SQL.

Find the names of sailors who have reserved all boats.

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid
FROM Boats B )
EXCEPT
(SELECT R.bid
FROM Reserves R
WHERE R.sid = S.sid ))
```

Notice that this query is correlated for each sailor *S*, we check to see that the set of boats reserved by *S* includes all boats. An alternative way to do this query without using EXCEPT follows:

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS ( SELECT B.bid
FROM Boats B
WHERE NOT EXISTS ( SELECT R.bid
FROM Reserves R
WHERE R.bid = B.bid
AND R.sid = S.sid ))
```

Aggregate Operators

In addition to simply retrieving data, we often want to perform some computation or summarization. As we noted earlier, SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL-92 does not preclude this).

Find the average age of all sailors.

```
SELECT AVG (S.age)
FROM Sailors S
```

Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating = 10
```

Find the name and age of the oldest sailor. Illegal!!!

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

The intent is for this query to return not only the maximum age but also the name of the sailors having that age. However, this query is illegal in SQL if the SELECT clause uses an aggregate operation, then it must use *only* aggregate operations unless the query contains a GROUP BY clause!

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MAX (S2.age)
FROM Sailors S2 )
```

We can count the number of sailors using COUNT. This example illustrates the use of * as an argument to COUNT, which is useful when we want to count all rows.

Count the number of sailors.

```
SELECT COUNT (*)
FROM Sailors S
```

Count the number of different sailor names.

```
SELECT COUNT ( DISTINCT S.sname )
FROM Sailors S
```

Aggregate operations offer an alternative to the ANY and ALL constructs. For example, consider the following query:

Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT S.sname
FROM Sailors S
WHERE S.age > ( SELECT MAX ( S2.age )
FROM Sailors S2
WHERE S2.rating = 10 )
```

On instance *S*3, the oldest sailor with rating 10 is sailor 58, whose age is 35. The names of older sailors are Bob, Dustin, Horatio, and Lubber. Using ALL, this query could alternatively be written as follows:

```
SELECT S.sname
FROM Sailors S
WHERE S.age > ALL ( SELECT S2.age
FROM Sailors S2
WHERE S2.rating = 10 )
```

However, the ALL query is more error prone one could easily (and incorrectly!) use ANY instead of ALL, and retrieve sailors who are older than *some* sailor with a rating of 10. The use of ANY intuitively corresponds to the use of MIN, instead of MAX, in the previous query.

The GROUP BY and HAVING Clauses

Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of **groups** of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). For example, consider the following query:

Find the age of the youngest sailor for each rating level.

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

Consider *i* = 1;2; … ; 10. Writing 10 such queries is tedious. More importantly, we may not know what rating levels exist in advance.

To write such queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause. In fact, the extension also includes an optional HAVING clause that can be used to specify qualifications over groups (for example, we may only be interested in rating levels > 6). The general form of an SQL query with these extensions is:

```
SELECT [ DISTINCT ] select-list
FROM from-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

Using the GROUP BY clause, we can write the previous sailor's query as follows:

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
GROUP BY S.rating
```

Let us consider some important points concerning the new clauses:

1. The **select-list** in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form **aggop** (*column-name*) AS *new-name*. The optional AS *new-name* term gives this column a name in the table that is the result of the query. Any of the aggregation operators can be used for **aggop**. Every column that appears in (1) must also appear in **grouping-list**. The reason is that each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on the values of columns in grouping-list. If a column appears in list (1), but not in **grouping-list**, it is not clear what value should be assigned to it in an answer row.
2. The expressions appearing in the **group-qualification** in the HAVING clause must have a *single* value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. Therefore, a column appearing in the **group-qualification** must appear as the argument to an aggregation operator, or it must also appear in **grouping-list**.
3. If the GROUP BY clause is omitted, the entire table is regarded as a single group.

More Examples of Aggregate Queries

For each red boat, find the number of reservations for this boat.

```
SELECT B.bid, COUNT (*) AS sailorcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
```

It is interesting to observe that the following version of the above query is illegal:

```
SELECT B.bid, COUNT (*) AS sailorcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid
GROUP BY B.bid
HAVING B.color = 'red'
```

Even though the group-qualification *B.color* = '*red*' is single-valued per group, since the grouping attribute *bid* is a key for Boats (and therefore determines *color*), SQL disallows this query. Only columns that appear in the GROUP BY clause can appear in the HAVING clause, unless they appear as arguments to an aggregate operator in the HAVING clause.

Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT S.rating, AVG (S.age) AS avgage
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECT S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
FROM Sailors S2
WHERE S.rating = S2.rating )
```

Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such *sailors.*

```
SELECT S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age > 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
FROM Sailors S2
WHERE S.rating = S2.rating AND S2.age >= 18 )
```

Find those ratings for which the average age of sailors is the minimum over all ratings.

```
SELECT S.rating
FROM Sailors S
WHERE AVG (S.age) = ( SELECT MIN (AVG (S2.age))
FROM Sailors S2
GROUP BY S2.rating )
```

A little thought shows that last query will not work even if the expression MIN (AVG (S2.age)), which is illegal, were allowed. In the nested query, Sailors is partitioned into groups by rating, and the average age is computed for each rating value. For each group, applying MIN to this average age value for the group will return the same value!

A correct version of the above query follows. It essentially computes a temporary table containing the average age for each rating value and then finds the rating(s) for which this average age is the minimum.

```
SELECT Temp.rating, Temp.avgage
FROM ( SELECT S.rating, AVG (S.age) AS avgage,
FROM Sailors S
GROUP BY S.rating) AS Temp
WHERE Temp.avgage = ( SELECT MIN (Temp.avgage) FROM Temp )
```

Null Values

Thus far, we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a *rating* column, what row should we insert for Dan? What is needed here is a special value that denotes *unknown*. Suppose the Sailor table definition was modified to also include a *maiden-name* column. However, only married women who take their husband's last name have a maiden name. For single women and for men, the *maiden-name* column is *inapplicable*. Again, what value do we include in this column for the row representing Dan?

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. Using our Sailor table definition, we might enter the row *á*98*;Dan; null;* 39*ñ* to represent Dan. The presence of *null* values complicates many issues, and we consider the impact of *null* values on SQL in this section.

Comparisons Using Null Values

Consider a comparison such as *rating = 8* . If this is applied to the row for Dan, is this condition true or false? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons *rating* > 8 and *rating* < 8 as well. Perhaps less obviously, if we compare two *null* values using *<; >;* =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

Logical Connectives AND, OR, and NOT

Now, what about boolean expressions such as *rating* = 8 OR *age* < 40 and *rating* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating*, but what about the second? We can only say unknown.

But this example raises an important point once we have *null* values, we must define the logical operators AND, OR, and NOT using a *three-valued* logic in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, it evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, it evaluates to true.)

Impact on SQL Constructs

Boolean expressions arise in many contexts in SQL, and the impact of *null* values must be recognized. For example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of *null* values, any row that evaluates to false or to unknown is eliminated. Eliminating rows that evaluate to unknown has a subtle but significant impact on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of *null* values is the definition of when two rows in a relation instance are regarded as *duplicates*. The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null*. Contrast this definition with the fact that if we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

As expected, the arithmetic operations + *;−;* _ , and = all return *null* if one of their arguments is *null*. However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles *null* values just like other values, that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard *null* values thus SUM cannot be understood as just the addition of all values in the (multi)set of values that it is applied to; a preliminary step of discarding all *null* values must also be accounted for. As a special case, if one of these operators other than COUNT is applied to *only* null values, the result is again *null*.

Outer Joins

Some interesting variants of the join operation that rely on *null* values, called **outer joins**, are supported in SQL. Consider the join of two tables, say Sailors $><_c$ Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition *c* do not appear in the result. In an outer join, on the other hand, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned *null* values.

In fact, there are several variants of the outer join idea. In a **left outer join**, Sailor rows without a matching Reserves row appear in the result, but not vice versa. In a **right outer join**, Reserves rows without a matching Sailors row appear in the result, but not vice versa. In a **full outer join**, both Sailors and Reserves rows without a match appear in the result.

SQL-92 allows the desired type of join to be specified in the FROM clause. For example, the following query lists *ásid,bidñ* pairs corresponding to sailors and boats they have reserved:

```
SELECT Sailors.sid, Reserves.bid
FROM Sailors NATURAL LEFT OUTER JOIN Reserves R
```

Disallowing Null Values

We can disallow *null* values by specifying NOT NULL as part of the field definition, for example, *sname* CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on *null* values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

Our coverage of *null* values is far from complete. The interested reader should consult one of the many books devoted to SQL for a more detailed treatment of the topic.

Complex Integrity Constraints in Sql-92

In this section we discuss the specification of complex integrity constraints in SQL-92, utilizing the full power of SQL query constructs.

Constraints over a Single Table

We can specify complex constraints over a single table using **table constraints**, which have the form CHECK *conditional-expression*. For example, to ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK ( rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves ( sid INTEGER,
bid INTEGER,
day DATE,
FOREIGN KEY (sid) REFERENCES Sailors
FOREIGN KEY (bid) REFERENCES Boats
CONSTRAINT noInterlakeRes
CHECK ( 'Interlake' <>
( SELECT B.bname
FROM Boats B
```

```
WHERE B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the *conditional expression* in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

Domain Constraints

A user can define a new domain using the CREATE DOMAIN statement, which makes use of CHECK constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 0
CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the **base type** for the domain *ratingval,* and every *ratingval* value must be of this type. Values in *ratingval* are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema declaration, for example:

```
rating ratingval
```

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation, and no value is entered for this column in an inserted tuple, the default value 0 associated with ratingval is used. (If a default value is specified for the column as part of the table definition, this takes precedence over the default value associated with the domain.) This feature can be used to minimize data entry errors; common default values are automatically filled in rather than being typed in.

SQL-92's support for the concept of a domain is limited in an important respect. For example, we can define two domains called Sailorid and Boatclass, each using INTEGER as a base type. The intent is to force a comparison of a Sailorid value with a Boatclass value to always fail (since they are drawn from different domains); however, since they both have the same base type, INTEGER, the comparison will succeed in SQL-92. This problem is addressed through the introduction of *distinct types* in SQL:1999.

Triggers and Active Databases

A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an **active database** . A trigger description contains three parts:

- **Event**: A change to the database that **activates** the trigger.
- **Condition**: A query or test that is run when the trigger is activated.
- **Action**: A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a "daemon" that monitors a database, and is executed when the database is modified in a way that matches the *event* specification. An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.

A *condition* in a trigger can be a true/false statement (e.g., all employee salaries are less than $100,000) or a query. A query is interpreted as *true* if the answer set is non empty, and *false* if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit), or call host language procedures.

An important issue is when the action part of a trigger executes in relation to the statement that activated the trigger. For example, a statement that inserts records into the Students table may activate a trigger that is used to maintain statistics on how many students younger than 18 are inserted at a time by a typical insert statement. Depending on exactly what the trigger does, we may want its action to execute *before* changes are made to the Students table, or *after* a trigger that initializes a variable used to count the number of qualifying insertions should be executed before, and a trigger that executes once per qualifying inserted record and increments the variable should be executed after each record is inserted (because we may want to examine the values in the new record to determine the action).

Examples of Triggers in SQL

The examples shown below is written using Oracle 7 Server syntax for defining triggers, illustrate the basic concepts behind triggers. (The SQL:1999 syntax for these triggers is similar; we will see an example using SQL:1999 syntax shortly.) The trigger called *init count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr_count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

```
CREATE TRIGGER init count BEFORE INSERT ON Students /* Event */
DECLARE
count INTEGER;
BEGIN /* Action */
count := 0;
END
CREATE TRIGGER incr count AFTER INSERT ON Students  /* Event */
WHEN (new.age < 18) /* Condition; 'new' is just-inserted tuple */
FOR EACH ROW
BEGIN /* Action; a procedure in Oracle's PL/SQL syntax */
count := count + 1;
END
```

One of the example triggers executes before the activating statement, and the other example executes after. A trigger can also be scheduled to execute *instead of* the activating statement, or in *deferred* fashion, at the end of the transaction containing the activating statement, or in *asynchronous* fashion, as part of a separate transaction.

The example illustrates another point about trigger execution: A user must be able to specify whether a trigger is to be executed once per modified record or once per activating statement. If the action depends on individual changed records, for example, we have to examine the *age* field of the inserted Students record to decide whether to increment the count, the triggering event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a **row-level trigger**. On the other hand, the *init count* trigger is executed just once per INSERT

statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a **statement-level trigger**.

In this example, the keyword new refers to the newly inserted tuple. If an existing tuple were modified, the keywords old and new could be used to refer to the values before and after the modification. The SQL:1999 draft also allows the action part of a trigger to refer to the *set* of changed records, rather than just one changed record at a time. For example, it would be useful to be able to refer to the set of inserted Students records in a trigger that executes once after the INSERT statement; we could count the number of inserted records with *age* < 18 through an SQL query over this set.

The definition in next example (set-oriented trigger) uses the syntax of the SQL:1999 draft, in order to illustrate the similarities and differences with respect to the syntax used in a typical current DBMS. The keyword clause NEW TABLE enables us to give a table name (InsertedTuples) to the set of newly inserted tuples. The FOR EACH STATEMENT clause specifies a statement-level trigger and can be omitted because it is the default. This definition does not have a WHEN clause; if such a clause is included, it follows the FOR EACH STATEMENT clause, just before the action specification.

The trigger is evaluated once for each SQL statement that inserts tuples into Students, and inserts a single tuple into a table that contains statistics on modifications to database tables. The first two fields of the tuple contain constants (identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18.

```
CREATE TRIGGER set count AFTER INSERT ON Students /* Event */
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
INSERT /* Action */
INTO StatisticsTable(Modi_edTable, Modi_cationType, Count)
SELECT ‘Students’, ‘Insert’, COUNT *
FROM InsertedTuples I
WHERE I.age < 18
```

Designing Active Databases

Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution. The effect of a collection of triggers can be very complex, and maintaining an active database can become very difficult. Often, a judicious use of integrity constraints can replace the use of triggers.

