DATABASE MANAGEMENT SYSTEM

(DBMS)

(R-13 Autonomous)

Department of Computer Science & Engineering

Malla Reddy College of Engineering & Technology

(Accredited by NBA with NAAC-A Grade, UGC-Autonomous, ISO Certified Institution)

Maisammaguda, Near Kompally, Medchal Road, Sec'bad-500 100.

SYLLABUS

(R15A0509) DATABASE MANAGEMENT SYSTEMS

Objectives:

- To Understand the basic concepts and the applications of database systems
- To Master the basics of SQL and construct queries using SQL
- To understand the relational database design principles
- To become familiar with the basic issues of transaction processing and concurrency control
- To become familiar with database storage structures and access techniques

UNIT I:

Data base System Applications, Purpose of Database Systems, View of Data – Data Abstraction – Instances and Schemas – data Models – the ER Model – Relational Model – Other Models – Database Languages – DDL – DML – database Access for applications Programs – data base Users and Administrator – Transaction Management – data base Architecture – Storage Manager – the Query Processor

Data base design and ER diagrams – ER Model - Entities, Attributes and Entity sets – Relationships and Relationship sets – ER Design Issues – Concept Design – Conceptual Design for University Enterprise.

Introduction to the Relational Model – Structure – Database Schema, Keys – Schema Diagrams

UNIT II:

Relational Query Languages, Relational Operations.

Relational Algebra – Selection and projection set operations – renaming – Joins – Division – Examples of Algebra overviews – Relational calculus – Tuple relational Calculus – Domain relational calculus.

Overview of the SQL Query Language – Basic Structure of SQL Queries, Set Operations, Aggregate Functions – GROUPBY – HAVING, Nested Sub queries, Views, Triggers.

UNIT III:

Normalization – Introduction, Non loss decomposition and functional dependencies, First, Second, and third normal forms – dependency preservation, Boyee/Codd normal form.

Higher Normal Forms - Introduction, Multi-valued dependencies and Fourth normal form, Join dependencies and Fifth normal form

UNIT IV:

Transaction Concept- Transaction State- Implementation of Atomicity and Durability – Concurrent – Executions – Serializability- Recoverability – Implementation of Isolation – Testing for serializability- Lock –Based Protocols – Timestamp Based Protocols- Validation- Based Protocols – Multiple Granularity.

Recovery and Atomicity – Log – Based Recovery – Recovery with Concurrent Transactions – Buffer Management – Failure with loss of nonvolatile storage-Advance Recovery systems- Remote Backup systems.

UNIT V:

File organization:– File organization – various kinds of indexes. Query Processing – Measures of query cost - Selection operation – Projection operation, - Join operation – set operation and aggregate operation – Relational Query Optimization – Transacting SQL queries – Estimating the cost – Equivalence Rules.

TEXT BOOKS:

1. Data base System Concepts, Silberschatz, Korth, McGraw hill, Sixth Edition.(All UNITS except III th)
2. Data base Management Systems, Raghurama Krishnan, Johannes Gehrke, TATA McGrawHill 3rd Edition.

REFERENCE BOOKS:

1. Fundamentals of Database Systems, Elmasri Navathe Pearson Education.
2. An Introduction to Database systems, C.J. Date, A.Kannan, S.Swami Nadhan, Pearson, Eight Edition for UNIT III.

URLs:

Outcomes:

- Demonstrate the basic elements of a relational database management system
- Ability to identify the data models for relevant problems
- Ability to design entity relationship and convert entity relationship diagrams into RDBMS and formulate SQL queries on the respect data
- Apply normalization for the development of application software's

UNIT-1

Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

What is a Database?

To find out what database is, we have to start from data, which is the basic building block of any DBMS.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).
Record: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

Roll	Name	Age
1	ABC	19

Table or **Relation**: Collection of related records.

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

The columns of this relation are called **Fields**, **Attributes** or **Domains**. The rows are called **Tuples** or **Records**.

Database: Collection of related relations. Consider the following collection of tables:

T1

Roll	Name	Age
1	ABC	19
2	DEF	22
3	XYZ	28

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3

Roll	Year
1	I
2	II
3	I

T4

Year	Hostel
I	H1
II	H2

We now have a collection of 4 tables. They can be called a "related collection" because we can clearly find out that there are some common attributes existing in a selected pair of tables. Because of these common attributes we may combine the data of two or more tables together to find out the complete details of a student. Questions like "Which hostel does the youngest student live in?" can be answered now, although

Age and ***Hostel*** attributes are in different tables.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

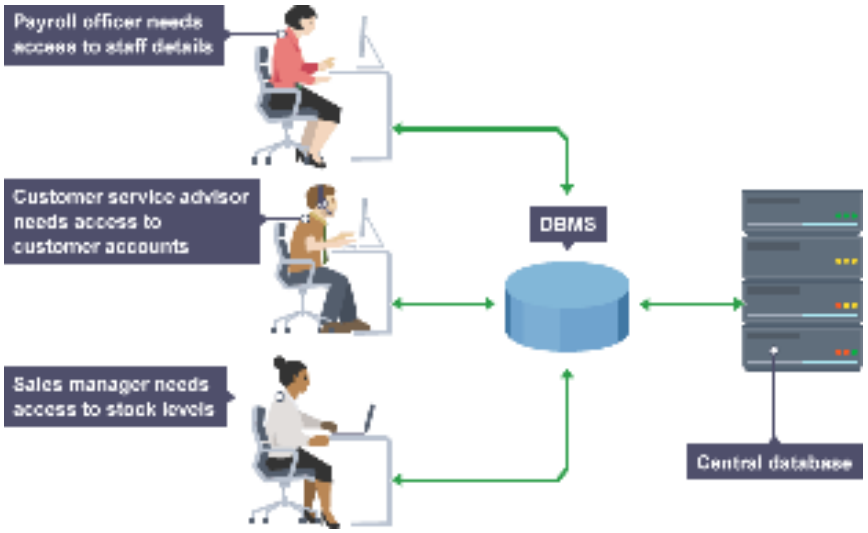


Figure 1.1: Empolyees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

In a database, data is organized strictly in row and column format. The rows are called **Tuple** or **Record**. The data items within one row may belong to different data types. On the other hand, the columns are often called **Domain** or **Attribute**. All the data items within a single attribute are of the same data type.

What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*. By **data,** we mean known facts that can be recorded and that have implicit meaning.

The management system is important because without the existence of some kind of rules and regulations it is not possible to maintain the database. We have to select the particular attributes which should be included in a particular table; the common attributes to create relationship between two tables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must be resolved by having some kind of rules to follow in order to maintain the integrity of the database.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This

chapter briefly introduces the principles of database systems.

Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow he users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

Databases touch all aspects of our lives. Some of the major areas of application are as follows:

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources

Enterprise Information

- *Sales*: For customer, product, and purchase information.
- *Accounting*: For payments, receipts, account balances, assets and other accounting information.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items inwarehouses and stores, and orders for items.

Online retailers: For sales data noted above plus online order tracking,generation of recommendation lists, and maintenance of online product evaluations.

Banking and Finance

- *Banking*: For customer information, accounts, loans, and banking transactions.
- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

- *Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

- ✓ Add new students, instructors, and courses
- ✓ Register students for courses and generate class rosters
- ✓ Assign grades to students, compute grade point averages (GPA), and generate transcripts

System programmers wrote these application programs to meet the needs of the university.
New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major (say, computer science).As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, etc. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems. Keeping organizational information in a file-processing system has a number of major disadvantages:

Data redundancy and inconsistency. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

Difficulty in accessing data. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* students.

The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure.

Consider a program to transfer $500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department *A* but was not credited to the balance of department *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur.

That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or $9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision.
But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

Security problems. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

Advantages of DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

Improved Data Sharing : DBMS allows a user to share the data in any number of application programs.

Data Integrity : Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can can enforce an integrity that it must accept the customer only from Noida and Meerut city.

Security : Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

Data Consistency : By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

Efficient Data Access : In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

Enforcements of Standards : With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

Data Independence : Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

Reduced Application Development and Maintenance Time : DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.
2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.
3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.
4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

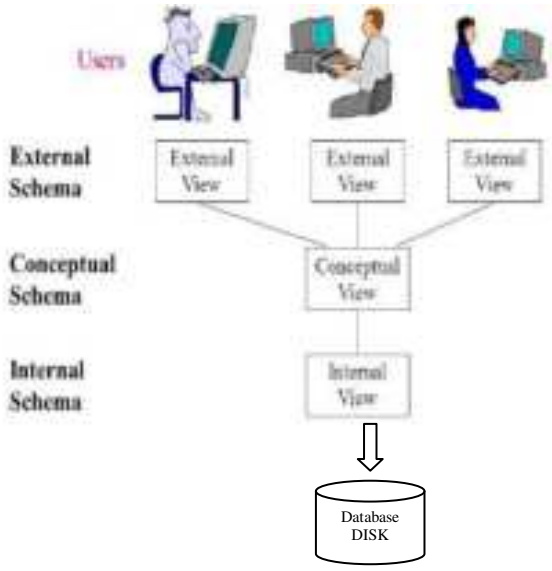


Figure 1.2 : Levels of Abstraction in a DBMS

• **Physical level (or Internal View / Schema)**: The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level (or Conceptual View / Schema)**: The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level (or External View / Schema):** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database. Figure 1.2 shows the relationship among the three levels of abstraction.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. For example, we may describe a record as follows:

```
type instructor = record
                    ID : char (5);
                    name : char (20);
                    dept name : char (20);
                    salary : numeric (8,2);
                end;
```

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

- *department*, with fields *dept_name*, *building*, and *budget*
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition
to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all. The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program.

Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema. Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, which describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

• **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**. The relational model is an example of a record-based model.
Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Entity-Relationship Model. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.
An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

Object-Based Data Model. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.

Semi-structured Data Model. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **Extensible Markup Language (XML)** is widely used to represent semi-structured data.

Historically, the **network data model** and the **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are used little now, except in old database code that is still in service in some places.

Database Languages
A database system provides a **data-definition language** to specify the database
schema and a **data-manipulation language** to express database queries and updates. In practice, the data-definition and data-manipulation languages are not
two separate languages; instead they simply form parts of a single database language,
such as the widely used SQL language.

Data-Manipulation Language

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

There are basically two types:
• **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
• **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing

data. A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

Data-Definition Language (DDL)
We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**). The DDL is also used to specify additional properties of the data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.
The data values stored in the database must satisfy certain **consistency constraints**.
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimal overhead.

• **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

• **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

• **Assertions**. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, “Every department must have at least five courses offered every semester” must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

• **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.
The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.
Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date. A passive data dictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.
The data dictionary's main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone data dictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization's data, whether or not they are computerized. Whatever the data dictionary's format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.
Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.
Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user. The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Database Architecture:

We are now in a position to provide a single picture (Figure 1.3) of the various components of a database system and the connections among them.

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

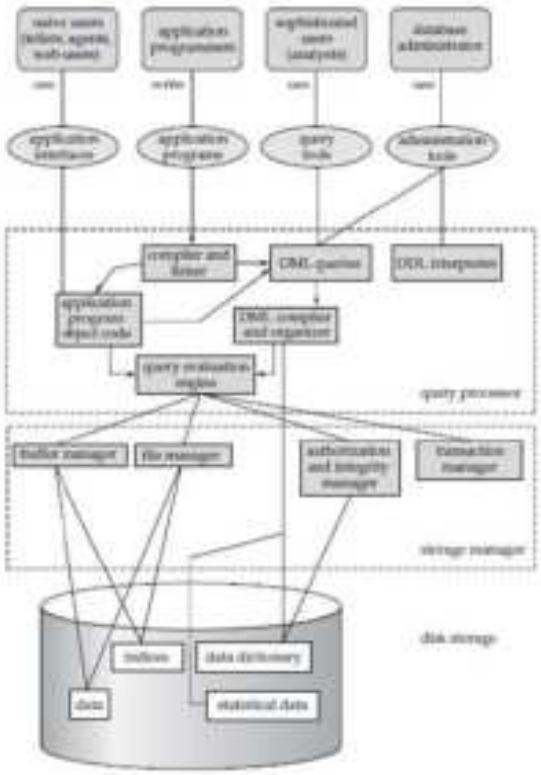


Figure 1.3: Database System Architecture

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components. The storage manager is important because databases typically require a large amount of storage space. The query processor is important because it helps the database system simplify and facilitate access to data.

It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Database applications are usually partitioned into two or three parts, as in Figure 1.4. In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface.

The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the WorldWideWeb.

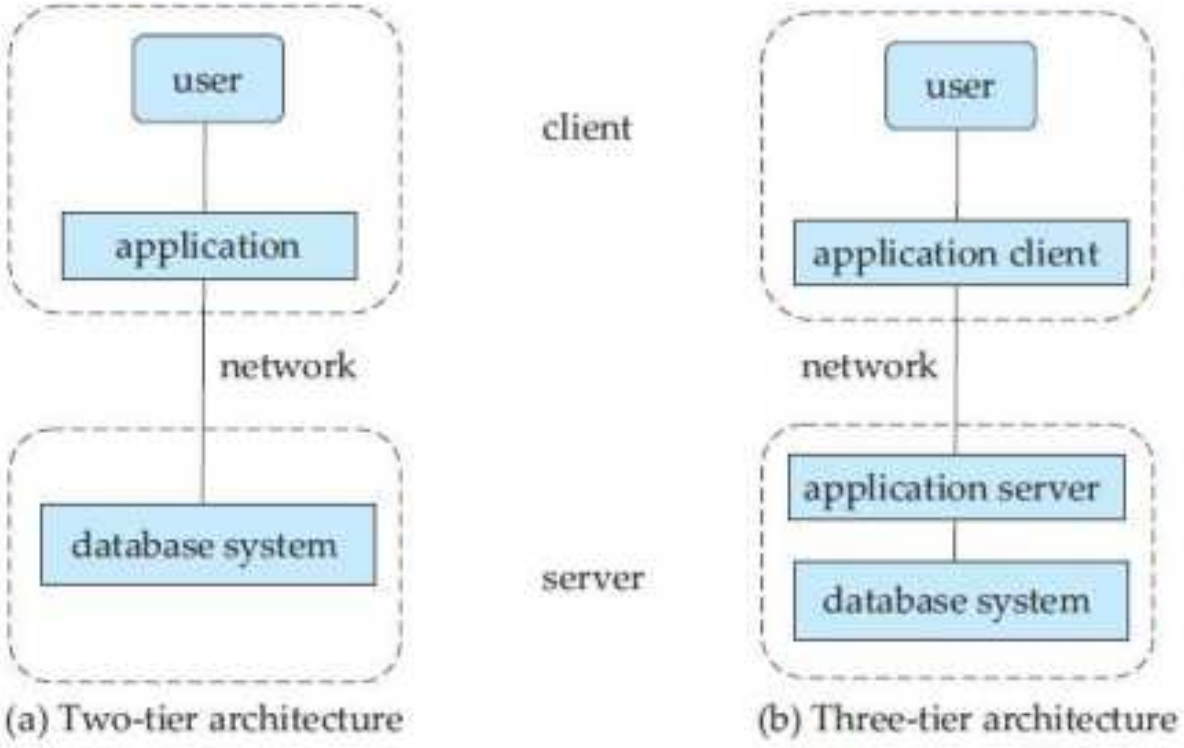


Figure 1.4: Two-tier and three-tier architectures.

Query Processor:

The query processor components include

· **DDL interpreter,** which interprets DDL statements and records the definitions in the data dictionary.