In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part. If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order. An important point is that the execution of the action part of a trigger could in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called **recursive triggers** . The potential for such *chain* activations, and the unpredictable order in which a DBMS processes activated triggers, can make it difficult to understand the effect of a collection of triggers.

Constraints versus Triggers

A common use of triggers is to maintain database consistency, and in such cases, we should always consider whether using an integrity constraint (e.g., a foreign key constraint) will achieve the same goals. The meaning of a constraint is not defined operationally, unlike the effect of a trigger. This property makes a constraint easier to

understand, and also gives the DBMS more opportunities to optimize execution. A constraint also prevents the data from being made inconsistent by *any* kind of statement, whereas a trigger is activated by a specific kind of statement (e.g., an insert or delete statement). Again, this restriction makes a constraint easier to understand.

On the other hand, triggers allow us to maintain database integrity in more flexible ways, as the following examples illustrate.

- Suppose that we have a table called Orders with fields *itemid*, *quantity*, *customerid*, and *unitprice*. When a customer places an order, the first three field values are filled in by the user (in this example, a sales clerk). The fourth field's value can be obtained from a table called Items, but it is important to include it in the Orders table to have a complete record of the order, in case the price of the item is subsequently changed. We can define a trigger to look up this value and include it in the fourth field of a newly inserted record. In addition to reducing the number of fields that the clerk has to type in, this trigger eliminates the possibility of an entry error leading to an inconsistent price in the Orders table.
- Continuing with the above example, we may want to perform some additional actions when an order is received. For example, if the purchase is being charged to a credit line issued by the company, we may want to check whether the total cost of the purchase is within the current credit limit. We can use a trigger to do the check; indeed, we can even use a CHECK constraint. Using a trigger, however, allows us to implement more sophisticated policies for dealing with purchases that exceed a credit limit. For instance, we may allow purchases that exceed the limit by no more than 10% if the customer has dealt with the company for at least a year, and add the customer to a table of candidates for credit limit increases.

Other Uses of Triggers

Many potential uses of triggers go beyond integrity maintenance. Triggers can alert users to unusual events (as reflected in updates to the database). For example, we may want to check whether a customer placing an order has made enough purchases in the past month to qualify for an additional discount; if so, the sales clerk must be informed so that he can tell the customer, and possibly generate additional sales! We can relay this information by using a trigger that checks recent purchases and prints a message if the customer qualifies for the discount.

Triggers can generate a log of events to support auditing and security checks. For example, each time a customer places an order, we can create a record with the customer's id and current credit limit, and insert this record in a customer history table. Subsequent analysis of this table might suggest candidates for an increased credit limit (e.g., customers who have never failed to pay a bill on time and who have come within 10% of their credit limit at least three times in the last month).

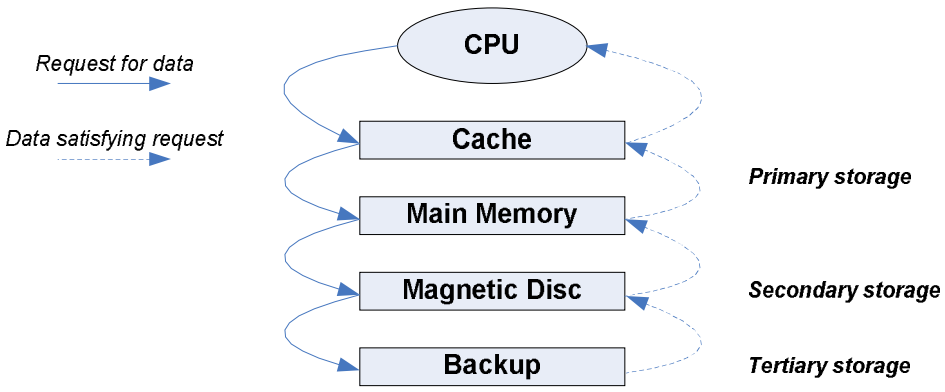
Lecture 4: Storing Data and Indexing

Data in a DBMS is stored on storage devices such as disks and tapes; we concentrate on disks and cover tapes briefly. The disk space manager is responsible for keeping track of available disk space. The file manager, which provides the abstraction of a file of records to higher levels of DBMS code, issues requests to the disk space manager to obtain and relinquish space on disk. The file management layer requests and frees disk space in units of a **page**; the size of a page is a DBMS parameter, and typical values are 4 KB or 8 KB. The file management layer is responsible for keeping track of the pages in a file and for arranging records within pages.

When a record is needed for processing, it must be fetched from disk to main memory. The page on which the record resides is determined by the file manager. Sometimes, the file manager uses auxiliary data structures to quickly identify the page that contains a desired record. After identifying the required page, the file manager issues a request for the page to a layer of DBMS code called the buffer manager. The buffer manager fetches a requested page from disk into a region of main memory called the buffer pool and tells the file manager the location of the requested page.

The Memory Hierarchy

Memory in a computer system is arranged in a hierarchy, as shown in figure below. At the top, we have **primary storage**, which consists of cache and main memory, and provides very fast access to data. Then comes **secondary storage**, which consists of slower devices such as magnetic disks. **Tertiary storage** is the slowest class of storage devices; for example, optical disks and tapes. Currently, the cost of a given amount of main memory is about 100 times the cost of the same amount of disk space, and tapes are even less expensive than disks. Slower storage devices such as tapes and disks play an important role in database systems because the amount of data is typically very large. Since buying enough main memory to store all data is prohibitively expensive, we must store data on tapes and disks and build database systems that can retrieve data from lower levels of the memory hierarchy into main memory as needed for processing.



There are reasons other than cost for storing data on secondary and tertiary storage. On systems with 32-bit addressing, only 232 bytes can be directly referenced in main memory; the number of data objects may exceed this number! Further, data must be maintained across program executions. This requires storage devices that retain information when the computer is restarted (after a shutdown or a crash); we call such storage **nonvolatile**. Primary storage is usually volatile (although it is possible to make it nonvolatile by adding a battery backup feature), whereas secondary and tertiary storage is nonvolatile.

Tapes are relatively inexpensive and can store very large amounts of data. They are a good choice for *archival* storage, that is, when we need to maintain data for a long period but do not expect to access it very often.

Performance Implications of Disk Structure

1. Data must be in memory for the DBMS to operate on it.
2. The unit for data transfer between disk and main memory is a block; if a single item on a block is needed, the entire block is transferred. Reading or writing a disk block is called an **I/O** (for input/output) operation.
3. The time to read or write a block varies, depending on the location of the data:

$$access\ time = seek\ time + rotational\ delay + transfer\ time$$

These observations imply that the time taken for database operations is affected significantly by how data is stored on disks. The time for moving blocks to or from disk usually dominates the time taken for database operations. To minimize this time, it is necessary to locate data records strategically on disk, because of the geometry and mechanics of disks. In essence, if two records are frequently used together, we should place them close together. The 'closest' that two records can be on a disk is to be on the same block. In decreasing order of closeness, they could be on the same track, the same cylinder, or an adjacent cylinder.

Two records on the same block are obviously as close together as possible, because they are read or written as part of the same block. As the platter spins, other blocks on the track being read or written rotate under the active head. In current disk designs, all the data on a track can be read or written in one revolution. After a track is read or written, another disk head becomes active, and another track in the same cylinder is read or written. This process continues until all tracks in the current cylinder are read or written, and then the arm assembly moves (in or out) to an adjacent cylinder. Thus, we have a natural notion of 'closeness' for blocks, which we can extend to a notion of *next* and *previous* blocks.

Exploiting this notion of next by arranging records so that they are read or written sequentially is very important for reducing the time spent in disk I/Os. Sequential access minimizes seek time and rotational delay and is much faster than random access.

RAID

Disks are potential bottlenecks for system performance and storage system reliability. Even though disk performance has been improving continuously, microprocessor performance has advanced much more rapidly. The performance of microprocessors has improved at about 50 percent or more per year, but disk access times have improved at a rate of about 10 percent per year and disk transfer rates at a rate of about 20 percent per year. In addition, since disks contain mechanical elements, they have much higher failure rates than electronic parts of a computer system. If a disk fails, all the data stored on it is lost.

A **disk array** is an arrangement of several disks, organized so as to increase performance and improve reliability of the resulting storage system. Performance is increased through data striping. Data striping distributes data over several disks to give the impression of having a single large, very fast disk. Reliability is improved through **redundancy**. Instead of having a single copy of the data, redundant information is maintained. The redundant information is carefully organized so that in case of a disk failure, it can be used to reconstruct the contents of the failed disk. Disk arrays that implement a combination of data striping and redundancy are called **redundant arrays of independent disks**, or in short, **RAID**. Several RAID organizations, referred to as **RAID**

levels, have been proposed. Each RAID level represents a different trade-of between reliability and performance.

In the remainder of this section, we will first discuss data striping and redundancy and then introduce the RAID levels that have become industry standards.

Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that store the data requested. These disk blocks may reside on a single disk in the array or may be distributed over several disks in the array. Then the set of blocks is retrieved from the disk(s) involved. Thus, how we distribute the data over the disks in the array influences how many disks are involved when an I/O request is processed.

Redundancy schemes: Alternatives to the parity scheme include schemes based on **Hamming codes** and **Reed-Solomon codes**. In addition to recovery from single disk failures, Hamming codes can identify which disk has failed. Reed-Solomon codes can recover from up to two simultaneous disk failures. A detailed discussion of these schemes is beyond the scope of our discussion here.

In **data striping**, the data is segmented into equal-size partitions that are distributed over multiple disks. The size of a partition is called the **striping unit**. The partitions are usually distributed using a round robin algorithm: If the disk array consists of D disks, then partition i is written onto disk $i \bmod D$.

As an example, consider a striping unit of a bit. Since any D successive data bits are spread over all D data disks in the array, all I/O requests involve all disks in the array. Since the smallest unit of transfer from a disk is a block, each I/O request involves transfer of at least D blocks. Since we can read the D blocks from the D disks in parallel, the transfer rate of each request is D times the transfer rate of a single disk; each request uses the aggregated bandwidth of all disks in the array. But the disk access time of the array is basically the access time of a single disk since all disk heads have to move for all requests. Therefore, for a disk array with a striping unit of a single bit, the number of requests per time unit that the array can process and the average response time for each individual request are similar to that of a single disk.

As another example, consider a striping unit of a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array. If there are many I/O requests of the size of a disk block and the requested blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request. Since we distributed the striping partitions round-robin, large requests of the size of many contiguous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate to the aggregated bandwidth of all D disks.

Redundancy

While having more disks increases storage system performance, it also lowers overall storage system reliability. Assume that the **mean-time-to-failure**, or MTTF, of a single disk is 50*;* 000 hours (about 5*:*7 years). Then, the MTTF of an array of 100 disks is only 50000/100 = 500 hours or about 21 days, assuming that failures occur independently and that the failure probability of a disk does not change over time. (Actually, disks have a higher failure probability early and late in their lifetimes. Early failures are often due to undetected manufacturing defects; late failures occur since the disk wears out. Failures do not occur independently either: consider a fire in the building, an earthquake, or purchase of a set of disks that come from a ‘bad’ manufacturing batch.)

Reliability of a disk array can be increased by storing redundant information. If a disk failure occurs, the redundant information is used to reconstruct the data on the failed

disk. Redundancy can immensely increase the MTTF of a disk array. When incorporating redundancy into a disk array design, we have to make two choices. First, we have to decide where to store the redundant information. We can either store the redundant information on a small number of **check disks** or we can distribute the redundant information uniformly over all disks.

The second choice we have to make is how to compute the redundant information. Most disk arrays store parity information. In the **parity scheme** , an extra check disk contains information that can be used to recover from failure of any one disk in the array. Assume that we have a disk array with D disks and consider the first bit on each data disk. Suppose that i of the D data bits are one. The first bit on the check disk is set to one if i is odd, otherwise it is set to zero. This bit on the check disk is called the **parity** of the data bits. The check disk contains parity information for each set of corresponding D data bits.

To recover the value of the first bit of a failed disk we first count the number of bits that are one on the $D - 1$ nonfailed disks; let this number be j. If j is odd and the parity bit is one, or if j is even and the parity bit is zero, then the value of the bit on the failed disk must have been zero. Otherwise, the value of the bit on the failed disk must have been one. Thus, with parity we can recover from failure of any one disk. Reconstruction of the lost information involves reading all data disks and the check disk.

For example, with an additional 10 disks with redundant information, the MTTF of our example storage system with 100 data disks can be increased to more than 250 years! What is more important, a large MTTF implies a small failure probability during the actual usage time of the storage system, which is usually much smaller than the reported lifetime or the MTTF. (Who actually uses 10-year-old disks?)

In a RAID system, the disk array is partitioned into **reliability groups** , where a reliability group consists of a set of *data disks* and a set of *check disks*. The number of check disks depends on the RAID level chosen. In the remainder of this section, we assume for ease of explanation that there is only one reliability group. The reader should keep in mind that actual RAID implementations consist of several reliability groups, and that the number of groups plays a role in the overall reliability of the resulting storage system.

Levels of Redundancy

Throughout the discussion of the different RAID levels, we consider sample data that would just fit on four disks. That is, without any RAID technology our storage system would consist of exactly four data disks. Depending on the RAID level chosen, the number of additional disks varies from zero to four.

Level 0: Nonredundant

A RAID Level 0 system uses data striping to increase the maximum bandwidth available. No redundant information is maintained. While being the solution with the lowest cost, reliability is a problem, since the MTTF decreases linearly with the number of disk drives in the array. RAID Level 0 has the best write performance of all RAID levels, because absence of redundant information implies that no redundant information needs to be updated! Interestingly, RAID Level 0 does not have the best read performance of all RAID levels, since systems with redundancy have a choice of scheduling disk accesses as explained in the next section.

In our example, the RAID Level 0 solution consists of only four data disks. Independent of the number of data disks, the effective space utilization for a RAID Level 0 system is always 100 percent.

Level 1: Mirrored

A RAID Level 1 system is the most expensive solution. Instead of having one copy of the data, two identical copies of the data on two different disks are maintained. This type of redundancy is often called **mirroring**. Every write of a disk block involves a write on both disks. These writes may not be performed simultaneously, since a global system failure (e.g., due to a power outage) could occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, we always write a block on one disk first and then write the other copy on the mirror disk. Since two copies of each block exist on different disks, we can distribute reads between the two disks and allow *parallel reads* of different disk blocks that conceptually reside on the same disk. A read of a block can be scheduled to the disk that has the smaller expected access time. RAID Level 1 does not stripe the data over different disks, thus the transfer rate for a single request is comparable to the transfer rate of a single disk.

In our example, we need four data and four check disks with mirrored data for a RAID Level 1 implementation. The effective space utilization is 50 percent, independent of the number of data disks.

Level 0+1: Striping and Mirroring

RAID Level 0+1 sometimes also referred to as RAID level 10 combines striping and mirroring. Thus, as in RAID Level 1, read requests of the size of a disk block can be scheduled both to a disk or its mirror image. In addition, read requests of the size of several contiguous blocks benefit from the aggregated bandwidth of all disks. The cost for writes is analogous to RAID Level 1.

As in RAID Level 1, our example with four data disks requires four check disks and the effective space utilization is always 50 percent.

Level 2: Error-Correcting Codes

In RAID Level 2 the striping unit is a single bit. The redundancy scheme used is Hamming code. In our example with four data disks, only three check disks are needed. In general, the number of check disks grows logarithmically with the number of data disks.

Striping at the bit level has the implication that in a disk array with D data disks, the smallest unit of transfer for a read is a set of D blocks. Thus, Level 2 is good for workloads with many large requests since for each request the aggregated bandwidth of all data disks is used. But RAID Level 2 is bad for small requests of the size of an individual block for the same reason. A write of a block involves reading D blocks into main memory, modifying $D + C$ blocks and writing $D + C$ blocks to disk, where C is the number of check disks. This sequence of steps is called a *read-modify-write* cycle.

For a RAID Level 2 implementation with four data disks, three check disks are needed. Thus, in our example the effective space utilization is about 57 percent. The effective space utilization increases with the number of data disks. For example, in a setup with 10 data disks, four check disks are needed and the effective space utilization is 71 percent. In a setup with 25 data disks, five check disks are required and the effective space utilization grows to 83 percent.

Level 3: Bit-Interleaved Parity

While the redundancy schema used in RAID Level 2 improves in terms of cost upon RAID Level 1, it keeps more redundant information than is necessary. Hamming code, as used in RAID Level 2, has the advantage of being able to identify which disk has failed. But disk controllers can easily detect which disk has failed. Thus, the check disks do not need to contain information to identify the failed disk. Information to recover the lost data is sufficient. Instead of using several disks to store Hamming code, RAID Level 3 has a

single check disk with parity information. Thus, the reliability overhead for RAID Level 3 is a single disk, the lowest overhead possible.

The performance characteristics of RAID Level 2 and RAID Level 3 are very similar. RAID Level 3 can also process only one I/O at a time, the minimum transfer unit is *D* blocks, and a write requires a read-modify-write cycle.

Level 4: Block-Interleaved Parity

RAID Level 4 has a striping unit of a disk block, instead of a single bit as in RAID Level 3. Block-level striping has the advantage that read requests of the size of a disk block can be served entirely by the disk where the requested block resides. Large read requests of several disk blocks can still utilize the aggregated bandwidth of the *D* disks.

The write of a single block still requires a read-modify-write cycle, but only one data disk and the check disk are involved. The parity on the check disk can be updated without reading all *D* disk blocks, because the new parity can be obtained by noticing the differences between the old data block and the new data block and then applying the difference to the parity block on the check disk:

NewParity = (OldData XOR NewData) XOR OldParity

The read-modify-write cycle involves reading of the old data block and the old parity block, modifying the two blocks, and writing them back to disk, resulting in four disk accesses per write. Since the check disk is involved in each write, it can easily become the bottleneck.

RAID Level 3 and 4 configurations with four data disks require just a single check disk. Thus, in our example, the effective space utilization is 80 percent. The effective space utilization increases with the number of data disks, since always only one check disk is necessary.

Level 5: Block-Interleaved Distributed Parity

RAID Level 5 improves upon Level 4 by distributing the parity blocks uniformly over all disks, instead of storing them on a single check disk. This distribution has two advantages. First, several write requests can potentially be processed in parallel, since the bottleneck of a unique check disk has been eliminated. Second, read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas in systems with a dedicated check disk the check disk never participates in reads.

A RAID Level 5 system has the best performance of all RAID levels with redundancy for small and large read and large write requests. Small writes still require a read-modify-write cycle and are thus less efficient than in RAID Level 1.

In our example, the corresponding RAID Level 5 system has 5 disks overall and thus the effective space utilization is the same as in RAID levels 3 and 4.

Level 6: P+Q Redundancy

The motivation for RAID Level 6 is the observation that recovery from failure of a single disk is not sufficient in very large disk arrays. First, in large disk arrays, a second disk might fail before replacement of an already failed disk could take place. In addition, the probability of a disk failure during recovery of a failed disk is not negligible.

A RAID Level 6 system uses Reed-Solomon codes to be able to recover from up to two simultaneous disk failures. RAID Level 6 requires (conceptually) two check disks, but it also uniformly distributes redundant information at the block level as in RAID Level 5. Thus, the performance characteristics for small and large read requests and for large write requests are analogous to RAID Level 5. For small writes, the read-modify-write procedure

involves six instead of four disks as compared to RAID Level 5, since two blocks with redundant information need to be updated.

For a RAID Level 6 system with storage capacity equal to four data disks, six disks are required. Thus, in our example, the effective space utilization is 66 percent.

Choice of RAID Levels

If data loss is not an issue, RAID Level 0 improves overall system performance at the lowest cost. RAID Level 0+1 is superior to RAID Level 1. The main application areas for RAID Level 0+1 systems are small storage subsystems where the cost of mirroring is moderate. Sometimes RAID Level 0+1 is used for applications that have a high percentage of writes in their workload, since RAID Level 0+1 provides the best write performance. RAID levels 2 and 4 are always inferior to RAID levels 3 and 5, respectively. RAID Level 3 is appropriate for workloads consisting mainly of large transfer requests of several contiguous blocks. The performance of a RAID Level 3 system is bad for workloads with many small requests of a single disk block. RAID Level 5 is a good general-purpose solution. It provides high performance for large requests as well as for small requests. RAID Level 6 is appropriate if a higher level of reliability is required.

Disk Space Management

The lowest level of software in the DBMS architecture discussed in Lecture 1, called the **disk space manager** , manages space on disk. Abstractly, the disk space manager supports the concept of a **page** as a unit of data, and provides commands to allocate or deallocate a page and read or write a page. The size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be done in one disk I/O.

It is often useful to allocate a sequence of pages as a *contiguous* sequence of blocks to hold data that is frequently accessed in sequential order. This capability is essential for exploiting the advantages of sequentially accessing disk blocks, which we discussed earlier. Such a capability, if desired, must be provided by the disk space manager to higher-level layers of the DBMS.

Thus, the disk space manager hides details of the underlying hardware (and possibly the operating system) and allows higher levels of the software to think of the data as a collection of pages.

Keeping Track of Free Blocks

A database grows and shrinks as records are inserted and deleted over time. The disk space manager keeps track of which disk blocks are in use, in addition to keeping track of which pages are on which disk blocks. Although it is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create ‘holes.’

One way to keep track of block usage is to maintain a list of free blocks. As blocks are deallocated (by the higher-level software that requests and uses these blocks), we can add them to the free list for future use. A pointer to the first block on the free block list is stored in a known location on disk.

A second way is to maintain a bitmap with one bit for each disk block, which indicates whether a block is in use or not. A bitmap also allows very fast identification and allocation of contiguous areas on disk. This is difficult to accomplish with a linked list approach.

Using OS File Systems to Manage Disk Space

Operating systems also manage space on disk. Typically, an operating system supports the abstraction of a *file as a sequence of bytes* . The OS manages space on the

disk and translates requests such as “Read byte *i* of file *f* ” into corresponding low-level instructions: “Read block *m* of track *t* of cylinder *c* of disk *d*.” A database disk space manager could be built using OS files. For example, the entire database could reside in one or more OS files for which a number of blocks are allocated (by the OS) and initialized. The disk space manager is then responsible for managing the space in these OS files.

Many database systems do not rely on the OS file system and instead do their own disk management, either from scratch or by extending OS facilities. The reasons are practical as well as technical. One practical reason is that a DBMS vendor who wishes to support several OS platforms cannot assume features specific to any OS, for portability, and would therefore try to make the DBMS code as self-contained as possible. A technical reason is that on a 32-bit system, the largest file size is 4 GB, whereas a DBMS may want to access a single file larger than that. A related problem is that typical OS files cannot span disk devices, which is often desirable or even necessary in a DBMS.

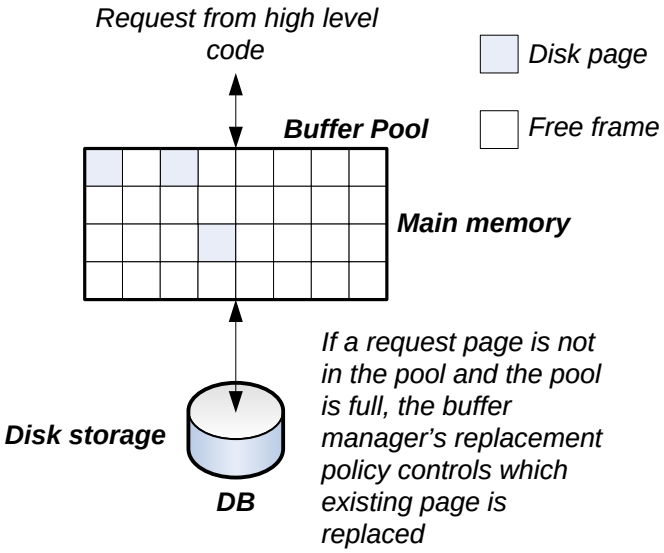
Buffer Manager

To understand the role of the buffer manager, consider a simple example. Suppose that the database contains 1,000,000 pages, but only 1,000 pages of main memory are available for holding data. Consider a query that requires a scan of the entire file. Because all the data cannot be brought into main memory at one time, the DBMS must bring pages into main memory as they are needed and, in the process, decide what existing page in main memory to replace to make space for the new page. The policy used to decide which page to replace is called the **replacement policy**.

In terms of the DBMS architecture presented in lecture 1, the **buffer manager** is the software layer that is responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the **buffer pool**. The main memory pages in the buffer pool are called **frames**; it is convenient to think of them as slots that can hold a page (that usually resides on disk or other secondary storage media).

Higher levels of the DBMS code can be written without worrying about whether data pages are in memory or not; they ask the buffer manager for the page, and it is brought into a frame in the buffer pool if it is not already there. Of course, the higher-level code that requests a page must also release the page when it is no longer needed, by informing the buffer manager, so that the frame containing the page can be reused.

The higher-level code must also inform the buffer manager if it modifies the requested page; the buffer manager then makes sure that the change is propagated to the copy of the page on disk. Buffer management is illustrated in following figure.



In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information, and two variables for each frame in the pool: *pin count* and *dirty*. The number of times that the page currently in a given frame has been requested but not release - the number of current users of the page-is recorded in the *pin-count* variable for that frame. The boolean variable *dirty* indicates whether the page has been modified since it was brought into the buffer pool from disk. Initially, the *pin count* for every frame is set to 0, and the *dirty* bits are turned off.

When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page, and if so increments the *pin-count* of that frame. If the page is not in the pool, the buffer manager brings it in as follows:
   (a) Chooses a frame for replacement, using the replacement policy, and increments its *pin-count*.
   (b) If the *dirty* bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame).
   (c) Reads the requested page into the replacement frame.
2. Returns the (main memory) address of the frame containing the requested page to the requestor.

Incrementing *pin-count* is often called **pinning** the requested page in its frame. When the code that calls the buffer manager and requests the page subsequently calls the buffer manager and releases the page, the *pin-count* of the frame containing the requested page is decremented. This is called **unpinning** the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the *dirty* bit for the frame is set. The buffer manager will not read another page into a frame until its *pin-count* becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool, and if a free frame is not available in the buffer pool, a frame with *pin-count* 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager's replacement policy.

When a page is eventually chosen for replacement, if the *dirty* bit is not set, it means that the page has not been modified since being brought into main memory. Thus, there is no need to write the page back to disk; the copy on disk is identical to the copy in the frame, and the frame can simply be overwritten by the newly requested page. Otherwise, the modifications to the page must be propagated to the copy on disk.

If there is no page in the buffer pool with *pin-count* 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before

responding to the page request. In practice, the transaction requesting the page may simply be aborted in this situation! So pages should be released-by the code that calls the buffer manager to request the page-as soon as possible.

A good question to ask at this point is “What if a page is requested by several different transactions?" That is, what if the page is requested by programs executing independently on behalf of different users? There is the potential for such programs to make conflicting changes to the page. The locking protocol (enforced by higher-level DBMS code, in particular the transaction manager) ensures that each transaction obtains a shared or exclusive lock before requesting a page to read or modify. Two different transactions cannot hold an exclusive lock on the same page at the same time; this is how conflicting changes are prevented. The buffer manager simply assumes that the appropriate lock has been obtained before a page is requested.

Files and Indexes

We now turn our attention from the way pages are stored on disk and brought into main memory to the way pages are used to store records and organized into logical collections or **files**. Higher levels of the DBMS code treat a page as effectively being a collection of records, ignoring the representation and storage details. In fact, the concept of a collection of records is not limited to the contents of a single page; a **file of records** is a collection of records that may reside on several pages.

The basic file structure that we consider, called a *heap file* , stores records in random order and supports retrieval of all records or retrieval of a particular record specified by its rid. Sometimes we want to retrieve records by specifying some condition on the fields of desired records, for example, “Find all employee records with *age* 35." To speed up such selections, we can build auxiliary data structures that allow us to quickly find the rids of employee records that satisfy the given selection condition.

Heap Files

The simplest file structure is an unordered file or **heap file**. The data in the pages of a heap file is not ordered in any way, and the only guarantee is that one can retrieve all records in the file by repeated requests for the next record. Every record in the file has a unique rid, and every page in a file is of the same size.