· **DML compiler,** which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

Storage Manager:

A *storage manager* is a program module that provides the interface between the lowlevel data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

· **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
· **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
· **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
· **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

Transaction Manager:

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. **Transaction - manager** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

Conceptual Database Design - Entity Relationship(ER) Modeling:

Database Design Techniques
1. ER Modeling (Top down Approach)
2. Normalization (Bottom Up approach)

What is ER Modeling?

A graphical technique for understanding and organizing the data independent of the actual database implementation

We need to be familiar with the following terms to go further.

Entity

Any thing that has an independent existence and about which we collect data. It is also known as entity type.

In ER modeling, notation for entity is given below.

Entity instance

Entity instance is a particular member of the entity type.

Example for entity instance : A particular employee

Regular Entity

An entity which has its own key attribute is a regular entity.

Example for regular entity : Employee.

Weak entity

An entity which depends on other entity for its existence and doesn't have any key attribute of its own is a weak

entity.

Example for a weak entity : In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.

In ER modeling, notation for weak entity is given below.

Attributes

Properties/characteristics which describe entities are called attributes.

In ER modeling, notation for attribute is given below.

Domain of Attributes

The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}. Hence this set can be termed as the domain of the attribute day.

Key attribute

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.g the employee_id of an employee, pan_card_number of a person etc.If the key attribute consists of two or more attributes in combination, it is called a composite key.

In ER modeling, notation for key attribute is given below.

Simple attribute

If an attribute cannot be divided into simpler components, it is a simple attribute.

Example for simple attribute : employee_id of an employee.

Composite attribute

If an attribute can be split into components, it is called a composite attribute.

Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

Single valued Attributes

If an attribute can take only a single value for each entity instance, it is a single valued attribute.

example for single valued attribute : age of a student. It can take only one value for a particular student.

Multi-valued Attributes

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi-valued

example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

In ER modeling, notation for multi-valued attribute is given below.

Stored Attribute

An attribute which need to be stored permanently is a stored attribute

Example for stored attribute : name of a student

Derived Attribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Example for derived attribute : age of employee which can be calculated from date of birth and current date.

In ER modeling, notation for derived attribute is given below.

Relationships

Associations between entities are called relationships

Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.

In ER modeling, notation for relationship is given below.

However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as given below

Degree of a Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively.

Example for unary relationship : An employee ia a manager of another employee

Example for binary relationship : An employee works-for department.

Example for ternary relationship : customer purchase item from a shop keeper

Cardinality of a Relationship

Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivities as given below.

1. One to one (1:1) relationship

2. One to many (1:N) relationship

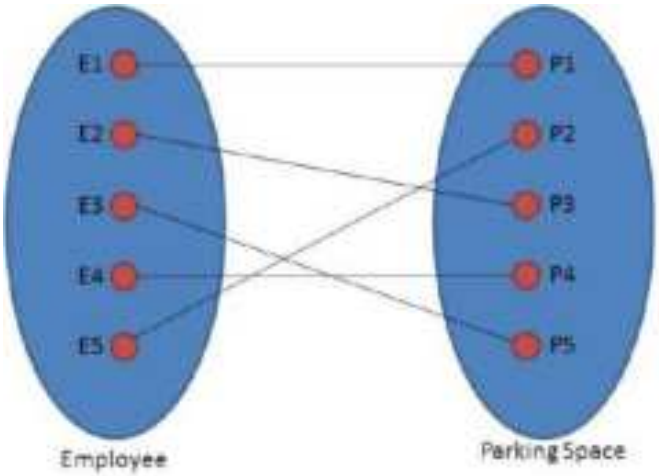
3. Many to one (M:1) relationship

4. Many to many (M:N) relationship

The minimum and maximum values of this connectivity is called the cardinality of the relationship

Example for Cardinality – One-to-One (1:1)

Employee is assigned with a parking space.

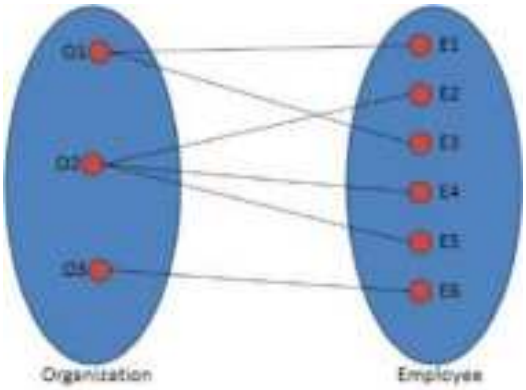


One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)

In ER modeling, this can be mentioned using notations as given below

Example for Cardinality – One-to-Many (1:N)

Organization has employees

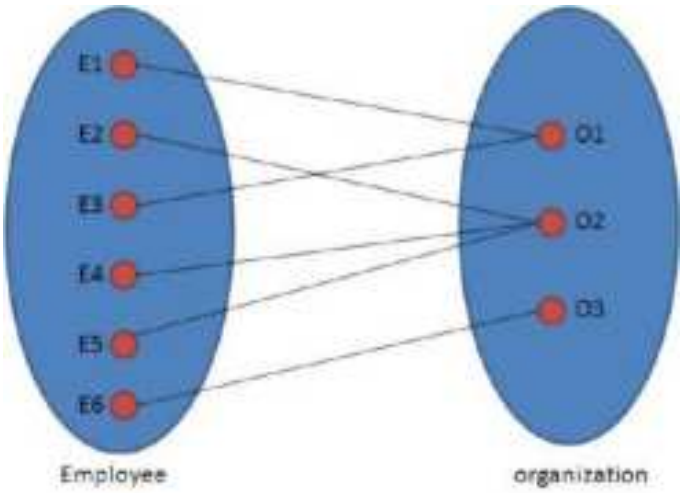


One organization can have many employees , but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)

In ER modeling, this can be mentioned using notations as given below

Example for Cardinality – Many-to-One (M :1)

It is the reverse of the One to Many relationship. employee works in organization

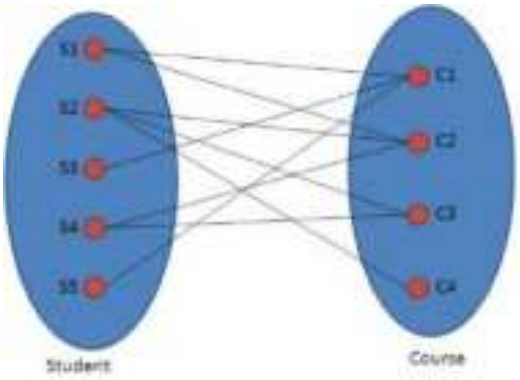


One employee works in only one organization But one organization can have many employees. Hence it is a M:1 relationship and cardinality is Many-to-One (M :1)

In ER modeling, this can be mentioned using notations as given below.

Cardinality – Many-to-Many (M:N)

Students enrolls for courses



One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to-Many (M:N)

In ER modeling, this can be mentioned using notations as given below

Relationship Participation

1. Total

In total participation, every entity instance will be connected through the relationship to another instance of the other participating entity types

2. Partial

Example for relationship participation

Consider the relationship - Employee is head of the department.

Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship.

However each department will be headed by some employee. So department entity's participation is total in the said relationship.

Advantages and Disadvantages of ER Modeling (Merits and Demerits of ER Modeling)

Advantages

1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.

2. Intuitive and helps in Physical Database creation.

3. Can be generalized and specialized based on needs.

4. Can help in database design.

5. Gives a higher level description of the system.

Disadvantages

1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.

2. Sometime diagrams may lead to misinterpretations

Relational Model

The relational model is today the primary data model for commercial data processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this, we first study the fundamentals of the relational model. A substantial theory exists for relational databases.

Structure of Relational Databases:

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure:1.5, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept name*, and *salary*. Similarly, the *course* table of Figure 1.6 stores information about courses, consisting of a *course id*, *title*, *dept name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

Figure 1.7 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course id* and *prereq id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*,*dept name*, and *salary* values.

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 1.5: The *instructor* relation (2.1)

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure: 1.6: The *course* relation (2.2)

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure: 1.7: The *prereq* relation. (2.3)

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 1.5, we can see that the relation *instructor* has four attributes:

***ID*, *name*, *dept name*, and *salary*.**

We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of *instructor* shown in Figure 1.5 has 12 tuples, corresponding to 12 instructors.

In this topic, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. They do not include all the data an actual university database would contain, in order to simplify our presentation.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 1.5, or are unsorted, as in Figure 1.8, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute. For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units.

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Figure: 1.8: Unsorted display of the *instructor* relation. (2-4)

For example, suppose the table *instructor* had an attribute *phone number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone number* would have an atomic domain.

The **null** value is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given

instant in time. The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 1.9: The *department* relation.(2-5)

similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor* schema," or "an instance of the *instructor* relation." However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of Figure 1.9. The schema for that relation is

***department* (*dept name*, *building*, *budget*)**

Note that the attribute *dept name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept name* of all the departments housed in Watson. Then, for each such department, we look in the *instructor* relation to find the information about the instructor associated with the corresponding *dept name*.

Let us continue with our university database example. Each course in a university may be offered multiple times, across different semesters, or even within a semester.We need a relation to describe each individual offering, or section, of the class. The schema is

***section* (*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)**

Figure 1.10 shows a sample instance of the *section* relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is

***teaches* (*ID*, *course id*, *sec id*, *semester*, *year*)**

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 1.10: The *section* relation.(2-6)

Figure 1.11 shows a sample instance of the *teaches* relation. As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*,we use the following relations in this text:

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure: 1.11: The *teaches* relation.(2-7)

- *student* (*ID*, *name*, *dept name*, *tot cred*)
- *advisor* (*s id*, *i id*)
- *takes* (*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
- *classroom* (*building*, *room number*, *capacity*)
- *time slot* (*time slot id*, *day*, *start time*, *end time*)

Keys

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from

another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name. Formally, let *R* denote the set of attributes in the schema of relation *r*. If we say that a subset *K* of *R* is a *superkey* for *r* , we are restricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if t_1 and t_2 are in *r* and $t_1 \neq t_2$, then $t_1.K \neq t_2.K$.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If *K* is a superkey, then so is any superset of *K*. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name*, *dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled.

Primary keys must be chosen with care. As we noted, the name of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social-security numbers, international enterprises must generate their own unique identifiers.

An alternative is to use some unique combination of other attributes as a key. The primary key should be chosen such that its attribute values are never, or very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined. A relation, say r_1, may include among its attributes the primary key of another relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.

The relation r_1 is also called the **referencing relation** of the foreign key dependency, and r_2 is called the **referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from *instructor*, referencing *department*, since *dept name* is the primary key of *department*. In any database instance, given any tuple, say t_a, from the *instructor* relation, there must be some tuple, say t_b, in the *department* relation such that the value of the *dept name* attribute of t_a is the same as the value of the primary key, *dept name*, of t_b.

Now consider the *section* and *teaches* relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by **schema diagrams**. Figure 1.12 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

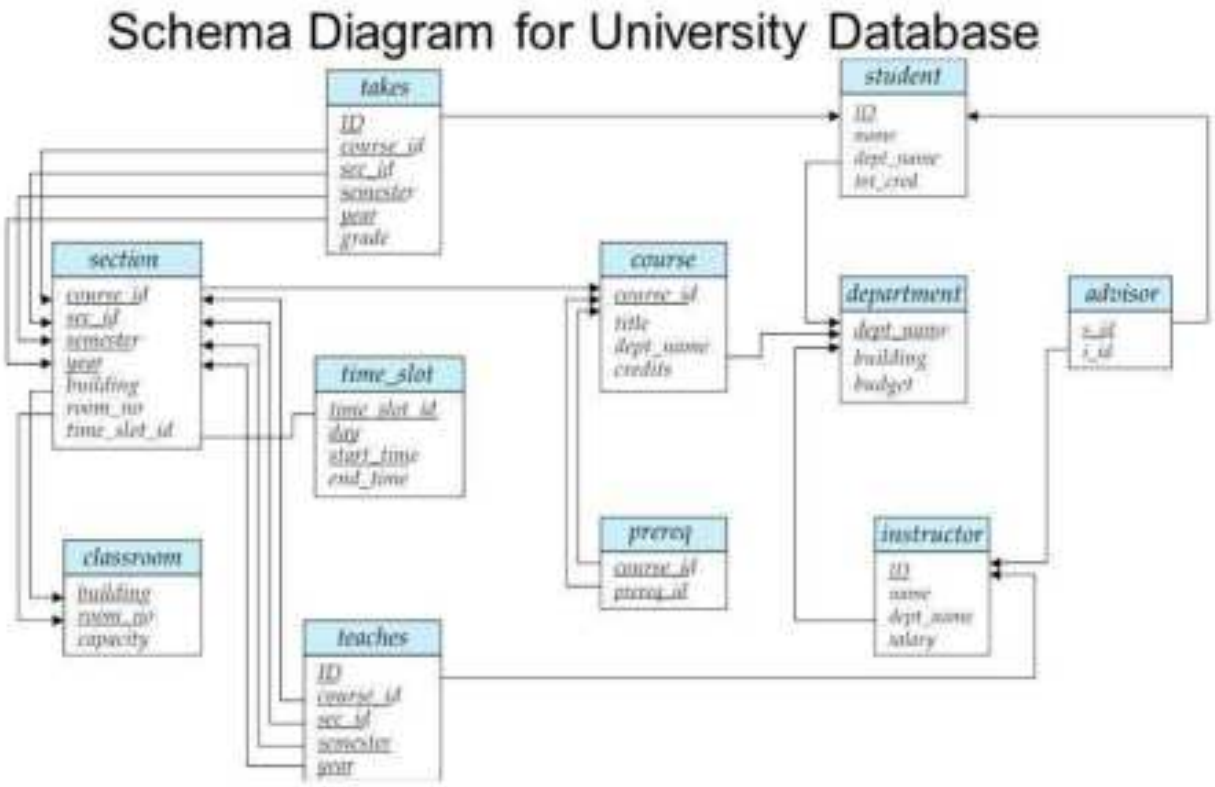


Figure 1.12 : Schema diagram for the university database.

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram.

UNIT-2

Relational Algebra and Calculus

PRELIMINARIES

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to 'inherit' field names, for convenience.

We present a number of sample queries using the following schema:

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

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

In several examples illustrating the relational algebra operators, we will use the in-stances $S1$ and $S2$ (of Sailors) and $R1$ (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively,

RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to compose operators to form a complex query—a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression,

$$\sigma_{rating>8}(S2)$$

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

sname	*rating*
yuppy	*9*
Lubber	*8*
Guppy	*5*
Rusty	*10*

Figure 4.5 $\pi_{sname,rating}(S2)$

sid	*sname*	*rating*	*age*
28	*Yuppy*	*9*	*35.0*
58	*Rusty*	*10*	*35.0*

Figure 4.4 $\sigma_{rating>8}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition*. In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, =, >=, or >. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance

The projection operator π allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using π. The expression $\pi_{sname,rating}(S2)$

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

We can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

Set Operations

The following standard operations on sets are also available in relational algebra: *union* ($\cup$), *intersection* ($\cap$), *set-difference* ($-$), and *cross-product* ($\times$).

- Union: $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance R or relation instance S (or both). R and S must be *union-compatible*, and the schema of the result is defined to be identical to the schema of R.

- Intersection: $R \cap S$ returns a relation instance containing all tuples that occur in *both* R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- Set-difference: $R - S$ returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

- Cross-product: $R \times S$ returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of $R \times S$ contains one tuple $\langle r, s \rangle$ (the concatenation of tuples r and s) for each pair of tuples $r \in R$, $s \in S$.

The cross-product opertion is sometimes called Cartesian product.

We now illustrate these definitions through several examples. The union of $S1$ and $S2$ is shown in Figure 4.8. Fields are listed in order; field names are also inherited from $S1$. $S2$ has the same field names, of course, since it is also an instance of Sailors.In general, fields of $S2$ may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both $S1$ and $S2$ appear only once in $S1 \cup S2$. Also, $S1 \cup R1$ is not a valid operation because the two relations are not union-compatible. The intersection of $S1$ and $S2$ is shown in Figure 4.9, and the set-difference $S1 - S2$ is shown in Figure 4.10.

sid	*sname*	*rating*	*age*
22	*Dustin*	*7*	*45.0*
31	*Lubber*	*8*	*55.5*
58	*Rusty*	*10*	*35.0*
28	*Yuppy*	*9*	*35.0*
44	*Guppy*	*5*	*35.0*

Figure 4.8 $S1 \cup S2$

sid	*sname*	*rating*	*age*
31	*Lubbe*	*8*	*55.5*
58	*Rusty*	*10*	*35.0*

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11 The fields in $S1 \times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11 *sid* is listed in parentheses to
emphasize that it is not an inherited field name; only the corresponding domain is inherited.

(*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*

Figure 4.11 $S1 \times R1$

Renaming

We introduce a renaming operator ρ for this purpose. The expression $\rho(R(F), E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the *renaming list* F.

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer,*day:* dates).

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the $\cap$ operator is redundant; $R \cap S$ can be defined as $R - (R - S)$.) We will consider these additional operators,and their definition in terms of the basic operators, in the next two subsections.

Joins

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products.

joins have received a lot of attention, and there are several variants of the join operation.

Condition Joins

The most general version of the join operation accepts a *join condition* c and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

$$R \lhd\rhd_c S = \sigma_c(R \times S)$$

Thus $\lhd\rhd$ is defined to be a cross-product followed by a selection. Note that the condition c can (and typically *does*) refer to attributes of both R and S. The reference to an

attribute of a relation, say R, can be by position (of the form $R.i$) or by name (of the form $R.name$).As an example, the result of $S1 \lhd\rhd_{S1.sid<R1.sid} R1$ is shown in Figure 4.12. Because *sid* appears in both $S1$ and $R1$, the corresponding fields in the result of the cross-product $S1 \times R1$ (and therefore in the result of $S1 \lhd\rhd_{S1.sid<R1.sid} R1$) are unnamed. Domains are inherited from the corresponding fields of $S1$ and $R1$.

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

Figure 4.12 $S1 \lhd\rhd_{S1.sid<R1.sid} R1$

Equijoin

A common special case of the join operation $R \lhd\rhd S$ is when the *join condition* consists solely of equalities (connected by $\wedge$) of the form $R.name1 = S.name2$, that is, equalities between two fields in R and S. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which $S.name2$ is dropped. The join operation with this refinement is called equijoin.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \lhd\rhd_{R.sid=S.sid} R1$ in Figure 4.13. Notice that only one field called *sid* appears in the result.

sid	*sname*	*rating*	*age*	*bid*	*day*
22	*Dustin*	*7*	*45.0*	*101*	*10/10/96*
58	*Rusty*	*10*	*35.0*	*103*	*11/12/96*

Figure 4.13 $S1 \lhd\rhd_{R.sid=S.sid} R1$

Natural Join

A further special case of the join operation $R \lhd\rhd S$ is an equijoin in which equalities are specified on *all* fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a *natural join*, and it has the nice property that the result is guaranteed not to have two fields with the same name.

The equijoin expression $S1 \lhd\rhd_{R.sid=S.sid} R1$ is actually a natural join and can simply be denoted as $S1 \lhd\rhd R1$, since the only common field is *sid*. If the two relations have no attributes in common, $S1 \lhd\rhd R1$ is simply the cross-product.

Division

The division operator is useful for expressing certain kinds of queries, for example: "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise. However,

the division operator does not have the same importance as the other operators—it is not needed as often, and database systems do not try to exploit the semantics of division by implementing it as a distinct operator (as, for example, is done with the join operator).

We discuss division through an example. Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the *division* operation A/B as the set of all x values (in the form of unary tuples) such that for *every* y value in (a tuple of) B, there is a tuple $\langle x,y \rangle$ in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, the x value is in the result of A/B.

An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that $Q * B \leq A$. For relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.

Division is illustrated in Figure 4.14. It helps to think of A as a relation listing the parts supplied by suppliers, and of the B relations as listing parts. A/Bi computes suppliers who supply *all* parts listed in relation instance Bi.

Expressing A/B in terms of the basic algebra operators is an interesting exercise, and the reader should try to do this before reading further. The basic idea is to compute all x values in A that are not *disqualified*. An x value is *disqualified* if by attaching a

y value from B, we obtain a tuple $\langle x,y \rangle$ that is not in A. We can compute disqualified tuples using the algebra expression

$$\pi_x((\pi_x(A) \times B) - A)$$

Thus we can define A/B as

$$\pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

To understand the division operation in full generality, we have to consider the case when both x and y are replaced by a set of attributes.

More Examples of Relational Algebra Queries

We illustrate queries using thei nstances $S3$ of Sailors, $R2$ of Reserves, and $B1$ of Boats, shown in Figures 4.15,4.16,and4.17, respectively.

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

Figure 4.15 An Instance $S3$ of Sailors

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

Figure 4.16 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	*Interlak*	*blue*
102	*Interlak*	*red*
103	*Clipper*	*green*
104	*Marine*	*red*

Figure 4.17 An Instance $B1$ of Boats

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

$\pi_{sname}((\sigma_{bid=103}Reserves) \lhd\rhd Sailors)$

We first compute the set of tuples in Reserves with $bid = 103$ and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances $R2$ and $S3$, it yields a relation that contains just one field, called *sname*, and three tuples $\langle Dustin \rangle$, $\langle Horatio \rangle$,and

$\langle Lubber \rangle$. (Observe that there are two sailors called Horatio, and only one of them has reserved a red boat.)

We can break this query into smaller pieces using the renaming operator ρ:

$$\rho(Temp1, \sigma_{bid=103} Reserves)$$

$$\rho(Temp2, Temp1 \lhd\rhd Sailors)\ \pi_{sname}(Temp2)$$

$Temp1$ denotes an intermediate relation that identifies reservations of boat 103. $Temp2$ is another intermediate relation, and it denotes sailors who have made a reservation in the set $Temp1$. The instances of these relations when evaluating this query on the instances $R2$ and $S3$ are illustrated in Figures 4.18 and 4.19. Finally, we extract the *sname* column from $Temp2$.

sid	*bid*	*day*
22	103	10/8/98
31	103	11/6/98
74	103	9/8/98

Figure 4.18 Instance of $Temp1$

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	103	10/8/98
31	Lubber	8	55.5	103	11/6/98
74	Horatio	9	35.0	103	9/8/98

Figure 4.19 Instance of $Temp2$

$$\pi_{sname}(\sigma_{bid=103}(Reserves \lhd\rhd Sailors))$$

The DBMS translates an SQL query into (an extended form of) relational algebra, and then looks for other algebra expressions that will produce the same answers but are cheaper to evaluate. If the user's query is first translated into the expression

$$\pi_{sname}(\sigma_{bid=103}(Reserves \lhd\rhd Sailors))$$

a good query optimizer will find the equivalent expression

$$\pi_{sname}((\sigma_{bid=103} Reserves) \lhd\rhd Sailors)$$

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi_{sname}((\sigma_{color='red'} Boats) \lhd\rhd Reserves \lhd\rhd Sailors$$

This query involves a series of two joins. First we choose (tuples describing) red boats. Then we join this set with Reserves (natural join, with equality specified on the *bid* column) to identify reservations of red boats. Next we join the resulting intermediate

relation with Sailors (natural join, with equality specified on the *sid* column) to retrieve the names of sailors who have made reservations of red boats. Finally, we project the sailors' names. The answer, when evaluated on the instances $B1$, $R2$ and $S3$, contains the names Dustin, Horatio, and Lubber.An equivalent expression is:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}\ Boats) \lhd\rhd Reserves) \lhd\rhd Sailors)$$

The reader is invited to rewrite both of these queries by using ρ to make the intermediate relations explicit and to compare the schemas of the intermediate relations. The second expression generates intermediate relations with fewer fields (and is therefore likely to result in intermediate relation instances with fewer tuples, as well). A relational query optimizer would try to arrive at the second expression if it is given the first.

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'}\ Sailors) \lhd\rhd Reserves \lhd\rhd Boats)$$

This query is very similar to the query we used to compute sailors who reserved red boats. On instances $B1$, $R2$, and $S3$, the query will return the colors gren and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \lhd\rhd Reserves)$$

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(T\ empboats,\ (\sigma_{color='red'}\ Boats)\ U\ (\sigma color='green'\ Boats))$$
$$\pi_{sname}(Tempboats \lhd\rhd Reserves \lhd\rhd Sailors)$$

We identify the set of all boats that are either red or green (Tempboats, which contains boats with the *bid*s 102, 103, and 104 on instances $B1$, $R2$, and $S3$). Then we join with Reserves to identify *sid*s of sailors who have reserved one of these boats; this gives us *sid*s 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of *sid*s) with Sailors to find the names of Sailors with these *sid*s. This gives us the names Dustin, Horatio, and Lubber on the instances $B1$, $R2$, and $S3$. Another equivalent definition is the following:

$$\rho(T\ empboats,\ (\sigma_{color='red' \vee color='green'}\ Boats))$$
$$\pi_{sname}(Tempboats \lhd\rhd Reserves \lhd\rhd Sailors)$$

(Q6) Find the names of sailors who have reserved a red and a green boat

$$\rho(T\ empboats2,\ (\sigma_{color='red'}\ Boats)\ \cap\ (\sigma color='green'\ Boats))\ \pi_{sname}(Tempboats2 \lhd\rhd Reserves \lhd\rhd Sailors)$$

However, this solution is incorrect—it instead tries to compute sailors who have reserved a boat that is both red and green. (Since *bid* is a key for Boats, a boat can be only one color; this query will always return an empty answer set.) The correct approach is to find sailors who have reserved a red boat, then sailors who have reserved a green boat, and then take the intersection of these two sets:

$$\rho(T\ empred,\ \pi_{sid}((\sigma_{color='red'}\ Boats) \lhd\rhd Reserves))$$

$$\rho(T\ empgreen,\ \pi_{sid}((\sigma_{color='green'}\ Boats) \lhd\rhd Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \lhd\rhd Sailors)$$

The two temporary relations compute the *sid*s of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances $B1$, $R2$, and $S3$, the *sid*s of sailors who have reserved a red boat are 22, 31, and 64. The *sid*s of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.

This formulation of Query Q6 can easily be adapted to find sailors who have reserved red *or* green boats (Query Q5); just replace $\cap$ by $\cup$:

$$\rho(T\ empred,\ \pi_{sid}((\sigma_{color='red'}\ Boats) \lhd\rhd Reserves))$$
$$\rho(T\ empgreen,\ \pi_{sid}((\sigma_{color='green'}\ Boats) \lhd\rhd Reserves))$$
$$\pi_{sname}((Tempred \cup Tempgreen) \lhd\rhd Sailors)$$

In the above formulations of Queries Q5 and Q6, the fact that *sid* (the field over which we compute union or intersection) is a key for Sailors is very important. Consider the following attempt to answer Query Q6:

$$\rho(T\ empred,\ \pi_{sname}((\sigma_{color='red'}\ Boats) \lhd\rhd Reserves \lhd\rhd Sailors))$$

$$\rho(T\ empgreen,\ \pi_{sname}((\sigma_{color='green'}\ Boats) \lhd\rhd Reserves \lhd\rhd Sailors))\ T\ empred \cap T\ empgreen$$

This attempt is incorrect for a rather subtle reason. Two distinct sailors with the same name, such as Horatio in our example instances, may have reserved red and

green boats, respectively. In this case, the name Horatio will (incorrectly) be included in the answer even though no one individual called Horatio has reserved a red boat and a green boat. The cause of this error is that *sname* is being used to identify sailors (while doing the intersection) in this version of the query, but *sname* is not a key.

(Q7) Find the names of sailors who have reserved at least two boats.

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \lhd\rhd Reserves))$$

$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$

$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$

$$\pi_{sname1}\sigma_{(sid1=sid2) \cap (bid1=bid2)} Reservationpairs$$

First we compute tuples of the form $\langle sid,sname,bid \rangle$, where sailor *sid* has made a reservation for boat *bid*; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: In order to show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances $B1$, $R2$, and $S3$, the sailors with *sid*s 22, 31, and 64 have each reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

Notice that we included *sid* in Reservations because it is the key field identifying sailors, and we need it to check that two Reservations tuples involve the same sailor. As noted in the previous example, we can't use *sname* for this purpose.

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$$\pi_{sid}(\sigma_{age>20} Sailors) - \pi_{sid}((\sigma_{color='red'} Boats) \lhd\rhd Reserves \lhd\rhd Sailors)$$

This query illustrates the use of the set-difference operator. Again, we use the fact that *sid* is the key for Sailors. We first identify sailors aged over 20 (over instances $B1$, $R2$, and $S3$, *sid*s 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (*sid*s 22, 31, and 64), to obtain the answer (*sid*s 29, 32, 58, 74,

85, and 95). If we want to compute the names of such sailors, we must first compute their *sid*s (as shown above), and then join with Sailors and project the *sname* values.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$$\rho(T\ empsids,\ (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \lhd\rhd Sailors)$$

The intermediate relation Tempsids is defined using division, and computes the set of *sid*s of sailors who have reserved every boat (over instances $B1$, $R2$, and $S3$, this is just *sid* 22). Notice how we define the two relations that the division operator (/) is applied to—the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sid*s such that there is a tuple $\langle sid,bid \rangle$ in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sid*s; for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(T\ empsids,\ (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}\ Boats)))\ \pi_{sname}(Tempsids \lhd\rhd Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute only *bid*s of boats named *Interlake* in defining the second argument to the division operator. Over instances $B1$, $R2$, and $S3$, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus that we present in detail is called the tuple relational calculus (TRC). Variables in TRC take on tuples as values. In another variant, called

the domain relational calculus (DRC), the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE.

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form $\{ T \mid p(T) \}$, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to true with $T = t$. The language for writing formulas $p(T)$ is thus at the heart of TRC and is essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

(Q11) Find all sailors with a rating above 7.

$\{S \mid S \in Sailors \wedge S.rating > 7\}$

When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test *S.rating*>*7* is applied. The answer contains those instances of S that pass this test. On instance $S3$ of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let Rel be a relation name, R and S be tuple variables, a an attribute of R, and b an attribute of S. Let op denote an operator in the set $\{<, >, =, \leq, \geq, =\}$. An atomic formula is one of the following:

a. $R \in Rel$
b. $R.a$ op $S.b$
c. $R.a$ op *constant*, or *constant* op $R.a$

A formula is recursively defined to be one of the following, where p and q are themselves formulas, and $p(R)$ denotes a formula in which the variable R appears:

any atomic formula

$\neg p$, $p \vee q$, $p \wedge q$, or $p \Rightarrow q$

$\exists R(p(R))$, where R is a tuple variable

$\forall R(p(R))$, where R is a tuple variable

We observe that every variable in a TRC formula appears in a subformula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type*, in the programming language sense. Informally, an atomic formula $R \in Rel$ gives R the type of tuples in Rel, and comparisons such as $R.a$ op $S.b$ and $R.a$ op *constant* induce type restrictions on the field $R.a$. If a variable R does not appear in an atomic formula of the form $R \in Rel$ (i.e., it appears only in atomic formulas that are comparisons), we will follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We will not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and that this is the domain associated with each field of each relation.)