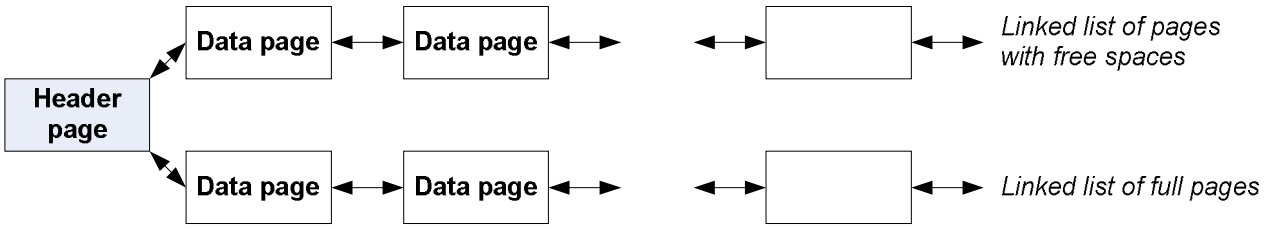
Supported operations on a heap file include *create* and *destroy* files, *insert* a record, *delete* a record with a given rid, *get* a record with a given rid, and *scan* all records in the file. To get or delete a record with a given rid, note that we must be able to find the id of the page containing the record, given the id of the record.

We must keep track of the pages in each heap file in order to support scans, and we must keep track of pages that contain free space in order to implement insertion efficiently. We discuss two alternative ways to maintain this information. In each of these alternatives, pages must hold two pointers (which are page ids) for file-level bookkeeping in addition to the data.

Linked List of Pages

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMS can remember where the first page is located by maintaining a table containing pairs of á*heap file name; page* 1 *addr*ñ in a known location on disk. We call the first page of the file the *header page*.

An important task is to maintain information about empty slots created by deleting a record from the heap file. This task has two distinct parts: how to keep track of free space within a page and how to keep track of pages that have some free space.

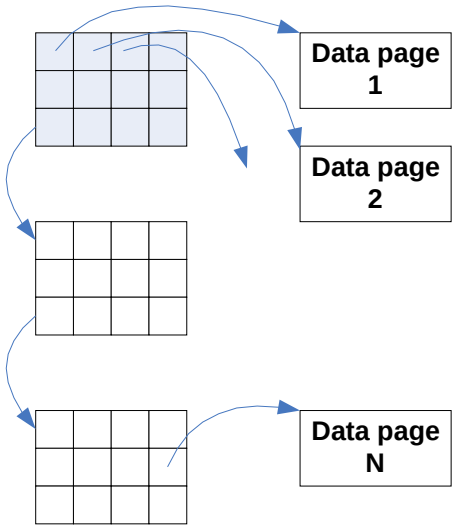


If a new page is required, it is obtained by making a request to the disk space manager and then added to the list of pages in the file (probably as a page with free space, because it is unlikely that the new record will take up all the space on the page). If a page is to be deleted from the heap file, it is removed from the list and the disk space manager is told to deallocate it. (Note that the scheme can easily be generalized to allocate or deallocate a sequence of several pages and maintain a doubly linked list of these page sequences.)

One disadvantage of this scheme is that virtually all pages in a file will be on the free list if records are of variable length, because it is likely that every page has at least a few free bytes. To insert a typical record, we must retrieve and examine several pages on the free list before we find one with enough free space. The directory-based heap file organization that we discuss next addresses this problem.

Directory of Pages

An alternative to a linked list of pages is to maintain a **directory of pages** . The DBMS must remember where the first directory page of each heap file is located.



Directory

Each directory entry identifies a page (or a sequence of pages) in the heap file. As the heap file grows or shrinks, the number of entries in the directory and possibly the number of pages in the directory itself grows or shrinks correspondingly. Note that since each directory entry is quite small in comparison to a typical page, the size of the directory is likely to be very small in comparison to the size of the heap file.

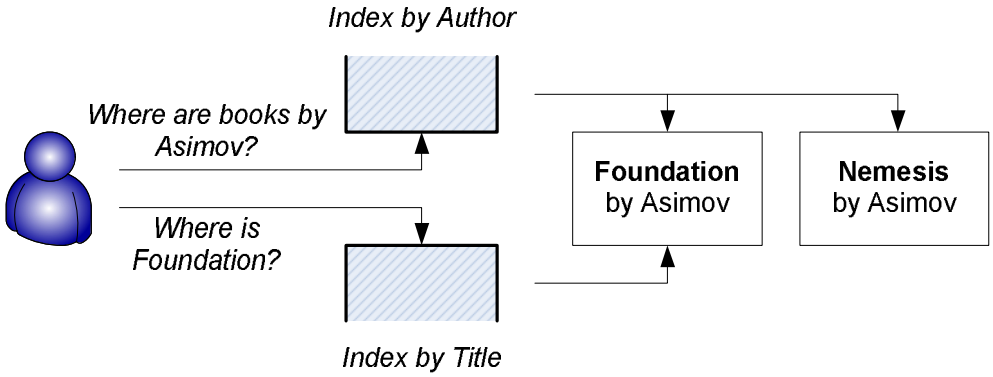
Free space can be managed by maintaining a bit per entry, indicating whether the corresponding page has any free space, or a count per entry, indicating the amount of free space on the page. If the file contains variable-length records, we can examine the free space count for an entry to determine if the record will fit on the page pointed to by the

entry. Since several entries fit on a directory page, we can efficiently search for a data page with enough space to hold a record that is to be inserted.

Introduction to Indexes

Sometimes we want to find all records that have a given value in a particular field. If we can find the rids of all such records, we can locate the page containing each record from the record's rid; however, the heap file organization does not help us to find the rids of such records. An **index** is an auxiliary data structure that is intended to help us find rids of records that meet a selection condition.

Consider how you locate a desired book in a library. You can search a collection of index cards, sorted on author name or book title, to find the call number for the book. Because books are stored according to call numbers, the call number enables you to walk to the shelf that contains the book you need. Observe that an index on author name cannot be used to locate a book by title, and vice versa; each index speeds up certain kinds of searches, but not all.



The same ideas apply when we want to support efficient retrieval of a desired subset of the data in a file. From an implementation standpoint, an index is just another kind of file, containing records that direct traffic on requests for data records. Every index has an associated **search key**, which is a collection of one or more fields of the file of records for which we are building the index; any subset of the fields can be a search key. We sometimes refer to the file of records as the **indexed file**.

An index is designed to speed up equality or range selections on the search key. For example, if we wanted to build an index to improve the efficiency of queries about employees of a given age, we could build an index on the *age* attribute of the employee dataset. The records stored in an index file, which we refer to as **entries** to avoid confusion with data records, allow us to find data records with a given search key value. In our example the index might contain ‹*age, rid*› pairs, where *rid* identifies a data record.

The pages in the index file are organized in some way that allows us to quickly locate those entries in the index that have a given search key value. For example, we have to find entries with *age*≥30 (and then follow the rids in the retrieved entries) in order to find employee records for employees who are older than 30. Organization techniques, or data structures, for index files are called **access methods**, and several are known, including B+ trees and hash-based structures. B+ tree index files and hash-based index files are built using the page allocation and manipulation facilities provided by the disk space manager, just like heap files.

Cost Model

In this section we introduce a cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We will use the following notation and assumptions in our analysis. There are *B* data pages with *R* records per page. The average time to read or write a disk page is *D*, and the average time to process a record (e.g., to compare a field value to a selection constant) is *C*. In the hashed file organization, we will use a function, called a *hash function*, to map a record into a range of numbers; the time required to apply the hash function to a record is *H*.

Typical values today are *D* = 15 milliseconds, *C* and *H* = 100 nanoseconds; we therefore expect the cost of I/O to dominate. This conclusion is supported by current hardware trends, in which CPU speeds are steadily rising, whereas disk speeds are not increasing at a similar pace. On the other hand, as main memory sizes increase, a much larger fraction of the needed pages are likely to fit in memory, leading to fewer I/O requests.

We therefore use the number of disk page I/Os as our cost metric.

- We emphasize that real systems must consider other aspects of cost, such as CPU costs (and transmission costs in a distributed database). However, our goal is primarily to present the underlying algorithms and to illustrate how costs can be estimated. Therefore, for simplicity, we have chosen to concentrate on only the I/O component of cost. Given the fact that I/O is often (even typically) the dominant component of the cost of database operations, considering I/O costs gives us a good first approximation to the true costs.
- Even with our decision to focus on I/O costs, an accurate model would be too complex for our purposes of conveying the essential ideas in a simple way. We have therefore chosen to use a simplistic model in which we just count the number of pages that are read from or written to disk as a measure of I/O. We have ignored the important issue of **blocked access** typically, disk systems allow us to read a block of contiguous pages in a single I/O request. The cost is equal to the time required to **seek** the first page in the block and to **transfer** all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively: We would have an additional seek cost for each page in the block.

We now compare the costs of some simple operations for three basic file organizations: *files of randomly ordered records, or heap files* ; *files sorted on a sequence of fields*; and *files that are hashed on a sequence of fields*. For sorted and hashed files, the sequence of fields (e.g., *salary*, *age*) on which the file is sorted or hashed is called the **search key**. Note that the search key for an index can be any sequence of one or more fields; it need not uniquely identify records. We observe that there is an unfortunate overloading of the term *key* in the database literature. A *primary key* or *candidate key* is unrelated to the concept of a search key.

Our goal is to emphasize how important the choice of an appropriate file organization can be. The operations that we consider are described below.

- **Scan:** Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).
- **Search with equality selection:** Fetch all records that satisfy an equality selection, for example, “Find the Students record for the student with *sid* 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

- **Search with range selection:** Fetch all records that satisfy a range selection, for example, “Find all Students records with *name* alphabetically after ‘Smith.’ "
- **Insert:** Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.
- **Delete:** Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Heap Files

Scan: The cost is $B(D + RC)$ because we must retrieve each of B pages taking time D per page, and for each page, process R records taking time C per record.

Search with equality selection: Suppose that we know in advance that exactly one record matches the desired equality selection, that is, the selection is specified on a candidate key. On average, we must scan half the file, assuming that the record exists and the distribution of values in the search field is uniform. For each retrieved data page, we must check all records on the page to see if it is the desired record. The cost is $0:5B(D + RC)$. If there is no record that satisfies the selection, however, we must scan the entire file to verify this.

If the selection is not on a candidate key field (e.g. “Find students aged 18"), we always have to scan the entire file because several records with *age* = 18 could be dispersed all over the file, and we have no idea how many such records exist.

Search with range selection: The entire file must be scanned because qualifying records could appear anywhere in the file and we do not know how many qualifying records exist. The cost is $B(D + RC)$.

Insert: We assume that records are always inserted at the end of the file. We must fetch the last page in the file, add the record, and write the page back. The cost is $2D + C$.

Delete: We must find the record, remove the record from the page, and write the modified page back. We assume that no attempt is made to compact the file to reclaim the free space created by deletions, for simplicity. The cost is the cost of searching plus $C + D$.

We assume that the record to be deleted is specified using the record id. Since the page id can easily be obtained from the record id, we can directly read in the page. The cost of searching is therefore D.

If the record to be deleted is specified using an equality or range condition on some fields, the cost of searching is given in our discussion of equality and range selections. The cost of deletion is also affected by the number of qualifying records, since all pages containing such records must be modified.

Sorted Files

Scan: The cost is $B(D + RC)$ because all pages must be examined. Note that this case is no better or worse than the case of unordered files. However, the order in which records are retrieved corresponds to the sort order.

Search with equality selection: We assume that the equality selection is specified on the field by which the file is sorted; if not, the cost is identical to that for a heap file. We can locate the first page containing the desired record or records, should any qualifying records exist, with a binary search in $log_2 B$ steps. (This analysis assumes that the pages in the sorted file are stored sequentially, and we can retrieve the i^{th} page on the file directly in one disk I/O. This assumption is not valid if, for example, the sorted file is implemented

as a heap file using the linked-list organization, with pages in the appropriate sorted order.) Each step requires a disk I/O and two comparisons. Once the page is known, the first qualifying record can again be located by a binary search of the page at a cost of *Clog*2*R*. The cost is *Dlog*2*B* + *Clog*2*R*, which is a significant improvement over searching heap files.

If there are several qualifying records (e.g., \Find all students aged 18"), they are guaranteed to be adjacent to each other due to the sorting on *age*, and so the cost of retrieving all such records is the cost of locating the first such record (*Dlog*2*B*+*Clog*2*R*) plus the cost of reading all the qualifying records in sequential order. Typically, all qualifying records fit on a single page. If there are no qualifying records, this is established by the search for the first qualifying record, which finds the page that would have contained a qualifying record, had one existed, and searches that page.

Search with range selection: Again assuming that the range selection is on the sort field, the first record that satisfies the selection is located as it is for search with equality. Subsequently, data pages are sequentially retrieved until a record is found that does not satisfy the range selection; this is similar to an equality search with many qualifying records.

The cost is the cost of search plus the cost of retrieving the set of records that satisfy the search. The cost of the search includes the cost of fetching the first page containing qualifying, or matching, records. For small range selections, all qualifying records appear on this page. For larger range selections, we have to fetch additional pages containing matching records.

Insert: To insert a record while preserving the sort order, we must first find the correct position in the file, add the record, and then fetch and rewrite all subsequent pages (because all the old records will be shifted by one slot, assuming that the file has no empty slots). On average, we can assume that the inserted record belongs in the middle of the file. Thus, we must read the latter half of the file and then write it back after adding the new record. The cost is therefore the cost of searching to find the position of the new record plus 2 * (0.5*B*(*D* + *RC*)), that is, search cost plus *B*(*D* + *RC*).

Delete: We must search for the record, remove the record from the page, and write the modified page back. We must also read and write all subsequent pages because all records that follow the deleted record must be moved up to compact the free space. The cost is the same as for an insert, that is, search cost plus *B*(*D* + *RC*). Given the rid of the record to delete, we can fetch the page containing the record directly.

If records to be deleted are specified by equality or range condition, the cost of deletion depends on the number of qualifying records. If the condition is specified on the sort field, qualifying records are guaranteed to be contiguous due to the sorting, and the first qualifying record can be located using binary search.

Hashed Files

A simple hashed file organization enables us to locate records with a given search key value quickly, for example, “Find the Students record for Joe," if the file is hashed on the *name* field.

The pages in a hashed file are grouped into **buckets**. Given a bucket number, the hashed file structure allows us to find the **primary page** for that bucket. The bucket to which a record belongs can be determined by applying a special function called a **hash function**, to the search field(s). On inserts, a record is inserted into the appropriate bucket, with additional ‘overflow’ pages allocated if the primary page for the bucket becomes full. The overflow pages for each bucket are maintained in a linked list. To search for a record with a given search key value, we simply apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket.

This organization is called a **static hashed file**, and its main drawback is that long chains of overflow pages can develop. This can affect performance because all pages in a bucket have to be searched.

Scan: In a hashed file, pages are kept at about 80 percent occupancy (to leave some space for future insertions and minimize overflow pages as the file expands). This is achieved by adding a new page to a bucket when each existing page is 80 percent full, when records are initially organized into a hashed file structure. Thus, the number of pages, and the cost of scanning all the data pages, is about 1.25 times the cost of scanning an unordered file, that is, 1.25B(D + RC).

Search with equality selection: This operation is supported very efficiently if the selection is on the search key for the hashed file. (Otherwise, the entire file must be scanned.) The cost of identifying the page that contains qualifying records is H; assuming that this bucket consists of just one page (i.e., no overflow pages), retrieving it costs D. The cost is $H + D + 0.5RC$ if we assume that we find the record after scanning half the records on the page. This is even lower than the cost for sorted files. If there are several qualifying records, or none, we still have to retrieve just one page, but we must scan the entire page.

Note that the hash function associated with a hashed file maps a record to a bucket based on the values in *all* the search key fields; if the value for any one of these fields is not specified, we cannot tell which bucket the record belongs to. Thus, if the selection is not an equality condition on all the search key fields, we have to scan the entire file.

Search with range selection: The hash structure offers no help; even if the range selection is on the search key, the entire file must be scanned. The cost is 1.25B(D + RC).

Insert: The appropriate page must be located, modified, and then written back. The cost is the cost of search plus C + D.

Delete: We must search for the record, remove it from the page, and write the modified page back. The cost is again the cost of search plus $C + D$ (for writing the modified page).

If records to be deleted are specified using an equality condition on the search key, all qualifying records are guaranteed to be in the same bucket, which can be identified by applying the hash function.

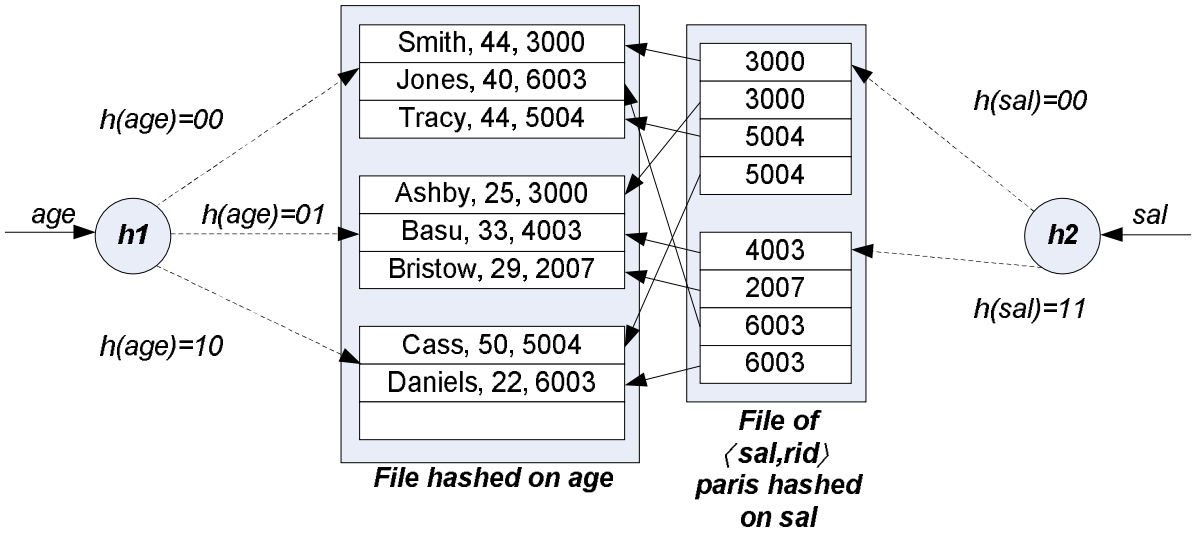
Choosing a File Organization

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0.5BD	BD	2D	*Search*+D
Sorted	BD	D$\log_2$B	D$\log_2$B+#*matches*	*Search*+BD	*Search*+BD
Hashed	1.25BD	D	1.25BD	2D	*Search*+D

In summary, the table demonstrates that no one file organization is uniformly superior in all situations. An unordered file is best if only full file scans are desired. A hashed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired. The organizations that we have studied here can be improved on the problems of overflow pages in static hashing can be overcome by using dynamic hashing structures, and the high cost of inserts and deletes in a sorted file can be overcome by using tree-structured indexes but the main observation, that the choice of an appropriate file organization depends on how the file is commonly used, remains valid.

Overview of Indexes

As we noted earlier, an index on a file is an auxiliary structure designed to speed up operations that are not efficiently supported by the basic organization of records in that file. An index can be viewed as a collection of **data entries**, with an efficient way to locate all data entries with search key value *k*. Each such data entry, which we denote as *k**, contains enough information to enable us to retrieve (one or more) data records with search key value *k*. (Note that a data entry is, in general, different from a data record!) The following figure shows an index with search key *sal* that contains á*sal, rid*ñ pairs as data entries. The *rid* component of a data entry in this index is a pointer to a record with search key value *sal*.



Two important questions to consider are:

1. How are data entries organized in order to support efficient retrieval of data entries with a given search key value?
2. Exactly what is stored as a data entry?

One way to organize data entries is to hash data entries on the search key. In this approach, we essentially treat the collection of data entries as a file of records, hashed on the search key. This is how the index on *sal* shown in previous figure is organized. The hash function *h* for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier. Another way to organize data entries is to build a data structure that directs a search for data entries. Several index data structures are known that allow us to efficiently find data entries with a given search key value.

Properties of Indexes

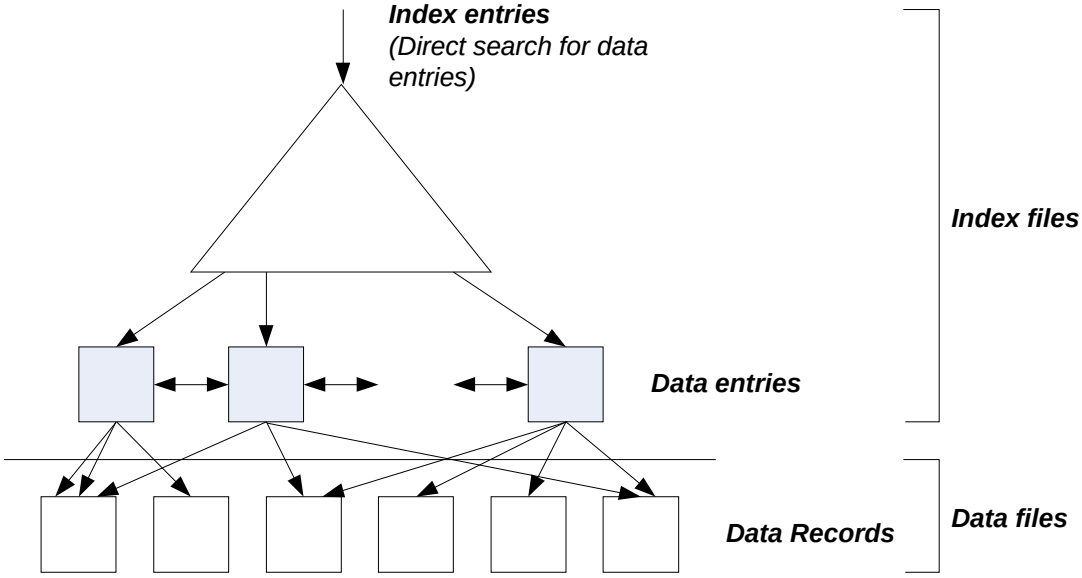
In this section, we discuss some important properties of an index that affect the efficiency of searches using the index.

Clustered versus Unclustered Indexes

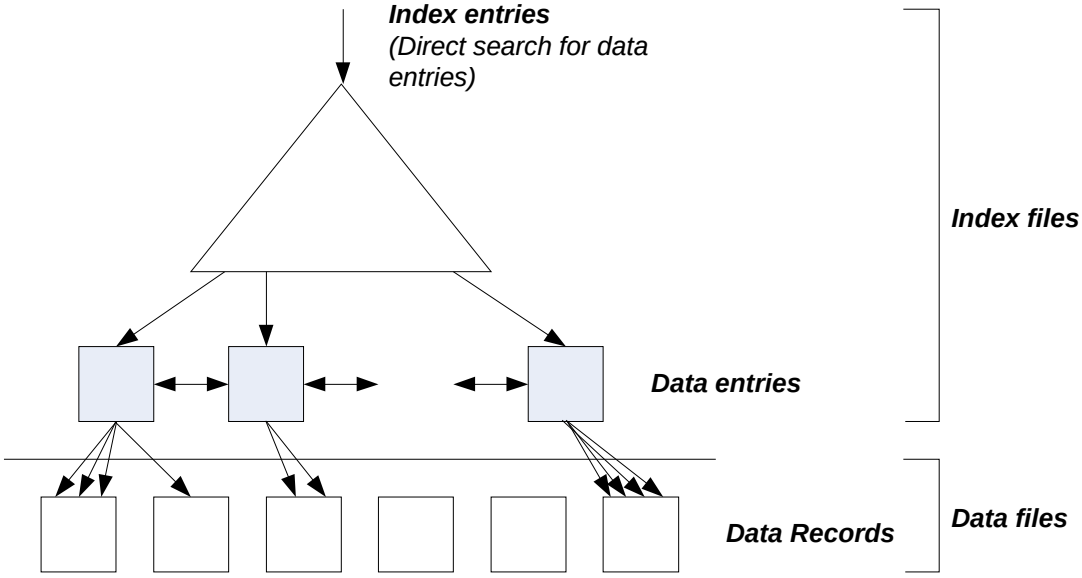
When a file is organized so that the ordering of data records is the same as or closes to the ordering of data entries in some index, we say that the index is **clustered**. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or Alternative (3) can be a clustered index only if the data records are sorted on the search key field. Otherwise, the order of the data records is random, defined purely by their physical order, and there is no reasonable way to arrange the data entries in the index in

the same order. (Indexes based on hashing do not store data entries in sorted order by search key, so a hash index is clustered only if it uses Alternative (1).)

Indexes that maintain data entries in sorted order by search key use a collection of *index entries*, organized into a tree structure, to guide searches for data entries, which are stored at the leaf level of the tree in sorted order. Clustered and unclustered tree indexes are illustrated in following figures.



UNCLUSTERED TREE INDEX



CLUSTERED TREE INDEX

In practice, data records are rarely maintained in fully sorted order, unless data records are stored in an index using Alternative (1), because of the high overhead of moving data records around to preserve the sort order as records are inserted and deleted. Typically, the records are sorted initially and each page is left with some free space to absorb future insertions. If the free space on a page is subsequently used up (by

records inserted after the initial sorting step), further insertions to this page are handled using a linked list of overflow pages. Thus, after a while, the order of records only approximates the intended sorted order, and the file must be **reorganized** (i.e., sorted afresh) to ensure good performance.

Thus, clustered indexes are relatively expensive to maintain when the file is updated. Another reason clustered indexes are expensive to maintain is that data entries may have to be moved across pages, and if records are identified by a combination of page id and slot, as is often the case, all places in the database that point to a moved record (typically, entries in other indexes for the same collection of records) must also be updated to point to the new location; these additional updates can be very time consuming.

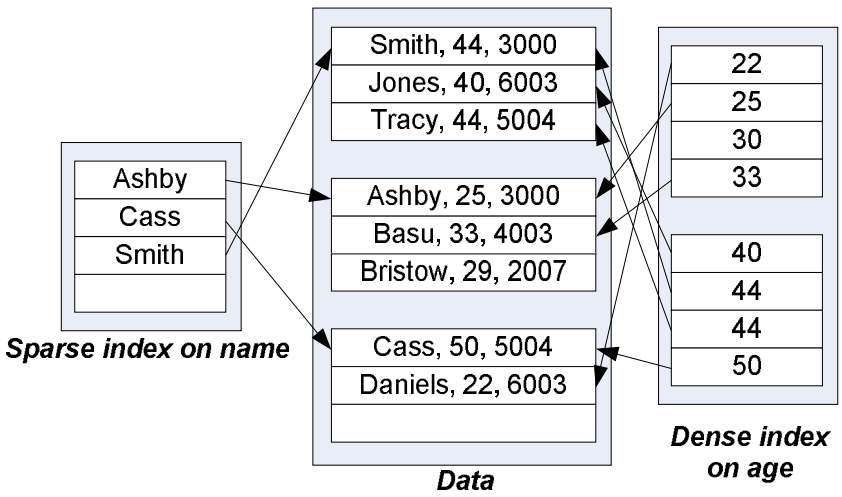
A data file can be clustered on at most one search key, which means that we can have at most one clustered index on a data file. An index that is not clustered is called an **unclustered** index; we can have several unclustered indexes on a data file. Suppose that Students records are sorted by *age*; an index on *age* that stores data entries in sorted order by *age* is a clustered index. If in addition we have an index on the *gpa* field, the latter must be an unclustered index.

The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, the rids in qualifying data entries point to a contiguous collection of records andwe need to retrieve only a few data pages. If the index is unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range selection!