A TRC query is defined to be expression of the form $\{T \mid p(T)\}$, where T is the only free variable in the formula p.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The answer to a TRC query $\{T \mid p(T)\}$, as we noted earlier, is the set of all tuples t for which the formula $p(T)$ evaluates to true with variable T assigned the tuple value t. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to true.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables,

with respect to the given database instance, F evaluates to (or simply 'is') true if one of the following holds:

- F is an atomic formula $R \in Rel$, and R is assigned a tuple in the instance of relation Rel.
- F is a comparison $R.a$ op $S.b$, $R.a$ op *constant*, or *constant* op $R.a$, and the tuples assigned to R and S have field values $R.a$ and $S.b$ that make the comparison true.
- F is of the form $\neg p$, and p is not true; or of the form $p \wedge q$, and both p and q are true; or of the form $p \vee q$, and one of them is true, or of the form $p \Rightarrow q$ and q is true whenever[4] p is true.
- F is of the form $\exists R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$, including the variable R,[5] that makes the formula $p(R)$ true.
- F is of the form $\forall R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$ that makes the formula $p(R)$ true no matter what tuple is assigned to R.

Examples of TRC Queries

We now illustrate the calculus through several examples, using the instances $B1$ of Boats, $R2$ of Reserves, and $S3$ of Sailors shown in Figures 4.15, 4.16, and 4.17. We will use parentheses as needed to make our formulas unambiguous. Often, a formula $p(R)$ includes a condition $R \in Rel$, and the meaning of the phrases *some tuple* R and *for all tuples* R is intuitive. We will use the notation $\exists R \in Rel(p(R))$ for $\exists R(R \in Rel \wedge p(R))$. Similarly, we use the notation $\forall R \in Rel(p(R))$ for $\forall R(R \in Rel \Rightarrow p(R))$.

(Q12) Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of P that are mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form $P \in Relname$. The result of this query is a relation with two fields, *name* and *age*. The atomic formulas $P.name = S.sname$

and $P.age = S.age$ give values to the fields of an answer tuple P. On instances $B1$, $R2$, and $S3$, the answer is the set of tuples $\langle Lubber, 55.5 \rangle$, $\langle Andy, 25.5 \rangle$, $\langle Rusty, 35.0 \rangle$, $\langle Zorba, 16.0 \rangle$, and $\langle Horatio, 35.0 \rangle$.

(Q13) Find the sailor name, boat id, and reservation date for each reservation

$$\{P \mid \exists R \in Reserves \quad \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

For each Reserves tuple, we look for a tuple in Sailors with the same *sid*. Given a pair of such tuples, we construct an answer tuple *P* with fields *sname, bid*, and *day* bycopying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances $B1$, $R2$, and $S3$ is shown in Figure 4.20.

sname	*bid*	*day*
Dustin	*10*	*10/10/98*
Dustin	*102*	*10/10/98*
Dustin	*103*	*10/8/98*
Dustin	*104*	*10/7/98*
Lubber	*102*	*11/10/98*
Lubber	*103*	*11/6/98*
Lubber	*104*	*11/12/98*
Horati	*101*	*9/5/98*
Horati	*102*	*9/8/98*
Horati	*103*	*9/8/98*

Figure 4.20 Answer to Query Q13

(Q1) Find the names of sailors who have reserved boat 103.

$$\{P \mid \exists S \in Sailors \; \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the *sid* field, and with $bid = 103$." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, *sname*

(Q2) Find the names of sailors who have reserved a red boat.

$$\{P \mid \exists S \in Sailors \quad \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.color = 'red'$." Another way to write this query, which corresponds more closely to this reading, is as follows:

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors \; \exists R1 \in Reserves \; \exists R2 \in Reserves \; (S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors \quad \forall B \in Boats \; (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Notice how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \in \forall B \in Boats \; (B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply the sailor having reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor's name, we have avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances $B1$, $R2$, and $S3$, the answer contains the Sailors tuples with *sid*s 22

and 31.We can write this query without using implication, by observing that an expression of the form $p \Rightarrow q$ is logically equivalent to $\neg p \wedge q$:

$$\{S \mid S \in Sailors \in \forall B \in Boats$$
$$(B.color \neq 'red' \cup (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

Domain Relational Calculus

A domain variable is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{\langle x_1, x_2, \ldots, x_n\rangle \mid p(\langle x_1, x_2, \ldots, x_n\rangle)\}$, where each x_i is either a *domain variable* or a constant and $p(\langle x_1, x_2, \ldots, x_n\rangle)$ denotes a DRC formula whose only free variables are thevari-ables among the x_i, $1 \leq i \leq n$. The result of this query is the set of all tuples $\langle x_1, x_2, \ldots, x_n\rangle$ for which the formula evaluates to true.

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{<, >, =, \leq, \geq, =\}$ and let X and Y be domain variables. An atomic formula in DRC is one of the following:

$\langle x_1, x_2, \ldots, x_n\rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \leq i \leq n$ is either a variable or a constant.

X op Y

X op *constant*, or *constant* op X

A formula is recursively defined to be one of the following, where p and q are themselves formulas, and $p(X)$ denotes a formula in which the variable X appears:

any atomic formula

$\neg p, p \vee q, p \wedge q,$ or $p \Rightarrow q$

$\exists X(p(X))$, where X is a domain variable

$\forall X(p(X))$, where X is a domain variable

Examples of DRC Queries

We now illustrate DRC through several examples. The reader is invited to compare these with the TRC versions.

(Q11) Find all sailors with a rating above 7.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition $\langle I, N, T, A\rangle \in Sailors$ ensures that the domain variables I, N, T, and A arerestricted to be fields of the *same* tuple. In comparison with the TRC query, we can say $T > 7$ instead of $S.rating > 7$, but we must specify the tuple $\langle I, N, T, A\rangle$ in the result, rather than just S.

(Q1) Find the names of sailors who have reserved boat 103.

$$\{\langle N \rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Ir, Br, D(\langle Ir, Br, D\rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

(Q2) Find the names of sailors who have reserved a red boat.

$$\{\langle N \rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists I, Br, D\rangle \in Reserves \wedge \exists \langle Br, BN, 'red'\rangle \in Boats)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{\langle N \rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 = Br2)$$

Notice how the repeated use of variable I ensures that the same sailor has reserved both the boats in question.

(Q9) Find the names of sailors who have reserved all boats.

$$\{\langle N \rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \wedge$$

$$(\exists \langle Ir, Br, D \rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

This query can be read as follows: "Find all values of N such that there is some tuple $\langle I, N,T,A \rangle$ in Sailors satisfying the following condition: for every $\langle B, BN,C \rangle$, either this is not a tuple in Boats or there is some tuple $\langle Ir, Br, D \rangle$ in Reserves that proves that Sailor I has reserved boat B." The $\forall$ quantifier allows the domain variables B, BN, and C to range over all values in their respective attribute domains, and the pattern '$\neg(\langle B, BN, C \rangle \in Boats) \vee$' is necessary to restrict attention to those values that appear in tuples of Boats.

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.

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

Figure 5.1 An Instance $S3$ of Sailors

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

Figure 5.2 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	*Interlake*	*blue*
102	*Interlake*	*red*
103	*Clipper*	*green*
104	*Marine*	*red*

Figure 5.3 An Instance $B1$ of Boats

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.

(Q15) Find the names and ages of all sailors.

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

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.

The answer to this query with and without the keyword DISTINCT on instance $S3$ of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

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

Figure 5.4 Answer to Q15

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

Figure 5.5 Answer to Q15 without DISTINCT

(Q11) 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

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 *{<, <=, = , <>, >=, >}*.[2] 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.

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.

(Q1) Find the names of sailors who have reserved boat number 103.

It can be expressed in SQL as follows.SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103Let us compute the answer to this query on the instances *R*3 of Reserves and *S*4 of Sailors shown in Figures 5.6 and 5.7, since the

computation on our usual example instances (*R*2 and *S*3) would be unnecessarily tedious.

sid	*sna*	*rati*	*age*
22	*Dus*	*7*	*45.0*
31	*Lub*	*8*	*55.5*
58	*Rust*	*10*	*35.0*

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

Figure 5.6 Instance *R*3 of Reserves

Figure 5.7 Instance *S*4 of Sailors

The first step is to construct the cross-product *S*4 × *R*3, which is shown in Figure 5.8.

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

Figure 5.8 *S4 × R3*

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 Figure 5.8. The third step is to eliminate unwanted columns; only *sname* appears in the SELECT clause. This step leaves us with the result shown in Figure 5.9, which is a table with a single column and, as it happens, just one row.

sname
rusty

Figure 5.9 Answer to Query Q1 on R3 and S4

Examples of Basic SQL Queries

Query Q1, which we discussed in the previous section, can also be expressed as follows:

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

(Q16) 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'

This query contains a join of two tables, followed by a selection on the color of boats.We can think of B and R as rows in the corresponding tables that 'prove' that a sailor with sid R.sid reserved a red boat B.bid. On our example instances *R*2 and *S*3.

(Q2) 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'

This query contains a join of three tables followed by a selection on the color of boats. The join with Sailors allows us to find the name of the sailor who, according to Reserves tuple R, has reserved a red boat described by tuple B.

(Q3) 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'

This query is very similar to the previous one. Notice that in general there may be more than one sailor called Lubber (since *sname* is not a key for Sailors); this query is still correct in that it will return the colors of boats reserved by *some* Lubber, if there are several sailors called Lubber

(Q4) 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

The join of Sailors and Reserves ensures that for each selected *sname*, the sailor has made some reservation. (If a sailor has not made a reservation, the second step in the conceptual evaluation strategy would eliminate all rows in the cross-product that involve this sailor.)

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.

(Q17) 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.
```

(Q18) 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'
```

The only such sailor is Bob, and his age is 63.5.

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.[4] SQL also provides other set operations: IN (to check if an element is in a given set),op ANY,op ALL(tocom-pare 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. We cover UNION, INTERSECT, and EXCEPT in this section.

Consider the following query:

(Q5) 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:

(Q6) Find the names of sailors who have reserved both a red and a green boat.

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. A correct statement of Query Q6 using AND is the following:

```
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 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 (Query Q6) 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'
```

(Q19) Find the **sid***s 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:

(Q20) 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

A nested query is a querythat has another query embedded within it; the embedded query is called a subquery.

Introduction to Nested Queries

As an example, let us rewrite the following query, which we discussed earlier, using a nested subquery:

(Q1) 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 $R2$ and $S3$), 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!

(Q2) 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 $B1$). The subquery one level above finds the set of *sid*s of sailors who have reserved one of these boats. On instances $B1$, $R2$, and $S3$, this set of *sid*s contains 22, 31, and64.

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.

(Q21) 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:

(Q1) 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*.

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.)

(Q22) 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.

(Q23) Find sailors whose rating is better than every sailor called Horatio.

We can obtain all such queries with a simple modification to Query Q22: just replace ANY with ALL in the WHERE clause of the outer query. On instance *S*3, we would get the *sid*s 58 and 71. If there were no sailor called Horatio, the comparison *S.rating* > ALL . . . is defined to return true! The query would then return the names of all sailors. Again, it is useful to think of the comparison being carried out repeatedly. Intuitively, the comparison must be true for every returned row in order for *S.rating* > ALL . . . to return true.

As another illustration of ALL, consider the following query:

(Q24) Find the sailors with the highest rating.

```
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.

(Q6) 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' )
```

As it turns out, writing this query (Q6) 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.

(Q9) 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 ))
```

Intuitively, for each sailor we check that there is no boat that has not been reserved by this sailor.

AGGREGATE OPERATORS

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).

(Q25) Find the average age of all sailors.

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

On instance *S*3, the average age is 37.4. Of course, the WHERE clause can be used to restrict the sailors who are considered in computing the average age:

(Q26) Find the average age of sailors with a rating of 10.

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

There are two such sailors, and their average age is 25.5. MIN (or MAX) can be used instead of AVG in the above queries to find the age of the youngest (oldest) sailor.

(Q27) Find the name and age of the oldest sailor. Consider the following attempt to answer this query:

```
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! (The intuition behind this restriction should become clear when we discuss the GROUP BY clause in Section 5.5.1.) Thus, we cannot use MAX (S.age) as well as S.sname in the SELECT clause. We have to use a nested query to compute the desired answer to Q27:

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

Observe that we have used the result of an aggregate operation in the subquery as an argument to a comparison operation. Strictly speaking, we are comparing an age value with the result of the subquery, which is a relation. However, because of the use of the aggregate operation, the subquery is guaranteed to return a single tuple with a single field, and SQL converts such a relation to a field value for the sake of the comparison. The following equivalent query for Q27 is legal in the SQL-92 standard but is not supported in many systems:

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

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.

(Q28) Count the number of sailors.

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

We can think of * as shorthand for all the columns (in the cross-product of the from-list in the FROM clause). Contrast this query with the following query, which computes the number of distinct sailor names. (Remember that *sname* is not a key!)

(Q30) 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 S3, the oldest sailor with rating 10 is sailor 58, names of older sailors are Bob, Dustin, Horatio, and Lubber. 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 )
```

The GROUP BY and HAVING Clauses

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.

(Q31) Find the age of the youngest sailor for each rating level.

If we know that ratings are integers in the range 1 to 10, we could write 10 queries of the form:

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

where $i = 1, 2, \ldots, 10$. 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 Q31 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:
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.

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.

If the GROUP BY clause is omitted, the entire table is regarded as a single group. We will explain the semantics of such a query through an example. Consider the query:

Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECT   S.rating, MIN (S.age) AS minage
FROM    Sailors S
WHERE S.age >= 18
```

GROUP BY S.rating
HAVING COUNT (*) > 1

Extending the conceptual evaluation strategy presented in Section 5.2, we proceed as follows. The first step is to construct the cross-product of tables in the from-list. Because the only relation in the from-list in Query Q32 is Sailors, the result is just the instance shown in Figure 5.10.

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	*Horati*	*7*	*35.0*
71	*Zorba*	*10*	*16.0*
74	*Horatio*	*9*	*35.0*
85	*Art*	*3*	*25.5*
95	*Bob*	*3*	*63.5*

Figure 5.10 Instance S3 of Sailors

The second step is to apply the qualification in the WHERE clause, *S.age* >= *18*. This step eliminates the row ⟨71, *zorba*, 10, 16⟩. The third step is to eliminate unwanted columns. Only columns mentioned in the SELECT clause, the GROUP BY clause, or the HAVING clause are necessary, which means we can eliminate *sid* and *sname* in our example. The result is shown in Figure 5.13. The fourth step is to sort the table

Rating	minage
3	25.5
7	35.0
8	25.5

Figure 5.13Final Result in Sample Evaluation

More Examples of Aggregate Queries

Q33) 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

On instances *B*1 and *R*2, the answer to this query contains the two tuples ⟨102, 3⟩ and ⟨104, 2⟩.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'
```

(Q34) 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
```

After identifying groups based on *rating*, we retain only groups with at least two sailors. The answer to this query on instance *S*3 is shown in Figure 5.14.

rating	*avgage*
3	44.5
7	40.0
8	40.5
10	25.5

Figure 5.14 Q34 Answer

rating	*avgage*
3	45.5
7	40.0
8	40.5
10	35.0

Figure 5.15 Q35 Answer

rating	*avgage*
3	45.5
7	40.0
8	40.5

Figure 5.16 Q36 Answer

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

(Q35) 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
```

(Q36) 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 )
```

The above formulation of the query reflects the fact that it is a variant of Q35. The answer to Q36 on instance $S3$ is shown in Figure 5.16. It differs from the answer to Q35 in that there is no tuple for rating 10, since there is only one tuple with rating 10 and *age* ≥ 18.