Dense versus Sparse Indexes

An index is said to be **dense** if it contains (at least) one data entry for every search key value that appears in a record in the indexed file. A **sparse** index contains one entry for each page of records in the data file. Alternative (1) for data entries always leads to a dense index. Alternative (2) can be used to build either dense or sparse indexes. Alternative (3) is typically only used to build a dense index.

We illustrate sparse and dense indexes in following figure. A data file of records with three fields (*name*, *age*, and *sal*) is shown with two simple indexes on it, both of which use Alternative (2) for data entry format. The first index is a sparse, clustered index on *name*. Notice how the order of data entries in the index corresponds to the order of records in the data file. There is one data entry per page of data records. The second index is a dense, unclustered index on the *age* field. Notice that the order of data entries in the index differs from the order of data records. There is one data entry in the index per record in the data file (because we use Alternative (2)).



We cannot build a sparse index that is not clustered. Thus, we can have at most one sparse index. A sparse index is typically much smaller than a dense index. On the other hand, some very useful optimization techniques rely on an index being dense.

A data file is said to be **inverted** on a field if there is a dense secondary index on this field. A **fully inverted** file is one in which there is a dense secondary index on each field that does not appear in the primary key.

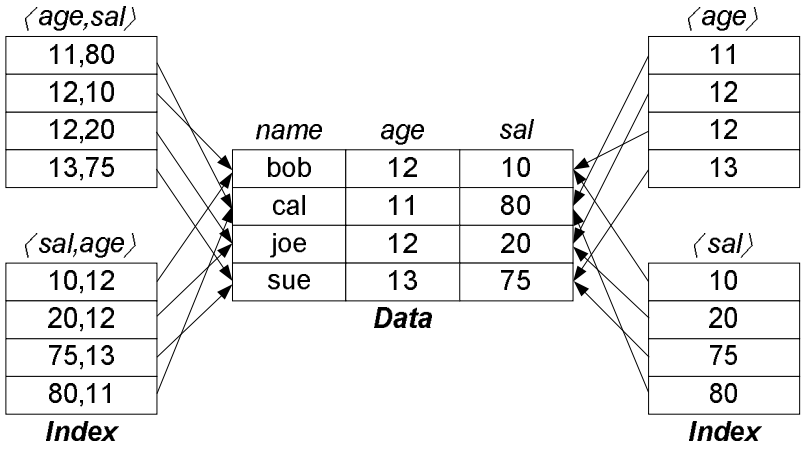
Primary and Secondary Indexes

An index on a set of fields that includes the *primary key* is called a **primary index**. An index that is not a primary index is called a **secondary** index. (The terms *primary index* and *secondary index* are sometimes used with a different meaning: An index that uses Alternative (1) is called a primary index, and one that uses Alternatives (2) or (3) is called a secondary index. We will be consistent with the definitions presented earlier, but the reader should be aware of this lack of standard terminology in the literature.)

Two data entries are said to be **duplicates** if they have the same value for the search key field associated with the index. A primary index is guaranteed not to contain duplicates, but an index on other (collections of) fields can contain duplicates. Thus, in general, a secondary index contains duplicates. If we know that no duplicates exist, that is, we know that the search key contains some candidate key, we call the index a **unique** index.

Indexes Using Composite Search Keys

The search key for an index can contain several fields; such keys are called **composite search keys** or **concatenated keys**. As an example, consider a collection of employee records, with fields *name, age*, and *sal*, stored in sorted order by *name*. The following figure illustrates the difference between a composite index with key *hage, sali*, a composite index with key *ásal, ageñ*, an index with key *age*, and an index with key *sal*. All indexes shown in the figure use Alternative (2) for data entries.



If the search key is composite, an **equality query** is one in which *each* field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with *age* = 20 and *sal* = 10. The hashed file organization supports only equality queries, since a hash function identifies the bucket containing desired records only if a value is specified for each field in the search key.

A **range query** is one in which not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with *age* = 20; this query

implies that any value is acceptable for the *sal* field. As another example of a range query, we can ask to retrieve all data entries with *age* < 30 and *sal* > 40.

Index Specification in SQL-92

The SQL-92 standard does *not* include any statement for creating or dropping index structures. In fact, the standard does not even require SQL implementations to support indexes! In practice, of course, every commercial relational DBMS supports one or more kinds of indexes.

```
CREATE INDEX IndAgeRating ON Students
WITH STRUCTURE = BTREE,
KEY = (age, gpa)
```

This specifies that a B+ tree index is to be created on the Students table using the concatenation of the *age* and *gpa* columns as the key. Thus, key values are pairs of the form á*age; gpa*ñ, and there is a distinct entry for each such pair. Once the index is created, it is automatically maintained by the DBMS adding/removing data entries in response to inserts/deletes of records on the Students relation.

Lecture 5: Transactions. Concurrency Control.

The Concept of a Transaction

A user writes data access/update programs in terms of the high-level query and update language supported by the DBMS. To understand how the DBMS handles such requests, with respect to concurrency control and recovery, it is convenient to regard an execution of a user program, or **transaction**, as a series of **reads** and **writes** of database objects:

- To read a database object, it is first brought into main memory (specifically, some frame in the buffer pool) from disk, and then its value is copied into a program variable.
- To write a database object, an in-memory copy of the object is first modified and then written to disk.

Database 'objects' are the units in which programs read or write information. The units could be pages, records, and so on, but this is dependent on the DBMS and is not central to the principles underlying concurrency control or recovery. We will consider a database to be a *fixed* collection of *independent* objects. When objects are added to or deleted from a database, or there are relationships between databases objects that we want to exploit for performance, some additional issues arise.

There are four important properties of transactions that a DBMS must ensure to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as **atomic**: either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).
2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. This property is called **consistency**, and the DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as **isolation**: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability**.

The acronym ACID is sometimes used to refer to the four properties of transactions that we have presented here: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

Consistency and Isolation

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that when run to completion by itself against a 'consistent' database instance, the transaction will leave the database in a 'consistent' state. For example, the user may (naturally!) have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts. To transfer money from one account to another, a transaction must debit one account, temporarily leaving the database inconsistent in a global sense, even though the new account balance may satisfy any integrity constraints with respect to the range of

acceptable account balances. The user's notion of a consistent database is preserved when the second account is credited with the transferred amount. If a faulty transfer program always credits the second account with one dollar less than the amount debited from the first account, the DBMS cannot be expected to detect inconsistencies due to such errors in the user program's logic.

The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order. For example, if two transactions *T*1 and *T*2 are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) *T*1 followed by executing *T*2 or executing *T*2 followed by executing *T*1. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) will also result in a consistent final database instance.

Database consistency is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation, and transaction consistency. Next, we discuss how atomicity and durability are guaranteed in a DBMS.

Atomicity and Durability

Transactions can be incomplete for three kinds of reasons. First, a transaction can be **aborted**, or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew. Second, the system may crash (e.g., because the power supply is interrupted) while one or more transactions are in progress. Third, a transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).

Of course, since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Thus a DBMS must find a way to remove the effects of partial transactions from the database, that is, it must ensure transaction atomicity: either all of a transaction's actions are carried out, or none are. A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the *log*, of all writes to the database. The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

Transactions and Schedules

A transaction is seen by the DBMS as a series, or *list*, of **actions**. The actions that can be executed by a transaction include **reads** and **writes** of *database objects* . A transaction can also be defined as a set of actions that are *partially* ordered. That is, the relative order of some of the actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a *list* of actions. Further, to keep our notation simple, we'll assume that an object *O* is always read into a program variable that is also named *O*. We can therefore denote the action of a transaction *T* reading an object *O* as $R_T(O)$; similarly, we can denote writing as $W_T(O)$. When the transaction *T* is clear from the context, we will omit the subscript. In addition to reading and writing, each transaction *must* specify as its final action either **commit** (i.e., complete successfully) or **abort** (i.e., terminate and undo all the actions carried out thus far). $Abort_T$ denotes the action of *T* aborting, and $Commit_T$ denotes *T* committing.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction *T* appear in a schedule must be the same as the order in which they appear in *T*. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in next figure shows an execution order for actions of two transactions *T*1 and *T*2. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS* . In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on.

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

Notice that the schedule in the figure does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule** . A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved that is, transactions are executed from start to finish, one by one we call the schedule a **serial schedule**.

Concurrent Execution of Transactions

Now that we've introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleavings should be allowed. In this section we consider what interleavings, or schedules, a DBMS should allow.

Motivation for Concurrent Execution

The schedule shown in last figure represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult, but is necessary for performance reasons. First, while one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle, and increases **system throughput** (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction leading to unpredictable delays in **response time**, or average time taken to complete a transaction.

Serializability

To begin with, we assume that the database designer has defined some notion of a **consistent database state** . For example, we can define a consistency criterion for a university database to be that the sum of employee salaries in each department should be less than 80 percent of the budget for that department. We require that each transaction

must **preserve** database consistency; it follows that any serial schedule that is complete will also preserve database consistency. That is, when a complete serial schedule is executed against a consistent database, the result is also a consistent database.

A **serializable schedule** over a set *S* of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over *S*. That is, the database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in *some* serial order. There are some important points to note in this definition:

- Executing the transactions serially in different orders may produce different results, but all are presumed to be acceptable; the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- The above definition of a serializable schedule does not cover the case of schedules containing aborted transactions.
- If a transaction computes a value and prints it to the screen, this is an 'effect' that is not directly captured in the state of the database. We will assume that all such values are also written into the database, for simplicity.

Some Anomalies Associated with Interleaved Execution

We now illustrate three main ways in which a schedule involving two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state. Two actions on the same data object **conflict** if at least one of them is a write. The three anomalous situations can be described in terms of when the actions of two transactions *T*1 and *T*2 conflict with each other: in a **write-read (WR) conflict** *T*2 reads a data object previously written by *T*1; we define **read-write (RW)** and **write-write (WW)** conflicts similarly.

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction *T*2 could read a database object *A* that has been modified by another transaction *T*1, which has not yet committed. Such a read is called a **dirty read**. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions *T*1 and *T*2, each of which, run alone, preserves database consistency: *T*1 transfers $100 from *A* to *B*, and *T*2 increments both *A* and *B* by 6 percent (e.g., annual interest is deposited into these two accounts). Suppose that their actions are interleaved so that (1) the account transfer program *T*1 deducts $100 from account *A*, then (2) the interest deposit program *T*2 reads the current values of accounts *A* and *B* and adds 6 percent interest to each, and then (3) the account transfer program credits $100 to account *B*. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in following figure. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of *A* written by *T*1 is read by *T*2 before *T*1 has completed all its changes.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit

R(B)	
W(B)	
Commit	

The general problem illustrated here is that *T*1 may write some value into *A* that makes the database inconsistent. As long as *T*1 overwrites this value with a 'correct' value of *A* before committing, no harm is done if *T*1 and *T*2 run in some serial order, because *T*2 would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress. Such a requirement would be too restrictive: To transfer money from one account to another, a transaction *must* debit one account, temporarily leaving the database inconsistent, and then credit the second account, restoring consistency again.

Unrepeatable Reads (RWConflicts)

The second way in which anomalous behavior could result is that a transaction *T*2 could change the value of an object *A* that has been read by a transaction *T*1, while *T*1 is still in progress. This situation causes two problems.

First, if *T*1 tries to read the value of *A* again, it will get a different result, even though it has not modified *A* in the meantime. This situation could not arise in a serial execution of two transactions; it is called an **unrepeatable read**.

Second, suppose that both *T*1 and *T*2 read the same value of *A*, say, 5, and then *T*1, which wants to increment *A* by 1, changes it to 6, and *T*2, which wants to decrement *A* by 1, decrements the value that it read (i.e., 5) and changes *A* to 4. Running these transactions in any serial order should leave *A* with a final value of 5; thus, the interleaved execution leads to an inconsistent state. The underlying problem here is that although *T*2's change is not directly read by *T*1, it invalidates *T*1's assumption about the value of *A*, which is the basis for some of *T*1's subsequent actions.

Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction *T*2 could overwrite the value of an object *A*, which has already been modified by a transaction *T*1, while *T*1 is still in progress. Even if *T*2 does not read the value of *A* written by *T*1, a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction *T*1 sets their salaries to \$1,000 and transaction *T*2 sets their salaries to \$2,000. If we execute these in the serial order *T*1 followed by *T* 2, both receive the salary \$2,000; the serial order *T*2 followed by *T* 1 gives each the salary \$1,000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Notice that neither transaction reads a salary value before writing it such a write is called a **blind write**, for obvious reasons.

Now, consider the following interleaving of the actions of *T*1 and *T*2: *T*1 sets Harry's salary to \$1,000, *T*2 sets Larry's salary to \$2,000, *T*1 sets Larry's salary to \$1,000, and finally *T*2 sets Harry's salary to \$2,000. The result is not identical to the result of either of the two possible serial executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted transactions. Intuitively, all actions of aborted transactions are to be undone, and we can therefore

imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A **serializable schedule** over a set *S* of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed* transactions in *S*.

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program *T*1 deducts $100 from account *A*, then (2) an interest deposit program *T*2 reads the current values of accounts *A* and *B* and adds 6 percent interest to each, then commits, and then (3) *T*1 is aborted. The corresponding schedule is shown in Figure 18.3. Now, *T*2 has read a value for *A* that should never have been there! (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If *T*2 had not yet committed, we could deal with the situation by *cascading* the abort of *T*1 and also aborting *T*2; this process would recursively abort any transaction that read data written by *T*2, and so on. But *T*2 has already committed, and so we cannot undo its actions! We say that such a schedule is *unrecoverable*. A **recoverable schedule** is one in which transactions commit only after (and if!) all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to **avoid cascading aborts**.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction *T*2 overwrites the value of an object *A* that has been modified by a transaction *T*1, while *T*1 is still in progress, and *T*1 subsequently aborts. All of *T*1's changes to database objects are undone by restoring the value of any object that it modified to the value of the object before *T*1's changes. When *T*1 is aborted, and its changes are undone in this manner, *T*2's changes are lost as well, even if *T*2 decides to commit. So, for example, if *A* originally had the value 5, then was changed by *T*1 to 6, and by *T*2 to 7, if *T*1 now aborts, the value of *A* becomes 5 again. Even if *T*2 commits, its change to *A* is inadvertently lost. A concurrency control technique called Strict 2PL (Strict Two-Phase Locking) can prevent this problem.

Lock-Based Concurrency Control

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol* to achieve this. A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called *Strict Two-Phase Locking* , or *Strict 2PL*, has two rules. The first rule is

(1) If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is:

(2) All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details.

In effect the locking protocol allows only 'safe' interleavings of transactions. If two transactions access completely independent parts of the database, they will be able to concurrently obtain the locks that they need and proceed merrily on their ways. On the other hand, if two transactions access the same object, and one of them wants to modify it, their actions are effectively ordered serial all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction *T* requesting a shared (respectively, exclusive) lock on object *O* as *ST* (*O*) (respectively, *XT* (*O*)), and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown before. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, *T*1 could change *A* from 10 to 20, then *T*2 (which reads the value 20 for *A*) could change *B* from 100 to 200, and then *T*1 would read the value 200 for *B*. If run serially, either *T*1 or *T*2 would execute first, and read the values 10 for *A* and 100 for *B*: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, the above interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as before, *T*1 would obtain an exclusive lock on *A* first and then read and write *A* (next figure). Then, *T*2 would request a lock on *A*. However, this request cannot be granted until *T*1 releases its exclusive lock on *A*, and the DBMS therefore suspends *T*2. *T*1 now proceeds to obtain an exclusive lock on *B*, reads and writes *B*, then finally commits, at which time its locks are released. *T*2's lock request is now granted, and it proceeds.

T1	T2
X(A)	
R(A)	
W(A)	

Introduction to Crash Recovery

The **recovery manager** of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit and durability by making sure that all actions of committed transactions survive **system crashes**, (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

T1	T2
X(A)	

R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
	Commit
X(C)	
R(C)	
W(C)	
Commit	

When a DBMS is restarted after crashes, the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The **transaction manager** of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol. For simplicity of exposition, we make the following assumption:

Atomic Writes: *Writing a page to disk is an atomic action.*

This implies that the system does not crash while a write is in progress and is unrealistic. In practice, disk writes do not have this property, and steps must be taken during restart after a crash to verify that the most recent write to a given page was completed successfully and to deal with the consequences if not.

Stealing Frames and Forcing Pages

With respect to writing objects, two additional questions arise:

1. Can the changes made to an object *O* in the buffer pool by a transaction *T* be written to disk before *T* commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the page containing *O*; of course, this page must have been

unpinned by *T*. If such writes are allowed, we say that a **steal** approach is used. (Informally, the second transaction 'steals' a frame from *T*.)

2. When a transaction commits, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? If so, we say that a **force** approach is used.

From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steal, force approach. If no-steal is used, we don't have to undo the changes of an aborted transaction (because these changes have not been written to disk), and if force is used, we don't have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs. If a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times. With a no-force approach, on the other hand, the in-memory copy of the page would be successively modified and written to disk just once, reflecting the effects of all 20 updates, when the page is eventually replaced in the buffer pool (in accordance with the buffer manager's page replacement policy).

For these reasons, most systems use a steal, no-force approach. Thus, if a frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active (*steal*); in addition, pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits (*no-force*).

Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions in order to enable it to perform its task in the event of a failure. In particular, a **log** of all modifications to the database is saved on **stable storage**, which is guaranteed (with very high probability) to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes. It is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change.

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and to redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log, and must be written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

Concurrency Control

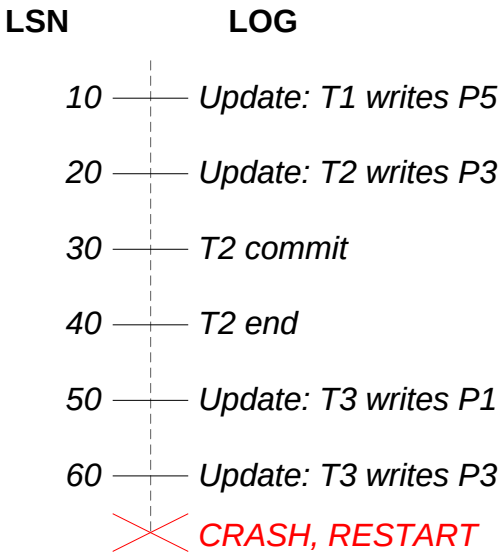
Introduction to Aries

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases:

1. **Analysis:** Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.

2. **Redo:** Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.
3. **Undo:** Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

Consider the simple execution history illustrated in next figure. When the system is restarted, the Analysis phase identifies *T*1 and *T*3 as transactions that were active at the time of the crash, and therefore to be undone; *T*2 as a committed transaction, and all its actions, therefore, to be written to disk; and *P*1, *P*3, and *P*5 as potentially dirty pages. All the updates (including those of *T*1 and *T*3) are reapplied in the order shown during the Redo phase. Finally, the actions of *T*1 and *T*3 are undone in reverse order during the Undo phase; that is, *T*3's write of *P*3 is undone, *T*3's write of *P*1 is undone, and then *T*1's write of *P*5 is undone.



There are three main principles behind the ARIES recovery algorithm:

- **Write-ahead logging:** Any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.
- **Repeating history during Redo:** Upon restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions that were still active at the time of the crash (effectively aborting them).
- **Logging changes during Undo:** Changes made to the database while undoing a transaction are logged in order to ensure that such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES from other recovery algorithms and is the basis for much of its simplicity and flexibility. In particular, ARIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The second and third points are also important in dealing with operations such that redoing and undoing the operation are not exact inverses of each other.

The Log

The log, sometimes called the **trail** or **journal**, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more

copies of the log on different disks (perhaps in different locations), so that the chance of al copies of the log being simultaneously lost is negligibly small.

The most recent portion of the log, called the **log tail**, is kept in main memory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every **log record** is given a unique *id* called the **log sequence number (LSN)**. As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in monotonically increasing order; this property is required for the ARIES recovery algorithm. If the log is a sequential file, in principle growing indefinitely, the LSN can simply be the address of the first byte of the log record.

For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the **pageLSN**.

A log record is written for each of the following actions:

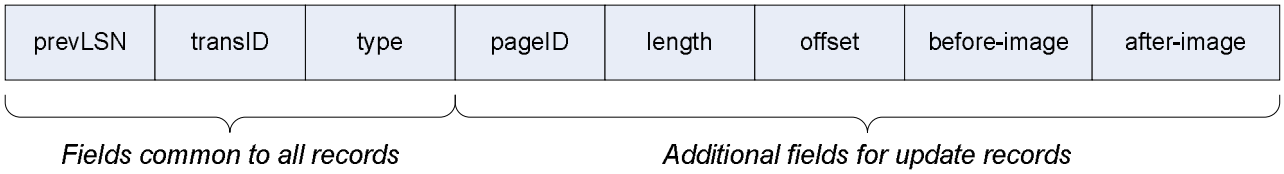
- **Updating a page** : After modifying the page, an *update* type record (described later in this section) is appended to the log tail. The pageLSN of the page is then set to the LSN of the update log record. (The page must be pinned in the buffer pool while these actions are carried out.)
- **Commit**: When a transaction decides to commit, it **force-writes** a *commit* type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record. The transaction is considered to have committed at the instant that its commit log record is written to stable storage. (Some additional steps must be taken, e.g., removing the transaction's entry in the transaction table; these follow the writing of the commit log record.)
- **Abort**: When a transaction is aborted, an *abort* type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction.
- **End**: As noted above, when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, an *end* type log record containing the transaction id is appended to the log.
- **Undoing an update:** When a transaction is rolled back (because the transaction is aborted, or during recovery from a crash), its updates are undone. When the action described by an update log record is undone, a *compensation log record*, or CLR, is written.

Every log record has certain fields: **prevLSN**, **transID**, and **type**. The set of all log records for a given transaction is maintained as a linked list going back in time, using the **prevLSN** field; this list must be updated whenever a log record is added. The transID field is the id of the transaction generating the log record, and the type field obviously indicates the type of the log record.

Additional fields depend on the type of the log record. We have already mentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

Update Log Records

The fields in an **update** log record are illustrated below.



The **pageID** field is the page id of the modified page; the length in bytes and the offset of the change are also included. The **before-image** is the value of the changed bytes before the change; the **after-image** is the value after the change. An update log record that contains both before- and after-images can be used to redo the change and to undo it. In certain contexts, which we will not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A **redo-only update** log record will contain just the after-image; similarly an **undo-only update r** ecord will contain just the before-image.

Compensation Log Records

A **compensation log record (CLR)** is written just before the change recorded in an update log record *U* is undone. (Such an undo can happen during normal system execution when a transaction is aborted or during recovery from a crash.) A compensation log record *C* describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. The compensation log record *C* also contains a field called **undoNextLSN**, which is the LSN of the next log record that is to be undone for the transaction that wrote update record *U*; this field in *C* is set to the value of prevLSN in *U*.

As an example, consider the fourth update log record shown in next figure. If this update is undone, a CLR would be written, and the information in it would include the transID, pageID, length, offset, and before-image fields from the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-image value; thus, this value and the location of the affected bytes constitute the redo information for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in next figure.

Unlike an update log record, a CLR describes an action that will never be *undone*, that is, we never undo an undo action. The reason is simple: an update log record describes a change made by a transaction during normal execution and the transaction may subsequently be aborted, whereas a CLR describes an action taken to rollback a transaction for which the decision to abort has already been made. Thus, the transaction *must* be rolled back, and the undo action described by the CLR is definitely required. This observation is very useful because it bounds the amount of space needed for the log during restart from a crash: The number of CLRs that can be written during Undo is no more than the number of update log records for active transactions at the time of the crash.

It may well happen that a CLR is written to stable storage (following WAL, of course) but that the undo action that it describes is not yet written to disk when the system crashes again. In this case the undo action described in the CLR is reapplied during the Redo phase, just like the action described in update log records.

For these reasons, a CLR contains the information needed to reapply, or redo, the change described but not to reverse it.

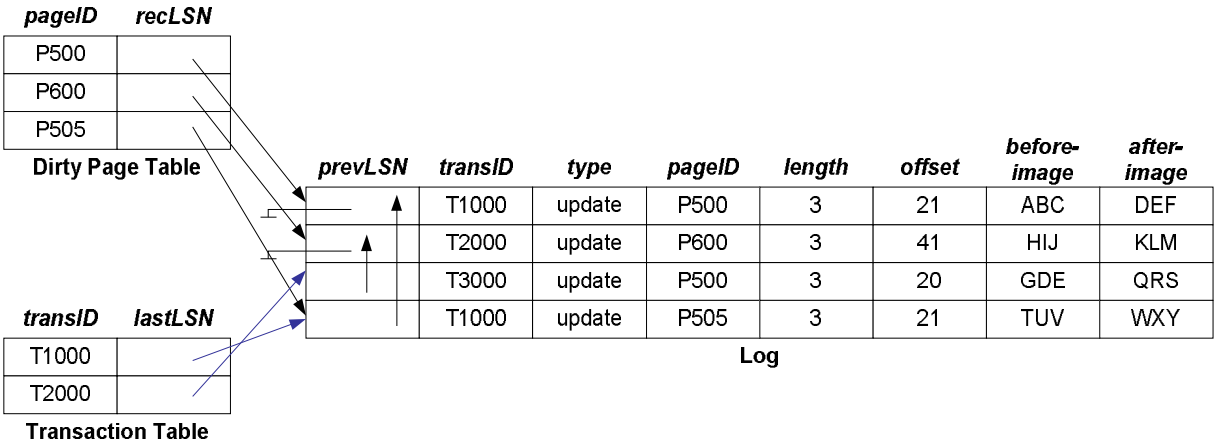
Other Recovery-Related Data Structures

In addition to the log, the following two tables contain important recovery-related information:

- **Transaction table:** This table contains one entry for each active transaction. The entry contains (among other things) the transaction id, the status, and a field called **lastLSN**, which is the LSN of the most recent log record for this transaction. The **status** of a transaction can be that it is in progress, is committed, or is aborted. (In the latter two cases, the transaction will be removed from the table once certain 'clean up' steps are completed.)
- **Dirty page table:** This table contains one entry for each dirty page in the buffer pool, that is, each page with changes that are not yet reflected on disk. The entry contains a field **recLSN**, which is the LSN of the fifirst log record that caused the page to become dirty. Note that this LSN identifies the earliest log record that might have to be redone for this page during restart from a crash.

During normal operation, these are maintained by the transaction manager and the buffer manager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the following simple example. Transaction *T*1000 changes the value of bytes 21 to 23 on page *P*500 from 'ABC' to 'DEF', transaction *T*2000 changes 'HIJ' to 'KLM' on page *P*600, transaction *T*2000 changes bytes 20 through 22 from 'GDE' to 'QRS' on page *P*500, then transaction *T*1000 changes 'TUV' to 'WXY' on page *P*505. The dirty page table, the transaction table, and the log at this instant are shown in next figure. Observe that the log is shown growing from top to bottom; older records are at the top. Although the records for each transaction are linked using the prevLSN field, the log as a whole also has a sequential order that is important for example, *T*2000's change to page *P*500 follows *T*1000's change to page *P*500, and in the event of a crash, these changes must be redone in the same order.



The Write-Ahead Log Protocol

Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage. This is accomplished by forcing all log records up to and including the one with LSN equal to the pageLSN to stable storage before writing the page to disk.

The importance of the WAL protocol cannot be overemphasized. WAL is the fundamental rule that ensures that a record of every change to the database is available while attempting to recover from a crash. If a transaction made a change and committed,

the no-force approach means that some of these changes may not have been written to disk at the time of a subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crashes. Note that the definition of a *committed transaction* is effectively a transaction whose log records, including a commit record, have all been written to stable storage.

When a transaction is committed, the log tail is forced to stable storage, even if a noforce approach is being used. It is worth contrasting this operation with the actions taken under a force approach: If a force approach is used, all the pages modified by the transaction, rather than a portion of the log that includes all its records, must be forced to disk when the transaction commits. The set of all changed pages is typically much larger than the log tail because the size of an update log record is close to (twice) the size of the changed bytes, which is likely to be much smaller than the page size.