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

This formulation of Q36 takes advantage of the fact that the WHERE clause is applied before grouping is done; thus, only sailors with *age* > 18 are left when grouping is done. It is instructive to consider yet another way of writing this query:

```
SELECT Temp.rating, Temp.avgage
FROM ( SELECT  S.rating, AVG ( S.age ) AS avgage,
                    COUNT (*) AS ratingcount
          FROM     Sailors S WHERE S. age > 18 GROUP BY  S.rating ) AS Temp
WHERE Temp.ratingcount > 1
```

This alternative brings out several interesting points. First, the FROM clause can also contain a nested subquery according to the SQL-92 standard.[6] Second, the HAVING clause is not needed at all. Any query with a HAVING clause can be rewritten without one, but many queries are simpler to express with the HAVING clause. Finally, when a subquery appears in the FROM clause, using the AS keyword to give it a name is necessary (since otherwise we could not express, for instance, the condition *Temp.ratingcount* > *1*).

(Q37) Find those ratings for which the average age of sailors is the minimum over all ratings.

We use this query to illustrate that aggregate operations cannot be nested. One might consider writing it as follows:

```
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 this 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 )
```

The answer to this query on instance $S3$ is $\langle 10, 25.5 \rangle$.As an exercise, the reader should consider whether the following query computes the same answer, and if not, why:

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

NULL VALUES

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 ⟨98, Dan, null, 39⟩ 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.)

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 de nition. Consider the Students and Enrolled relations. Suppose that we are often interested in nding the names and student identi ers of students who got a grade of B in some course, together with the cid for the course. We can de ne 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 elds called *name*, *sid*, and *course* with the same domains as the elds *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.)

This view can be used just like a base table, or explicitly stored table, in de ning new queries or views. Given the instances of Enrolled and Students shown in Figure 3.4, B-Students contains the tuples shown in Figure 3.18. Conceptually, whenever B-Students is used in a query, the view de nition is rst evaluated to obtain the corresponding instance of B-Students, and then the rest of the query is evaluated treating B-Students like any other relation referred to in the query.

name	*sid*	*course*

Jones	53666	History105
Guldu	53832	Reggae203

Figure 3.18 An Instance of the B-Students View

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 de ning 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 speci ed only on views that are de ned on a single base table using just selection and projection, with no use of aggregate operations. Such views are called updatable views. This de nition is oversimpli ed, 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
```

An important observation is that an INSERT or UPDATE may change the underlying base table so that the resulting (i.e., inserted or modi ed) 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 elds 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 de nition of the view.

We caution the reader that when a view is de ned in terms of another view, the inter-action between these view de nitions with respect to updates and the CHECK OPTION clause can be complex; we will not go into the details.

Need to Restrict View Updates

there are some fundamental problems with updates speci ed on views, and there is good reason to limit the class of views that can be updated. Consider the Students relation and a new relation called Clubs:

Clubs(*cname:* string, *jyear:* date, *mname:* string)

A tuple in Clubs denotes that the student called *mname* has been a member of the club *cname* since the date *jyear*.[4] Suppose that we are often interested in nding the names and logins of students with a gpa greater than 3 who belong to at least one club, along with the club name and the date they joined the club. We can de ne a view for this purpose:

```
CREATE VIEW ActiveStudents (name, login, club, since)
        AS SELECT S.sname, S.login, C.cname, C.jyear
           FROM Students S, Clubs C
           WHERE    S.sname = C.mname AND S.gpa > 3
```

Consider the instances of Students and Clubs shown in Figures 3.19 and 3.20. When

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

Figure 3.19 An Instance *C* of Clubs

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

Figure 3.20 An Instance *S*3 of Students

evaluated using the instances *C* and *S*3, ActiveStudents contains the rows shown in Figure 3.21.

name	*login*	*club*	*since*
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997

Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

Figure 3.21 Instance of ActiveStudents

Now suppose that we want to delete the row *hSmith, smith@ee, Hiking, 1997i* from ActiveStudents. How are we to do this? ActiveStudents rows are not stored explicitly but are computed as needed from the Students and Clubs tables using the view de nition. So we must change either Students or Clubs (or both) in such a way that evaluating the view de nition on the modi ed instance does not produce the row *hSmith, smith@ee, Hiking, 1997.i* This task can be accomplished in one of two ways: by either deleting the row *h53688, Smith, smith@ee, 18, 3.2i* from Students or deleting the row *hHiking, 1997, Smithi* from Clubs. But neither solution is satisfactory. Removing the Students row has the e ect of also deleting the row *hSmith, smith@ee, Rowing, 1998i* from the view ActiveStudents. Removing the Clubs row has the e ect of also deleting the row *hSmith, smith@math, Hiking, 1997i* from the view ActiveStudents. Neither of these side e ects is desirable. In fact, the only reasonable solution is to *disallow* such updates on views.

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 de nition 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 ref-erencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be speci ed. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

ALTER TABLE modi es 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 de nition of Students is modi ed 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 view

TRIGGERS

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 nonempty, 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 in Figure 5.19, 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
```

Figure 5.19 Examples Illustrating Triggers

One of the example triggers in Figure 5.19 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 in Figure 5.19 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 Figure 5.19, 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. Such a trigger is shown in Figure 5.20 and is an alternative to the triggers shown in Figure 5.19.

The definition in Figure 5.20 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. (The trigger in Figure 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

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

Figure 5.20 Set-Oriented Trigger

SCHEMA REFINEMENT

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

15.1.1 Problems Caused by Redundancy

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set from Chapter 2:

Hourly_Emps(*ssn*, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

In this chapter we will omit attribute type information for brevity, since our focus is on the grouping of attributes into relations. We will often abbreviate an attribute name to a single letter and refer to a relation schema by a string of letters, one per attribute. For example, we will refer to the Hourly_Emps schema as *SNLRWH* (*W* denotes the *hourly_wages* attribute).

The key for Hourly_Emps is *ssn*. In addition, suppose that the *hourly_wages* attribute is determined by the *rating* attribute. That is, for a given *rating* value, there is only one permissible *hourly_wages* value. This IC is an example of a *functional dependency*. It leads to possible redundancy in the relation Hourly_Emps, as illustrated in Figure 15.1.

If the same value appears in the *rating* column of two tuples, the IC tells us that the same value must appear in the *hourly_wages* column as well. This redundancy has several negative consequences:

ssn	*name*	*lot*	*rating*	*hourly wages*	*hours worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

An Instance of the Hourly Emps Relation

- Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times, redundancy leads to potential inconsistency. For example, the *hourly wages* in the first tuple could be updated without making a similar change in the second tuple, which is an example of an *update anomaly*. Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value, which is an example of an *insertion anomaly.*
- If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that *rating* value and its *hourly_wage* value (a *deletion anomaly*).

Let us consider whether the use of *null* values can address some of these problems. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies.

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

15.1.2 Use of Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other ICs) can be used to identify such situations and to suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of `smaller' relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as *decomposition* of the larger relation into the smaller relations.

We can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly_Emps2(*ssn*, *name*, *lot*, *rating*, *hours_worked*)
Wages(*rating*, *hourly_wages*)

The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 15.1 is shown in Figure 15.2.

ssn	*name*	*lot*	*rating*	*hours_worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	*hourly_wages*
8	10
5	7

Instances of Hourly Emps2 and Wages

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it also eliminates the potential for inconsistency. Notice that the insertion and deletion anomalies have also been eliminated.

15.1.3 Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decom-posed relation from corresponding instances of the smaller relations. The *dependency-preservation* property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of IC that generalizes the concept of a *key*. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \,!\, Y$ [1] if the following holds for every pair of tuples t_1 and t_2 in r:

If $t1{:}X = t2{:}X$, then $t1{:}Y = t2{:}Y$.

We use the notation $t1{:}X$ to refer to the projection of tuple t_1 onto the attributes in X, in a natural extension of our TRC notation (see Chapter 4) $t{:}a$ for referring to attribute a of tuple t. An FD $X \,!\, Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

Figure 15.3 illustrates the meaning of the FD $AB \,!\, C$ by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, AB is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the A

*fi*eld or the *B fi*eld, they can differ in the *C fi*eld without violating the FD. On the other hand, if we add a tuple *ha*1*; b*1*; c*2*; d*1*i* to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

A	*B*	*C*	*D*
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

An Instance that Satisfies *AB ! C*

Recall that a *legal* instance of a relation must satisfy all specified ICs, including all specified FDs. As noted in Section 3.2, ICs must be identified and specified based on the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does *not* hold. However, we can never deduce that an FD *does* hold by looking at one or more instances of the relation because an FD, like other ICs, is a statement about *all* possible legal instances of the relation.

A primary key constraint is a special case of an FD. The attributes in the key play the role of *X*, and the set of all attributes in the relation plays the role of *Y*. Note, however, that the definition of an FD does not require that the set *X* be minimal; the additional minimality condition must be met for *X* to be a key. If *X ! Y* holds, where *Y* is the set of all attributes, and there is some subset *V* of *X* such that *V ! Y* holds, then *X* is a *super key*; if *V* is a strict subset of *X*, then *X* is not a key.

In the rest of this chapter, we will see several examples of FDs that are not key constraints.

REASONING ABOUT FUNCTIONAL DEPENDENCIES

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further re ne relations obtained through ER design (or, for that matter, through other approaches to conceptual design). Before proceeding with the main task at hand, which is the discussion of such schema refinement techniques, we digress to examine FDs in more detail because they play such a central role in schema analysis and refinement.

Given a set of FDs over a relation schema *R*, there are typically several additional FDs that hold over *R* whenever all of the given FDs hold. As an example, consider:

Workers(*ssn*, *name*, *lot*, *did*, *since*)

We know that *ssn ! did* holds, since *ssn* is the key, and FD *did ! lot* is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same *ssn* value, they must have the same *did* value (from the first FD), and because they have the same *did* value, they must also have the same *lot* value (from the second FD). Thus, the FD *ssn ! lot* also holds on Workers.

We say that an FD *f* is implied by a given set *F* of FDs if *f* holds on every relation instance that satisfies all dependencies in *F*, that is, *f* holds whenever all FDs in *F* hold. Note that it is not sufficient for *f* to hold on some instance that satisfies all dependencies in *F*; rather, *f* must hold on *every* instance that satisfies all dependencies in *F*.

15.4.1 Closure of a Set of FDs

The set of all FDs implied by a given set *F* of FDs is called the closure of F and is denoted as F^+. An important question is how we can infer, or compute, the closure of a given set *F* of FDs. The answer is simple and elegant. The following three rules, called Armstrong's Axioms, can be applied repeatedly to infer all FDs implied by a set *F* of FDs. We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

- Reflexivity: If *X* *Y*, then *X ! Y*.
- Augmentation: If *X ! Y*, then *XZ ! YZ* for any *Z*.
- Transitivity: If *X ! Y* and *Y ! Z*, then *X ! Z*.

Armstrong's Axioms are sound in that they generate only FDs in F^+ when applied to a set *F* of FDs. They are complete in that repeated application of these rules will generate all FDs in the closure F^+. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about F^+:

- Union: If *X ! Y* and *X ! Z*, then *X ! YZ*.
- Decomposition: If *X ! YZ*, then *X ! Y* and *X ! Z*.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

use a more elaborate version of the Contracts relation:

Contracts (*contractid*, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *value*)

We denote the schema for Contracts as *CSJDPQV*. The meaning of a tuple in this relation is that the contract with *contractid C* is an agreement that supplier *S* (*supplierid*) will supply *Q* items of part *P* (*partid*) to project *J* (*projectid*) associated with

department *D* (*deptid*); the value *V* of this contract is equal to *value*.

The following ICs are known to hold:

1. The contract id *C* is a key: *C ! CSJDPQV*.
2. A project purchases a given part using a single contract: *JP ! C*.
3. A department purchases at most one part from a supplier: *SD ! P*.

Several additional FDs hold in the closure of the set of given FDs:

From *JP ! C*, *C ! CSJDPQV* and transitivity, we infer *JP ! CSJDPQV*.

From *SD ! P* and augmentation, we infer *SDJ ! JP*.

From *SDJ ! JP*, *JP ! CSJDPQV* and transitivity, we infer *SDJ ! CSJDPQV*. (Incidentally, while it may appear tempting to do so, we *cannot* conclude *SD ! CSDPQV*, canceling *J* on both sides. FD inference is not like arithmetic multiplication!)

We can infer several additional FDs that are in the closure by using augmentation or decomposition. For example, from *C ! CSJDPQV*, using decomposition we can infer:

C ! C, *C ! S*, *C ! J*, *C ! D*, etc.

Finally, we have a number of trivial FDs from the reflexivity rule.

NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are fi*rst normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our de nition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are

important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema *R* with attributes *ABC*. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD *A ! B*. Now if several tuples have the same *A* value, they must also have the same *B* value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

We will primarily discuss redundancy that is revealed by FD information. In Sec-tion 15.8, we discuss more sophisticated ICs called *multivalued dependencies* and *join dependencies* and normal forms based on them.

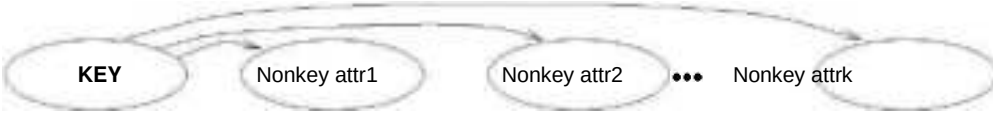
15.5.1 Boyce-Codd Normal Form

Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and let *A* be an attribute of *R*. *R* is in Boyce-Codd normal form if for every FD *X ! A* that holds over *R*, one of the following statements is true:

- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key.

Note that if we are given a set *F* of FDs, according to this de nition, we must consider each dependency *X ! A* in the closure F^+ to determine whether *R* is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in *F* is a super key (by computing the attribute closure and seeing if it includes all attributes of *R*).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: \Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)



FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.

X	*Y*	*A*
x	y_1	*a*
x	y_2	?

Instance Illustrating BCNF

This figure shows (two tuples in) an instance of a relation with three attributes *X*, *Y*, and *A*. There are two tuples with the same value in the *X* column. Now suppose that we know that this instance satisfies an FD *X ! A*. We can see that one of the tuples has the value *a* in the *A* column. What can we infer about the value in the *A* column in the second tuple? Using the FD, we can conclude that the second tuple also has the value *a* in this column. (Note that this is really the only kind of inference we can make about values in the fields of tuples by using FDs.)

But isn't this situation an example of redundancy? We appear to have stored the value *a* twice. Can such a situation arise in a BCNF relation? No! If this relation is in BCNF, because *A* is distinct from *X* it follows that *X* must be a key. (Otherwise, the FD *X ! A* would violate BCNF.) If *X* is a key, then $y_1 = y_2$, which means that the two tuples are identical. Since a relation is defined to be a *set* of tuples, we cannot have two copies of the same tuple and the situation shown in Figure 15.8 cannot arise.

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other elds in (all tuples of) the relation instance.

Third Normal Form

Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and *A* be an attribute of *R*. *R* is in third normal form if for every FD *X ! A* that holds over *R*, one of

the following statements is true:

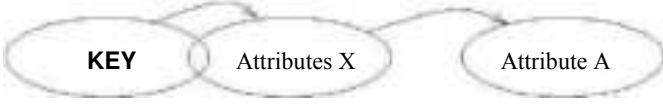
- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key, or
- *A* is part of some key for *R*.

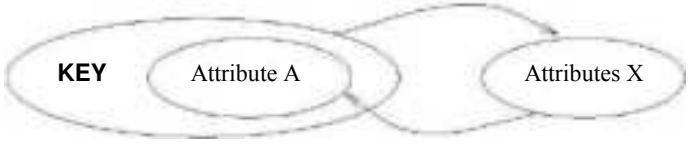
The de nition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determines all other attributes. *A* must be part of a key (any key, if there are several). It is not enough for *A* to be part of a super key, because the latter condition is satisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

.

Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure. Note that in Figure 15.10, the set *X* of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that *X* is not a subset of KEY.

Figure 15.9 Partial Dependencies





Transitive Dependencies

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties (Section 15.6). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise by settling for a

3NF design. As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.

DECOMPOSITIONS

As we have seen, a relation in BCNF is free of redundancy (to be precise, redundancy that can be detected using FD information), and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

A decomposition of a relation schema R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R. Intuitively, we want to store the information in any given instance of R by storing projections of the instance. This section examines the use of decompositions through several examples.

We begin with the Hourly_Emps example from Section 15.3.1. This relation has attributes *SNLRWH* and two FDs: *S ! SNLRWH* and *R ! W*. Since R is not a key and W is not part of any key, the second dependency causes a violation of 3NF.

The alternative design consisted of replacing Hourly_Emps with two relations having attributes *SNLRH* and *RW*. *S ! SNLRH* holds over *SNLRH*, and S is a key. *R ! W* holds over *RW*, and R is a key for *RW*. The only other dependencies that hold over these schemas are those obtained by augmentation. Thus both schemas are in BCNF.

Our decision to decompose *SNLRWH* into *SNLRH* and *RW*, rather than, say, *SNLR* and *LRWH*, was not just a good guess. It was guided by the observation that the dependency *R ! W* caused the violation of 3NF; the most natural way to deal with this violation is to remove the attribute W from this schema. To compensate for removing W from the main schema, we can add a relation *RW*, because each R value is associated with at most one W value according to the FD *R ! W*.

A very important question must be asked at this point: If we replace a legal instance r of relation schema *SNLRWH* with its projections on *SNLRH* (r_1) and *RW* (r_2), can we recover r from r_1 and r_2? The decision to decompose *SNLRWH* into *SNLRH* and *RW* is equivalent to saying that we will store instances r_1 and r_2 instead of r. However

it is the instance r that captures the intended entities or relationships. If we cannot compute r from r_1 and r_2, our attempt to deal with redundancy has e ectively thrown out the baby with the bathwater. We consider this issue in more detail below.

Lossless-Join Decomposition

Let R be a relation schema and let F be a set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a lossless-join decomposition with respect to F if for every instance r of R that satis es the dependencies in F, $_X(r)$ ./ $_Y(r) = r$.

This de nition can easily be extended to cover a decomposition of R into more than two relations. It is easy to see that $r \subseteq _X(r)$ ./ $_Y(r)$ always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3

Instance r

S	P
s1	p1
s2	p2
s3	p1

$_{SP}(r)$

P	D
p1	d1
p2	d2
p1	d3

$_{PD}(r)$

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

$_{SP}(r)$ ./ $_{PD}(r)$

Instances Illustrating Lossy Decompositions

By replacing the instance r shown in Figure 15.11 with the instances $_{SP}(r)$ and $_{PD}(r)$, we lose some information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that the relationships (s_1; p_1; d_3) and (s_3; p_1; d_1) do not hold. The decomposition of schema SPD into SP and PD is therefore a `lossy' decomposition if the instance r shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. (Observe the similarities between this example and the Contracts relationship set in Section 2.5.3.)

All decompositions used to eliminate redundancy must *be lossless.* The following simple test is very useful:

> Let R be a relation and F be a set of FDs that hold over R. The decomposition of R into relations with attribute sets R_1 and R_2 is lossless if and only if F^+ contains either the FD $R_1 \setminus R_2 \to R_1$ or the FD $R_1 \setminus R_2 \to R_2$.

In other words, the attributes common to R_1 and R_2 must contain a key for either R_1 or R_2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join.[2] If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the Hourly_Emps relation again. It has attributes *SNLRWH*, and the FD *R ! W* causes a violation of 3NF. We dealt with this violation by decomposing the relation into *SNLRH* and *RW*. Since *R* is common to both decomposed relations, and *R ! W* holds, this decomposition is lossless-join.

This example illustrates a general observation:

> If an FD *X ! Y* holds over a relation *R* and *X \ Y* is empty, the decomposition of *R* into *R − Y* and *XY* is lossless.

X appears in both *R −Y* (since *X \ Y* is empty) and *XY*, and it is a key for *XY*. Thus, the above observation follows from the test for a lossless-join decomposition.

Another important observation has to do with repeated decompositions. Suppose that a relation *R* is decomposed into *R1* and *R2* through a lossless-join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition. Then the decomposition of *R* into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then joining *R1* and *R2*, we can recover *R*.

Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *CSJDPQV* from Section 15.4.1. The given FDs are *C ! CSJDPQV*, *JP ! C*, and *SD ! P*. Because *SD* is not a key the dependency *SD ! P* causes a violation of BCNF.

We can decompose Contracts into two relations with schemas *CSJDQV* and *SDP* to address this violation; the decomposition is lossless-join. There is one subtle problem, however. We can enforce the integrity constraint *JP ! C* easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same *JP* values (as the inserted tuple) but different *C* values. Once we decompose Contracts into *CSJDQV* and *SDP*, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into *CSJDQV*. We say that this decomposition is not dependency-preserving.

Intuitively, a *dependency-preserving decomposition* allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To de ne dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let R be a relation schema that is decomposed into two schemas with attribute sets X and Y, and let F be a set of FDs over R. The projection of F on X is the set of FDs in the closure F^+ (not just F !) that involve only attributes in X. We will denote the projection of F on attributes X as F_X . Note that a dependency U ! V in F^+ is in F_X only if *all* the attributes in U and V are in X.

The decomposition of relation schema R with FDs F into schemas with attribute sets X and Y is dependency-preserving if $(F_X [F_Y)^+ = F^+$. That is, if we take the dependencies in F_X and F_Y and compute the closure of their union, we get back all dependencies in the closure of F. Therefore, we need to enforce only the dependencies in F_X and F_Y ; all FDs in F^+ are then sure to be satisfied. To enforce F_X , we need to examine only relation X (on inserts to that relation). To enforce F_Y , we need to examine only relation Y.

NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and de-compositions in database design, we now consider algorithms for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas. Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas

Decomposition into BCNF

We now present an algorithm for decomposing a relation schema R into a collection of BCNF relation schemas:

1. Suppose that R is not in BCNF. Let X R, A be a single attribute in R, and X ! A be an FD that causes a violation of BCNF. Decompose R into R −> A and X→A.
2. If either R −> A or X→A is not in BCNF, decompose them further by a recursive application of this algorithm.

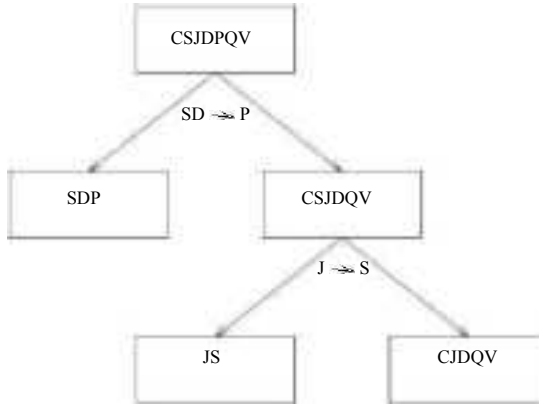
R −> A denotes the set of attributes other than A in R, and X→A denotes the union of attributes in X and A. Since X ! A violates BCNF, it is not a trivial dependency; further, A is a single attribute. Therefore, A is not in X; that is, X \A is empty. Thus,

each decomposition carried out in Step is lossless-join.

The set of dependencies associated with *R −> A* and *→XA* is the projection of *F* onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step. Since a decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relation schemas that are all in BCNF.

Consider the Contracts relation with attributes *CSJDPQV* and key *C*. We are given FDs *JP ! C* and *SD ! P*. By using the dependency *SD ! P* to guide the decomposition, we get the two schemas *SDP* and *CSJDQV*. *SDP* is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: *J ! S*. This means that the schema *CSJDQV* is not in BCNF. So we decompose it further into *JS* and *CJDQV*. *C ! JDQV* holds over *CJDQV*; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDs contain a key in the left side. Thus, each of the schemas *SDP*, *JS*, and *CJDQV* is in BCNF, and this collection of schemas also represents a lossless-join decomposition of *CSJDQV*.

The steps in this decomposition process can be visualized as a tree, as shown in Figure. The root is the original relation *CSJDPQV*, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, *SDP*, *JS*, and *CSDQV*. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.



Decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV*

Redundancy in BCNF Revisited

The decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV* is not dependency-preserving. Intuitively, dependency *JP ! C* cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes *CJP*. In e ect, this solution amounts to storing some information redundantly in order to make the dependency enforcement cheaper.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation instances for *SDP* and *CJDQV* and project onto the attributes *CJP*, we must get exactly the instance stored in the relation with schema *CJP*. We saw in Section 15.5.1 that there is no such redundancy within a single BCNF relation. The current example shows that redundancy can still occur across relations, even though there is no redundancy within a relation.

Minimal Cover for a Set of FDs

A minimal cover for a set *F* of FDs is a set *G* of FDs such that:

1. Every dependency in *G* is of the form *X ! A*, where *A* is a single attribute.
2. The closure F^+ is equal to the closure G^+.
3. If we obtain a set *H* of dependencies from *G* by deleting one or more dependencies, or by deleting attributes from a dependency in *G*, then $F^+ 6= H^+$.

Intuitively, a minimal cover for a set *F* of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required in order for the closure to be equal to F^+.

As an example, let *F* be the set of dependencies:

A ! B, *ABCD ! E*, *EF ! G*, *EF ! H*, and *ACDF ! EG*.

First, let us rewrite *ACDF ! EG* so that every right side is a single attribute:

ACDF ! E and *ACDF ! G*.

Next consider *ACDF ! G*. This dependency is implied by the following FDs:

A ! B, *ABCD ! E*, and *EF ! G*.

Therefore, we can delete it. Similarly, we can delete *ACDF ! E*. Next consider *ABCD ! E*. Since *A ! B* holds, we can replace it with *ACD ! E*. (At this point, the reader should verify that each remaining FD is minimal and required.) Thus, a minimal cover for *F* is the set:

A ! B, *ACD ! E*, *EF ! G*, and *EF ! H*.

The preceding example suggests a general algorithm for obtaining a minimal cover of a set *F* of FDs:

1. Put the FDs in a standard form: Obtain a collection G of equivalent FDs with a single attribute on the right side (using the decomposition axiom).

2. Minimize the left side of each FD: For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to F^+.
3. Delete redundant FDs: Check each remaining FD in G to see if it can be deleted while preserving equivalence to F^+.

Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.

More important, it is necessary to minimize the left sides of FDs *before* checking for redundant FDs. If these two steps are reversed, the nal set of FDs could still contain some redundant FDs (i.e., not be a minimal cover), as the following example illustrates. Let F be the set of dependencies, each of which is already in the standard form:

$ABCD \to E$, $E \to D$, $A \to B$, and $AC \to D$.

Observe that none of these FDs is redundant; if we checked for redundant FDs rst, we would get the same set of FDs F. The left side of $ABCD \to E$ can be replaced by AC while preserving equivalence to F^+, and we would stop here if we checked for redundant FDs in F before minimizing the left sides. However, the set of FDs we have is not a minimal cover:

$AC \to E$, $E \to D$, $A \to B$, and $AC \to D$.

From transitivity, the first two FDs imply the last FD, which can therefore be deleted while preserving equivalence to F^+. The important point to note is that $AC \to D$ becomes redundant only after we replace $ABCD \to E$ with $AC \to E$. If we minimize left sides of FDs first and then check for redundant FDs, we are left with the rst three FDs in the preceding list, which is indeed a minimal cover for F.

Dependency-Preserving Decomposition into 3NF

Returning to the problem of obtaining a lossless-join, dependency-preserving decom-position into 3NF relations, let R be a relation with a set F of FDs that is a minimal cover, and let $R_1; R_2; : : : ; R_n$ be a lossless-join decomposition of R. For 1 i n, suppose that each R_i is in 3NF and let F_i denote the projection of F onto the attributes of R_i. Do the following:

Identify the set N of dependencies in F that are not preserved, that is, not included in the closure of the union of F_is.

- For each FD $X ! A$ in N, create a relation schema XA and add it to the decomposition of R.

Obviously, every dependency in F is preserved if we replace R by the R_is plus the schemas of the form XA added in this step. The R_is are given to be in 3NF. We can show that each of the schemas XA is in 3NF as follows: Since $X ! A$ is in the minimal cover F, $Y ! A$ does not hold for any Y that is a strict subset of X. Therefore, X is a key for XA. Further, if any other dependencies hold over XA, the right side can involve only attributes in X because A is a single attribute (because $X ! A$ is an FD in a minimal cover). Since X is a key for XA, none of these additional dependencies causes a violation of 3NF (although they might cause a violation of BCNF).

As an optimization, if the set N contains several FDs with the same left side, say, $X ! A_1; X ! A_2; : : : ; X ! A_n$, we can replace them with a single equivalent FD $X ! A_1 : : : A_n$. Therefore, we produce one relation schema $XA_1 : : : A_n$, instead of several schemas $XA_1; : : : ; XA_n$, which is generally preferable.

Consider the Contracts relation with attributes *CSJDPQV* and FDs *JP ! C*, *SD ! P*, and *J ! S*. If we decompose *CSJDPQV* into *SDP* and *CSJDQV*, then *SDP* is in BCNF, but *CSJDQV* is not even in 3NF. So we decompose it further into *JS* and *CJDQV*.
The relation schemas *SDP*, *JS*, and *CJDQV* are in 3NF (in fact, in BCNF), and the decomposition is lossless-join. However, the dependency *JP ! C* is not preserved. This problem can be addressed by adding a relation schema *CJP* to the decomposition.

This set of FDs is not a minimal cover, and so we must nd one. We rst replace *C ! CSJDPQV* with the FDs:

C ! S, *C ! J*, *C ! D*, *C ! P*, *C ! Q*, and *C ! V*.

The FD *C ! P* is implied by *C ! S*, *C ! D*, and *SD ! P*; so we can delete it. The FD *C ! S* is implied by *C ! J* and *J ! S*; so we can delete it. This leaves us with a minimal cover:

C ! J, *C ! D*, *C ! Q*, *C ! V*, *JP ! C*, *SD ! P*, and *J ! S*.

Using the algorithm for ensuring dependency-preservation, we obtain the relational schema *CJ*, *CD*, *CQ*, *CV*, *CJP*, *SDP*, and *JS*. We can improve this schema by combining relations for which *C* is the key into *CDJPQV*. In addition, we have *SDP* and *JS* in our decomposition. Since one of these relations (*CDJPQV*) is a super key, we are done.

Comparing this decomposition with the one that we obtained earlier in this section, we find that they are quite close, with the only difference being that one of them has *CDJPQV* instead of *CJP* and *CJDQV*. In general, however, there could be significant differences.

Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

OTHER KINDS OF DEPENDENCIES *

FDs are probably the most common and important kind of constraint from the point of view of database design. However, there are several other kinds of dependencies. In particular, there is a well-developed theory for database design using *multivalued dependencies* and *join dependencies*. By taking such dependencies into account, we can identify potential redundancy problems that cannot be detected using FDs alone.