Further, the log is maintained as a sequential file, and thus all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is much smaller than the cost of writing all changed pages to disk.

Checkpointing

A **checkpoint** is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash.

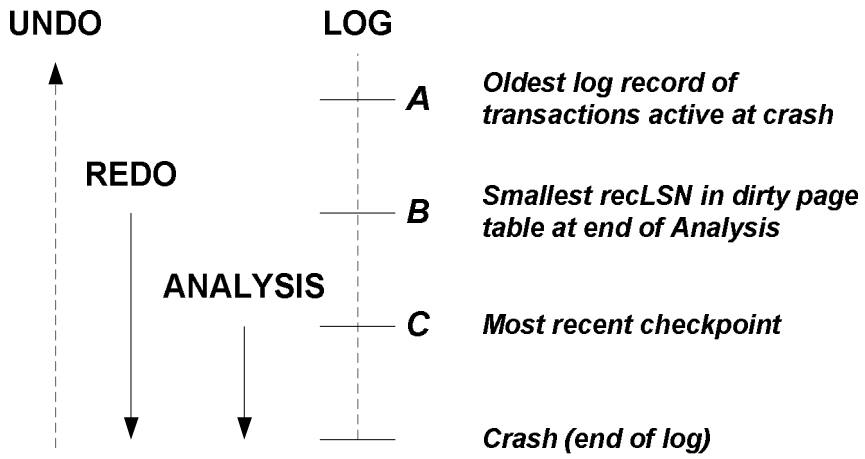
Checkpointing in ARIES has three steps. First, a **begin checkpoint** record is written to indicate when the checkpoint starts. Second, an **end checkpoint** record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log. The third step is carried out after the **end checkpoint** record is written to stable storage: A special **master** record containing the LSN of the *begin checkpoint* log record is written to a known place on stable storage. While the end checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have is that the transaction table and dirty page table are accurate *as of the time of the begin checkpoint record*.

This kind of checkpoint is called a **fuzzy checkpoint** and is inexpensive because it does not require quiescing the system or writing out pages in the buffer pool (unlike some other forms of checkpointing). On the other hand, the effectiveness of this checkpointing technique is limited by the earliest recLSN of pages in the dirty pages table, because during restart we must redo changes starting from the log record whose LSN is equal to this recLSN. Having a background process that periodically writes dirty pages to disk helps to limit this problem.

When the system comes back up after a crash, the restart process begins by locating the most recent checkpoint record. For uniformity, the system always begins normal execution by taking a checkpoint, in which the transaction table and dirty page table are both empty.

Recovering From a System Crash

When the system is restarted after a crash, the recovery manager proceeds in three phases, as shown below.



The Analysis phase begins by examining the most recent begin checkpoint record, whose LSN is denoted as C in previous, and proceeds forward in the log until the last log record. The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash; this set of pages and the starting point for Redo (the smallest recLSN of any dirty page) are determined during Analysis. The Undo phase follows Redo and undoes the changes of all transactions that were active at the time of the crash; again, this set of transactions is identified during the Analysis phase. Notice that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the most recent change first.

Observe that the relative order of the three points A, B, and C in the log may differ from that shown in previous figure. The three phases of restart are described in more detail in the following sections.

Analysis Phase

The **Analysis** phase performs three tasks:

1. It determines the point in the log at which to start the Redo pass.
2. It determines (a conservative superset of the) pages in the buffer pool that were dirty at the time of the crash.
3. It identifies transactions that were active at the time of the crash and must be undone.

Analysis begins by examining the most recent begin checkpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the time of the checkpoint.

Analysis then scans the log in the forward direction until it reaches the end of the log:

- If an end log record for a transaction *T* is encountered, *T* is removed from the transaction table because it is no longer active.
- If a log record other than an end record for a transaction *T* is encountered, an entry for *T* is added to the transaction table if it is not already there. Further, the entry for *T* is modified:
  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a commit record, the status is set to C, otherwise it is set to U (indicating that it is to be undone).
- If a redoable log record affecting page *P* is encountered, and *P* is not in the dirty page table, an entry is inserted into this table with page id *P* and recLSN

equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page *P* that may not have been written to disk.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the time of the crash this is the set of transactions with status U. The dirty page table includes all pages that were dirty at the time of the crash, but may also contain some pages that were written to disk. If an *end write* log record were written at the completion of each write operation, the dirty page table constructed during Analysis could be made more accurate, but in ARIES, the additional cost of writing end write log records is not considered to be worth the gain.

Redo Phase

During the **Redo** phase, ARIES reapplies the updates of *all* transactions, committed or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in the CLRs are also reapplied. This **repeating history** paradigm distinguishes ARIES from other proposed WALbased recovery algorithms and causes the database to be brought to the same state that it was in at the time of the crash.

The Redo phase begins with the log record that has the smallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that may not have been written to disk prior to the crash. Starting from this log record, Redo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action must be redone. The action must be redone unless one of the following conditions holds:

- The affected page is not in the dirty page table, or
- The affected page is in the dirty page table, but the recLSN for the entry is *greater than* the LSN of the log record being checked, or
- The pageLSN (stored on the page, which must be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously means that all changes to this page have been written to disk. Because the recLSN is the first update to this page that may not have been written to disk, the second condition means that the update being checked was indeed propagated to disk. The third condition, which is checked last because it requires us to retrieve the page, also ensures that the update being checked was written to disk, because either this update or a later update to the page was written. (Recall our assumption that a write to a page is atomic; this assumption is important here!)

If the logged action must be redone:

1. The logged action is reapplied.
2. The pageLSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

Undo Phase

The Undo phase, unlike the other two phases, scans backward from the end of the log. The goal of this phase is to undo the actions of all transactions that were active at the time of the crash, that is, to effectively abort them. This set of transactions is identified in the transaction table constructed by the Analysis phase.

The Undo Algorithm

Undo begins with the transaction table constructed by the Analysis phase, which identifies all transactions that were active at the time of the crash, and includes the LSN of the most recent log record (the lastLSN field) for each such transaction. Such transactions

are called **loser transactions**. All actions of losers must be undone, and further, these actions must be undone in the reverse of the order in which they appear in the log.

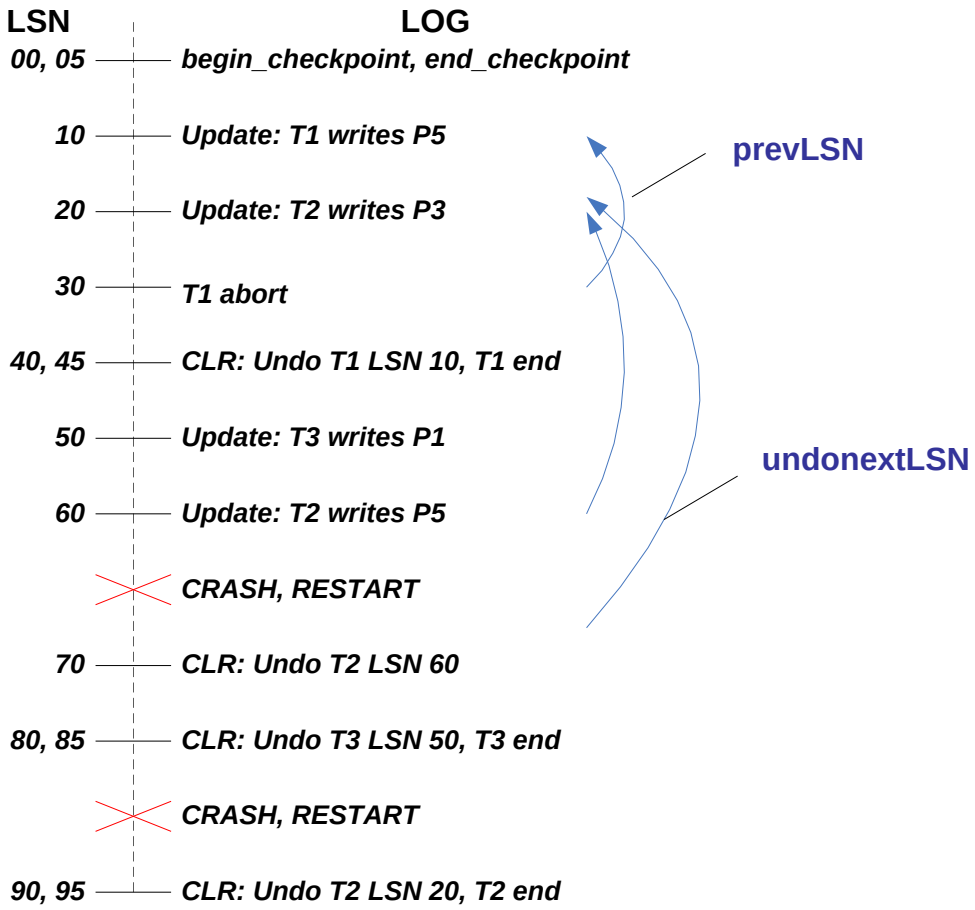
Consider the set of lastLSN values for all loser transactions. Let us call this set **ToUndo**. Undo repeatedly chooses the largest (i.e., most recent) LSN value in this set and processes it, until ToUndo is empty. To process a log record:

1. If it is a CLR, and the undoNextLSN value is not *null*, the undoNextLSN value is added to the set ToUndo; if the undoNextLSN is *null*, an end record is written for the transaction because it is completely undone, and the CLR is discarded.
2. If it is an update record, a CLR is written and the corresponding action is undone, and the prevLSN value in the update log record is added to the set ToUndo.

When the set ToUndo is empty, the Undo phase is complete. Restart is now complete, and the system can proceed with normal operations.

Aborting a Transaction

Aborting a transaction is just a special case of the Undo phase of Restart in which a single transaction, rather than a set of transactions, is undone. The example in following figure, discussed next, illustrates this point.



Crashes during Restart

It is important to understand how the Undo algorithm presented in previous section handles repeated system crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we will discuss Undo in

the presence of system crashes using an execution history, shown in last figure, that abstracts away unnecessary detail. This example illustrates how aborting a transaction is a special case of Undo and how the use of CLRs ensures that the Undo action for an update log record is not applied twice.

The log shows the order in which the DBMS executed various actions; notice that the LSNs are in ascending order, and that each log record for a transaction has a prevLSN field that points to the previous log record for that transaction. We have not shown *null* prevLSNs, that is, some special value used in the prevLSN field of the first log record for a transaction to indicate that there is no previous log record. We have also compacted the figure by occasionally displaying two log records (separated by a comma) on a single line.

Log record (with LSN) 30 indicates that *T*1 aborts. All actions of this transaction should be undone in reverse order, and the only action of *T*1, described by the update log record 10, is indeed undone as indicated by CLR 40.

After the first crash, Analysis identifies *P*1 (with recLSN 50), *P*3 (with recLSN 20), and *P*5 (with recLSN 10) as dirty pages. Log record 45 shows that *T*1 is a completed transaction; thus, the transaction table identifies *T*2 (with lastLSN 60) and *T*3 (with lastLSN 50) as active at the time of the crash. The Redo phase begins with log record 10, which is the minimum recLSN in the dirty page table, and reapplies all actions (for the update and CLR records), as per the Redo algorithm.

The ToUndo set consists of LSNs 60, for *T*2, and 50, for *T*3. The Undo phase now begins by processing the log record with LSN 60 because 60 is the largest LSN in the ToUndo set. The update is undone, and a CLR (with LSN 70) is written to the log. This CLR has undoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for *T*2. Now the largest remaining LSN in the ToUndo set is 50. The write corresponding to log record 50 is now undone, and a CLR describing the change is written. This CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction *T*3. Thus *T*3 is completely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the system crashes a second time; however, the changes described by these records may not have been written to disk.

When the system is restarted after the second crash, Analysis determines that the only active transaction at the time of the crash was *T*2; in addition, the dirty page table is identical to what it was during the previous restart. Log records 10 through 85 are processed again during Redo. (If some of the changes made during the previous Redo were written to disk, the pageLSNs on the affected pages are used to detect this situation and avoid writing these pages again.) The Undo phase considers the only LSN in the ToUndo set, 70, and processes it by adding the undoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed by undoing *T*2's write of page *P*3, and a CLR is written (LSN 90). Because 20 is the first of *T*2's log records and therefore, the last of its records to be undone the undoNextLSN field in this CLR is *null*, an end record is written for *T*2, and the ToUndo set is now empty.

Recovery is now complete, and normal execution can resume with the writing of a checkpoint record.

This example illustrated repeated crashes during the Undo phase. For completeness, let us consider what happens if the system crashes while Restart is in the Analysis or Redo phases. If a crash occurs during the Analysis phase, all the work done in this phase is lost, and on restart the Analysis phase starts afresh with the same information as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that some of the changes made during Redo may have been written to disk prior to the crash. Restart starts again with the Analysis phase and then the Redo phase, and some update log records that were redone the first time around will not be

redone a second time because the pageLSN will now be equal to the update record's LSN (although the pages will have to fetched again to detect this).

We can take checkpoints during Restart to minimize repeated work in the event of a crash, but we will not discuss this point.

Media Recovery

Media recovery is based on periodically making a copy of the database. Because copying a large database object such as a file can take a long time, and the DBMS must be allowed to continue with its operations in the meantime, creating a copy is handled in a manner similar to taking a fuzzy checkpoint. When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of committed transactions and undo the changes of uncommitted transactions (as of the time of the media recovery operation).

The begin_checkpoint LSN of the most recent complete checkpoint is recorded along with the copy of the database object in order to minimize the work in reapplying changes of committed transactions. Let us compare the smallest recLSN of a dirty page in the corresponding end checkpoint record with the LSN of the begin checkpoint record and call the smaller of these two LSNs *I*. We observe that the actions recorded in all log records with LSNs less than *I* must be reflected in the copy. Thus, only log records with LSNs greater than *I* need to be reapplied to the copy.

Finally, the updates of transactions that are incomplete at the time of media recovery or that were aborted after the fuzzy copy was completed need to be undone to ensure that the page reflects only the actions of committed transactions. The set of such transactions can be identified as in the Analysis pass, and we omit the details.