This section illustrates the kinds of redundancy that can be detected using multivalued dependencies. Our main observation, however, is that simple guidelines (which can be checked using only FD reasoning) can tell us whether we even need to worry about complex constraints such as multivalued and join dependencies. We also comment on the role of *inclusion dependencies* in database design.

Multivalued Dependencies

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*. The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course. There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor. The instance shown in Figure 15.13 illustrates this situation.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

The relation schema *CTB* is in BCNF; thus we would not consider decomposing

- it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be expressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively, the multivalued dependency $X \;!!\; Y$ is said to hold over R if, in every legal instance r of R, each X value is associated with a set of Y values and this set is independent of the values in the other attributes.

Formally, if the MVD $X \;!!\; Y$ holds over R and $Z = R - XY$, the following must be true for every legal instance r of R:

> If $t_1 \; 2 \; r$, $t_2 \; 2 \; r$ and $t_1{:}X = t_2{:}X$, then there must be some $t_3 \; 2 \; r$ such that $t_1{:}XY = t_3{:}XY$ and $t_2{:}Z = t_3{:}Z$.

Figure illustrates this de nition. If we are given the first two tuples and told that the MVD $X \;!!\; Y$ holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the rst two tuples treating the first tuple as t_2 and the second tuple as t_1|we can deduce that the tuple t_4 must also be in the relation instance.

X	Y	Z	
a	b_1	c_1	tuple t_1
a	b_2	c_2	tuple t_2
a	b_1	c_2	tuple t_3
a	b_2	c_1	tuple t_4

This table suggests another way to think about MVDs: If $X \;!!\; Y$ holds over R, then $_Y Z(\ _{X=x}(R)) = {}_Y(\ _{X=x}(R))\ _Z(\ _{X=x}(R))$ in every legal instance of R, for any value x that appears in the X column of R. In other words, consider groups of tuples in R with the same X-value, for each X-value. In each such group consider the projection onto the attributes YZ. This projection must be equal to the cross-product of the projections onto Y and Z. That is, for a given X-value, the Y-values and Z-values are independent. (From this de nition it is easy to see that $X \;!!\; Y$ must hold whenever $X \;!\; Y$ holds. If the FD $X \;!\; Y$ holds, there is exactly one Y-value for a given X-value, and the conditions in the MVD de nition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our *CTB* example, the constraint that course texts are independent of instructors can be expressed as $C \;!!\; T$. In terms of the de nition of MVDs, this constraint can be read as follows:

> \If (there is a tuple showing that) C is taught by teacher T,
> and (there is a tuple showing that) C has book B as text,
> then (there is a tuple showing that) C is taught by T and has text B.

Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus ve additional rules. Three of the additional rules involve only MVDs:

- MVD Complementation: If $X \;!!\; Y$, then $X \;!!\; R - XY$.
- MVD Augmentation: If $X \;!!\; Y$ and $W \quad Z$, then $WX \;!!\; YZ$.
- MVD Transitivity: If $X \;!!\; Y$ and $Y \;!!\; Z$, then $X \;!!\; (Z - Y)$.

As an example of the use of these rules, since we have $C \;!!\; T$ over CTB, MVD complementation allows us to infer that $C \;!!\; CT\ B - CT$ as well, that is, $C \;!!\; B$. The remaining two rules relate FDs and MVDs:

- Replication: If $X \;!\; Y$, then $X \;!!\; Y$.
- Coalescence: If $X \;!!\; Y$ and there is a W such that $W \setminus Y$ is empty, $W \;!\; Z$, and Y Z, then $X \;!\; Z$.

Observe that replication states that every FD is also an MVD.

Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let R be a relation schema, X and Y be nonempty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in fourth normal form (4NF) if for every MVD X !! Y that holds over R, one of the following statements is true:

- Y X or $XY = R$, or
- X is a Superkey.

In reading this definition, it is important to understand that the de nition of a key has not changed the key must uniquely determine all attributes through FDs alone. X !! Y is a trivial MVD if Y X R or $XY = R$; such MVDs always hold.

The relation CTB is not in 4NF because C !! T is a nontrivial MVD and C is not a key. We can eliminate the resulting redundancy by decomposing CTB into CT and CB; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions detected using only FD information!|under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

> If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation*. This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

Figure shows three tuples from an instance of $ABCD$ that satisfies the given MVD B !! C. From the definition of an MVD, given tuples t_1 and t_2, it follows

B	C	A	D	
b	c_1	a_1	d_1	| tuple t_1
b	c_2	a_2	d_2	| tuple t_2
b	c_1	a_2	d_2	| tuple t_3

Three Tuples from a Legal Instance of $ABCD$

that tuple t_3 must also be included in the instance. Consider tuples t_2 and t_3. From the given FD A ! BCD and the fact that these tuples have the same A-value, we can

deduce that $c_1 = c_2$. Thus, we see that the FD *B ! C* must hold over *ABCD* whenever the FD *A ! BCD* and the MVD *B !! C* hold. If *B ! C* holds, the relation *ABCD* is not in BCNF (unless additional FDs hold that make *B* a key)!

Join Dependencies

A join dependency is a further generalization of MVDs. A join dependency (JD) *./ fR*$_1$*; : : : ; R*$_n$*g* is said to hold over a relation *R* if *R*$_1$*; : : : ; R*$_n$ is a lossless-join decomposition of *R*.

An MVD *X !! Y* over a relation *R* can be expressed as the join dependency *./ f*XY, X(R−Y)*g*. As an example, in the *CTB* relation, the MVD *C !! T* can be expressed as the join dependency *./ f*CT, CB*g*.

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form

A relation schema *R* is said to be in fth normal form (5NF) if for every JD *./ fR*$_1$*; : : : ; R*$_n$*g* that holds over *R*, one of the following statements is true:

- *R*$_i$ = *R* for some *i*, or
- The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of *R* into *fR*$_1$*; : : : ; R*$_n$*g* is lossless-join whenever the key dependencies (FDs in which the left side is a key for *R*) hold. *./ fR*$_1$*; : : : ; R*$_n$*g* is a trivial JD if *R*$_i$ = *R* for some *i*; such a JD always holds.

The following result, also due to Date and Fagin, identifies conditions again, detected using only FD information under which we can safely ignore JD information.

> If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF *without ever identifying the MVDs and JDs that may hold over the relation*.

INTENTION LOCK MODES

A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity).

Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, *Q*—must traverse a path in the tree from the root to *Q*. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure: Compatibility matrix

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

1. Transaction *Ti* must observe the lock-compatibility function of Figure above.
2. Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.
3. Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.
4. Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
5. Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
6. Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.

Locking: Top-Down and Bottom-up

Observe that the multiple-granularity protocol requires that locks be acquired in ***top-down*** (root-to-leaf) order, whereas locks must be released in ***bottom-up*** (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure (above) and these transactions:
• Suppose that transaction *T*21 reads record *ra*2 in file *Fa* . Then, *T*21 needs to lock the database, area *A*1, and *Fa* in IS mode (and in that order), and finally to lock *ra*2 in S mode.

• Suppose that transaction *T*22 modifies record *ra*9 in file *Fa* . Then, *T*22 needs to lock the database, area *A*1, and file *Fa* (and in that order) in IX mode, and finally to lock *ra*9 in X mode.

• Suppose that transaction *T*23 reads all the records in file *Fa* . Then, *T*23 needs to lock the database and area *A*1 (and in that order) in IS mode, and finally to lock *Fa* in S mode.

• Suppose that transaction *T*24 reads the entire database. It can do so after locking the database in S mode.

Precedence graph

Reference 'encyclopedia'

A **precedence graph**, also named **conflict graph** and **serializability graph**, is used in the context of **concurrency control in databases**.

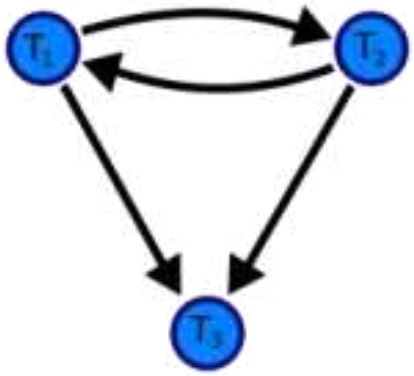
The precedence graph for a schedule S contains:

- A node for each committed transaction in S
- An arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j's actions.

Precedence graph example

Example 1:

$$D = \begin{bmatrix} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \\ & & \end{bmatrix}$$



Example 2

A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is *not* Conflict serializable.

Testing Serializability with Precedence Graph

The drawing sequence for the precedence graph:-

1. For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
2. For each case in S where T_i executes a write_item(X) then T_j executes a read_item(X), create an edge (T_i --> T_j) in the precedence graph. This occurs nowhere in the above example, as there is no read after write.
3. For each case in S where T_i executes a read_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in a directed edge from T_1 to T_2.
4. For each case in S where T_i executes a write_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edges from T_2 to T_1, T_1 to T_3, and T_2 to T_3.
5. The schedule S is conflict serializable if the precedence graph has no cycles. As T_1 and T_2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked using other methods.

Concurrent Executions

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- An example of a **serial** schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Note -- In schedules 1, 2 and 3, the sum “A + B” is preserved.

Schedule 4

- The following concurrent schedule does not preserve the sum of "$A + B$"

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Let I_i and I_j be two Instructions of transactions T_i and T_j respectively. Instructions I_i and I_j **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6 -- a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions. Therefore, Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

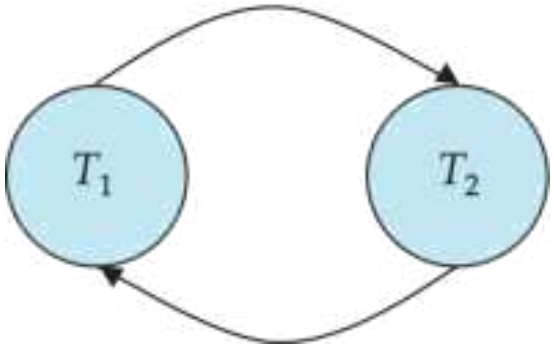
- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

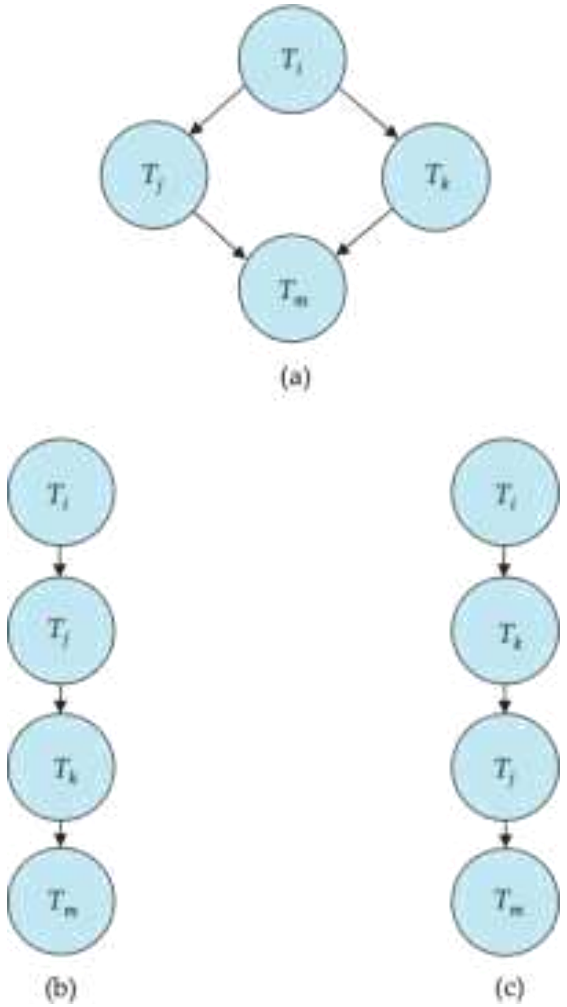
Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph**— a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example**



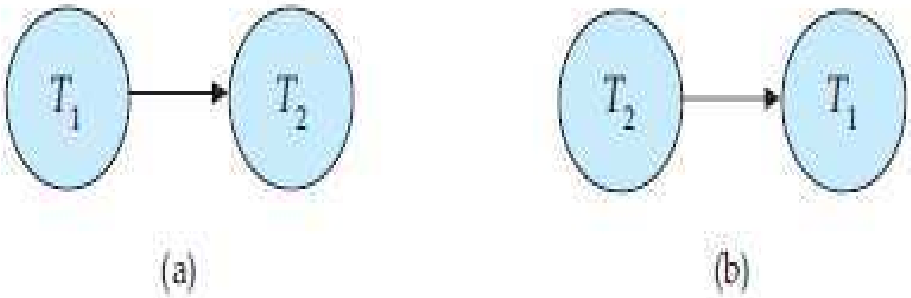
Testing for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- To test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - That is, a linear order consistent with the partial order of the graph.
  - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



- A **serializability order of the transactions can be obtained by finding a linear** order consistent with the partial order of the precedence graph. This process is called **topological sorting**.

Precedence graph of Schedule 1 and 2



Precedence graph for (a) schedule 1 and (b) schedule 2.

Example (Serializability)

We now present a simple and efficient method for determining conflict serializability of a schedule. Consider a schedule *S. We construct a directed graph,* called a **precedence graph, from*S. This graph consists of a pair G = (V, E), where*** *V is a set of vertices and E is a set of edges. The set of vertices consists of all the* transactions participating in the schedule.

The set of edges consists of all edges

Ti →Tj for which one of three conditions holds:

1. ***Ti executes write(Q) before Tj executes read(Q).***
2. ***Ti executes read(Q) before Tj executes write(Q).***
3. ***Ti executes write(Q) before Tj executes write(Q).***

Cycle-detection Algorithms

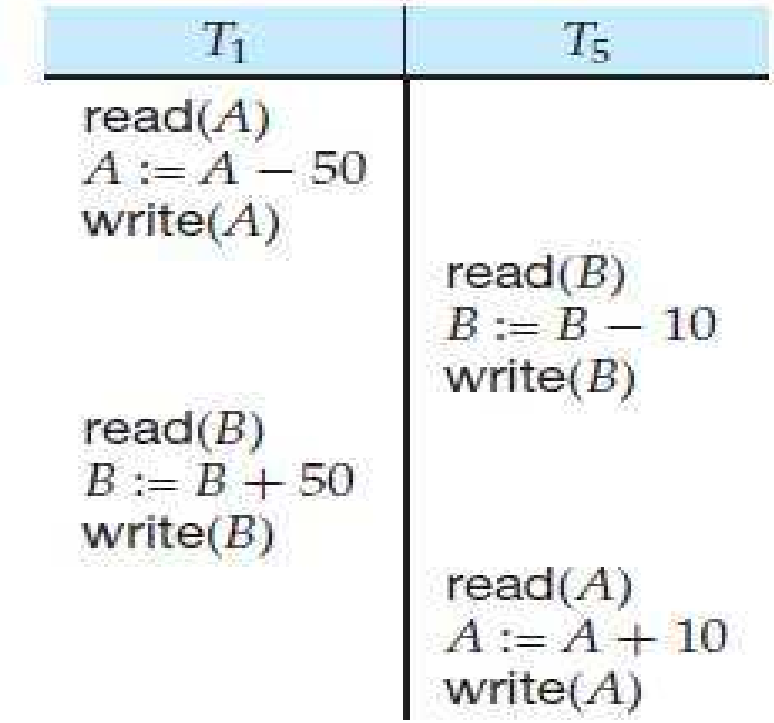
- To test for conflict serializability, we need to construct the precedence graph and to invoke a **cycle-detection algorithm.**
- Cycle-detection algorithms, such as those based on depth-first search, require on the order of *n2 operations, where n is the number of vertices in the graph (that is, the number of transactions).*

Example

- consider transaction *T5, which transfers* $10 from account *B to account A. Let schedule 8 be as defined in below figure,* We claim that schedule 8 is not conflict equivalent to the serial schedule *<T1,T5>,* since, in schedule 8, the write(*B) instruction of T5 conflicts with the read(B) instruction* of *T1. This creates an edge T5 → T1 in the precedence graph. Similarly,* we see that the write(*A) instruction of T1 conflicts with the read instruction of T5* creating an edge *T1 → T5. This shows that the precedence graph has a cycle and*

  that schedule 8 is not serializable.

Figure: schedule 8 →



- Example

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

Recoverable Schedules

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i **must** appear before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
  - Conflict serializable.
  - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
  - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);

Other Notions of Serializability

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However ,practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

More Complex Notions of Serializability

- The schedule below produces the same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

- If we start with A = 1000 and B = 2000, the final result is 960 and 2040
- Determining such equivalence requires analysis of operations other than read and write.

End of Concurrency Control Topic

Reference

Database System Concepts, 6th Ed.

Serializability – By M V Kamal

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable. Certainly, serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable. Since transactions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact.

For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: ***read*** and ***write***. We assume that, between a read(Q) instruction and a write(Q) instruction on a data item Q, a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. In this model, the only significant operations of a transaction, from a scheduling point of view, are its read and write instructions. Commit operations, though relevant, are not considered until Section 14.7.We therefore may show only read and write instructions in schedules, as we do for schedule 3 in Figure below. In this section, we discuss different forms of schedule equivalence, but focus on a particular form called **conflict serializability**.

Let us consider a schedule S in which there are two consecutive instructions, I and J , of transactions Ti and Tj , respectively (i _= j). If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule. However, if I and J refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. I = read(Q), J = read(Q). The order of I and J does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.

2. I = read(Q), J = write(Q). If I comes before J , then Ti does not read the value of Q that is written by Tj in instruction J . If J comes before I, then Ti reads the value of Q that is written by Tj. Thus, the order of I and J matters.

3. I = write(Q), J = read(Q). The order of I and J matters for reasons similar to those of the previous case.

4. I = write(Q), J = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after I and J in S, then the order of I and J directly affects the final value of Q in the database state that results from schedule S.

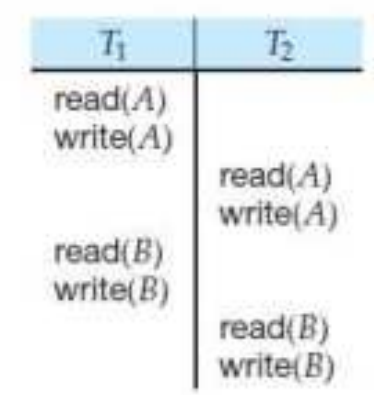


Fig: Schedule 3—showing only the read and write instructions.

We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3in Figure above. The write(A) instruction of $T1$ conflicts with the read(A) instruction of $T2$. However, the write(A) instruction of $T2$ does not conflict with the read(B) instruction of $T1$, because the two instructions access different data items.

Transaction Characteristics

-Prepared by M V Kamal, Associate Professor, CSE Dept

Every transaction has three characteristics: *access mode*, *diagnostics size*, and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded.

If the **access mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure given below. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. The highest degree of isolation from the effects of other

Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Figure: Transaction Isolation Levels in SQL-92

transactions is achieved by setting isolation level for a transaction T to SERIALIZABLE. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 19.3.1), and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that *T* reads only the changes made by committed transactions, and that no value read or written by *T* is changed by any other transaction until *T* is complete. However, *T* could experience the phantom phenomenon; for example, while *T* examines all Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by *T*.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that *T* reads only the changes made by committed transactions, and that no value written by *T* is changed by any other transaction until *T* is complete. However, a value read by *T* may well be modified by another transaction while *T* is still in progress, and *T* is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction *T* can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while *T* is in progress, and *T* is also vulnerable to the phantom problem.

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.

For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the **ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

```
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;
```

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction, (i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which two transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.

Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

Durability: It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

Partially Committed: At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

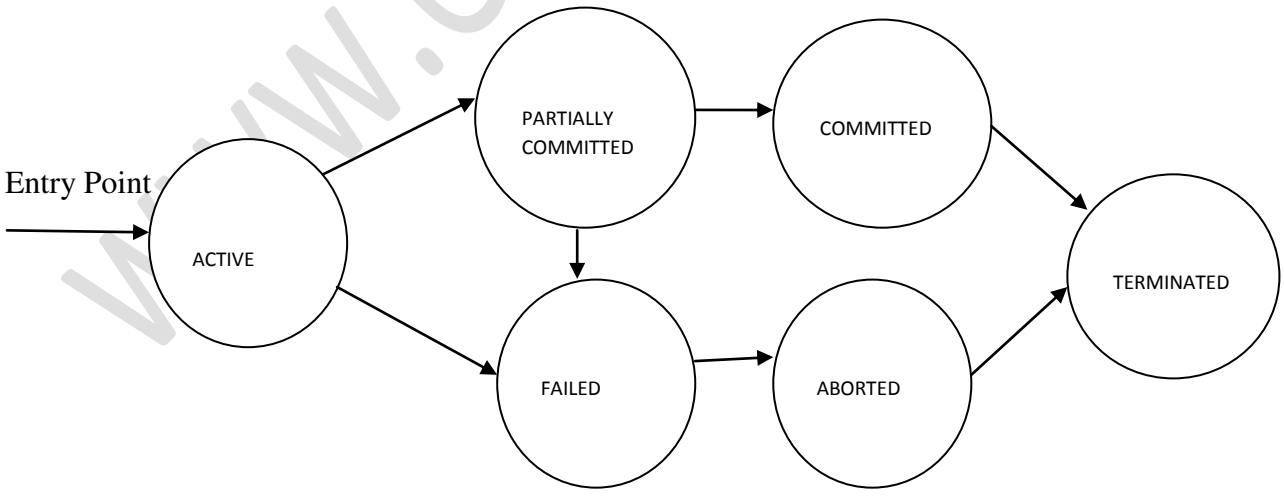
Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.

Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.

Terminated: Either committed or aborted, the transaction finally reaches this state.

The whole process can be described using the following diagram:



Concurrent Execution

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

T1

```
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;
```

T2

```
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;
```

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

T1	T2
Read A;	
A = A – 100;	
Write A;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Read B;	
B = B + 100;	
Write B;	

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

T1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

In the above example, we detected the error simple by examining the schedule and applying common sense. But there must be some well formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules. This brings us to our next topic, the concept of Serializability.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

It may happen that we may want to execute the same set of transaction in a different schedule on another day. Keeping in mind these rules, we may sometimes alter parts of one schedule (S1) to create another schedule (S2) by swapping only the non-conflicting parts of the first schedule. The conflicting parts cannot be swapped in this way because the ordering of the conflicting instructions is important and cannot be changed in any other schedule that is derived from the first. If these two schedules are made of the same set of transactions, then both S1 and S2 would yield the same result if the conflict resolution rules are maintained while creating the new schedule. In that case the schedule S1 and S2 would be called **Conflict Equivalent**.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules

S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

Except in these three cases, any alteration can be possible while creating S2 by modifying S1.

The above notes are submitted by:

Sabyasachi De

MCA

sabyasachide@yahoo.com

Recovery System

(Unit-IV- Part B)

Reference

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted as result of a system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory,
    non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies.

  Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Stable-Storage Implementation (Cont.)

Protecting storage media from failure during data transfer (cont.):

- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **System buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is denoted by x_i.
  - B_X denotes block containing X
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - The **write**(X) can be executed at any time before the transaction commits
- Note that **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access

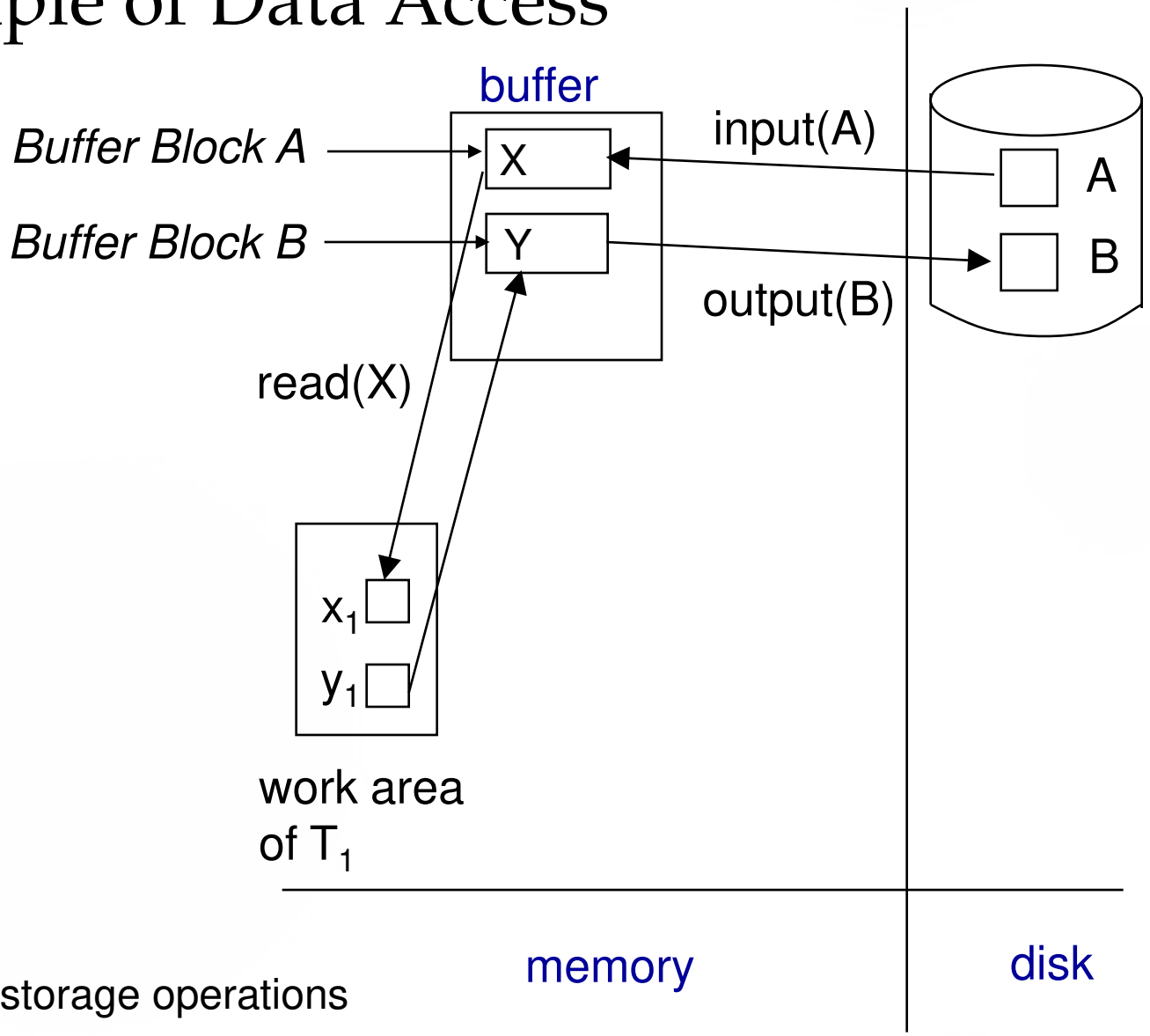


Fig: Block storage operations

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

Log and Log Records

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, which maintains information about update activities on the database.
- There are several types of log records. An update log record describes a single database write. It has these fields:
  - Transaction Identifier
  - Data-Item Identifier
  - Old Value
  - New Value

Log-Based Recovery

- When transaction T_i starts, it registers itself by writing a record
  $<T_i$ **start**$>$ to the log
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record $<T_i$ **commit**$>$ is written.
- **<Ti abort>.**Transaction Ti has aborted.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification

Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written ***before*** a database item is written
  - We assume that the log record is output directly to stable storage
- Output of updated blocks to disk storage can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Undo and Redo Operations

- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
    - Each time a data item X is restored to its old value V a special log record (called r**edo-only**) <T_i , X, V> is written out
    - When undo of a transaction is complete, a log record <T_i **abort**> is written out (to indicate that the undo was completed)
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
    - No logging is done in this case

Undo and Redo Operations (Cont.)

- The **undo** and **redo** operations are used in several different circumstances:
  - The **undo is** used for transaction rollback during normal operation(in case a transaction cannot complete its execution due to some logical error).
  - The **undo** and **redo** operations are used during recovery from failure.
- We need to deal with the case where during recovery from failure another failure occurs prior to the system having fully recovered.

Transaction rollback (during normal operation)

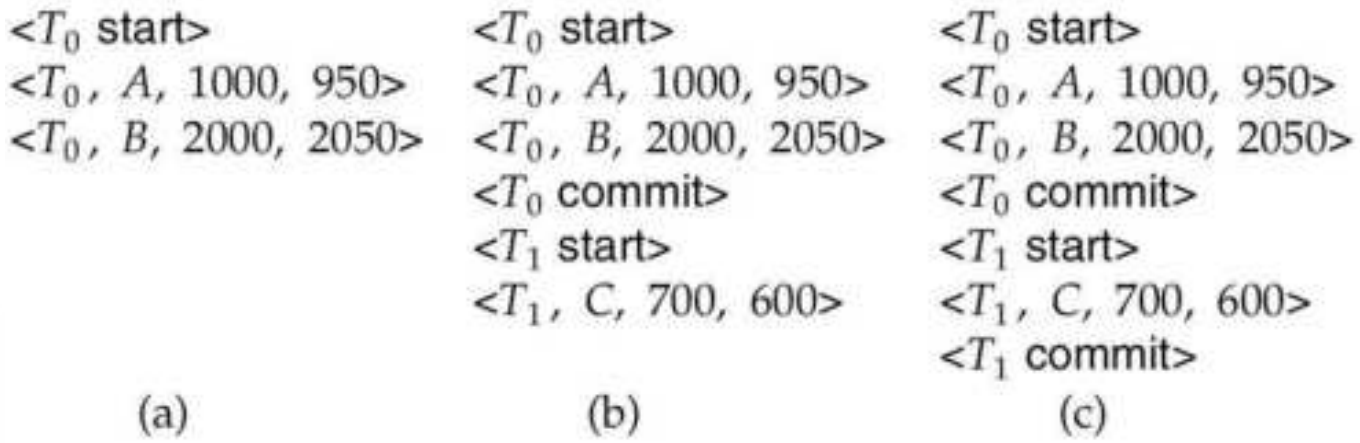
- Let T_i be the transaction to be rolled back
- Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
  - Perform the undo by writing V_1 to X_j,
  - Write a log record <T_i , X_j, V_1>
    - such log records are called **compensation log records**
- Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Undo and Redo on Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - contains the record <T_i **start**>,
    - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - contains the records <T_i **start**>
    - and contains the record <T_i **commit**> *or* <T_i **abort**>
  - It may seem strange to redo transaction T_i if the record <T_i **abort**> record is in the log. To see why this works, note that if <T_i **abort**> is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo T_i 's modifications in this case. This slight redundancy simplifies the recovery algorithm and enables faster overall recovery time.
    - such a redo redoes all the original actions including the steps that restored old value. Known as **repeating history**

Immediate Modification Recovery Example

Below we show the log as it appears at three instances of time.



Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050

respectively. Then *C* is set to 600

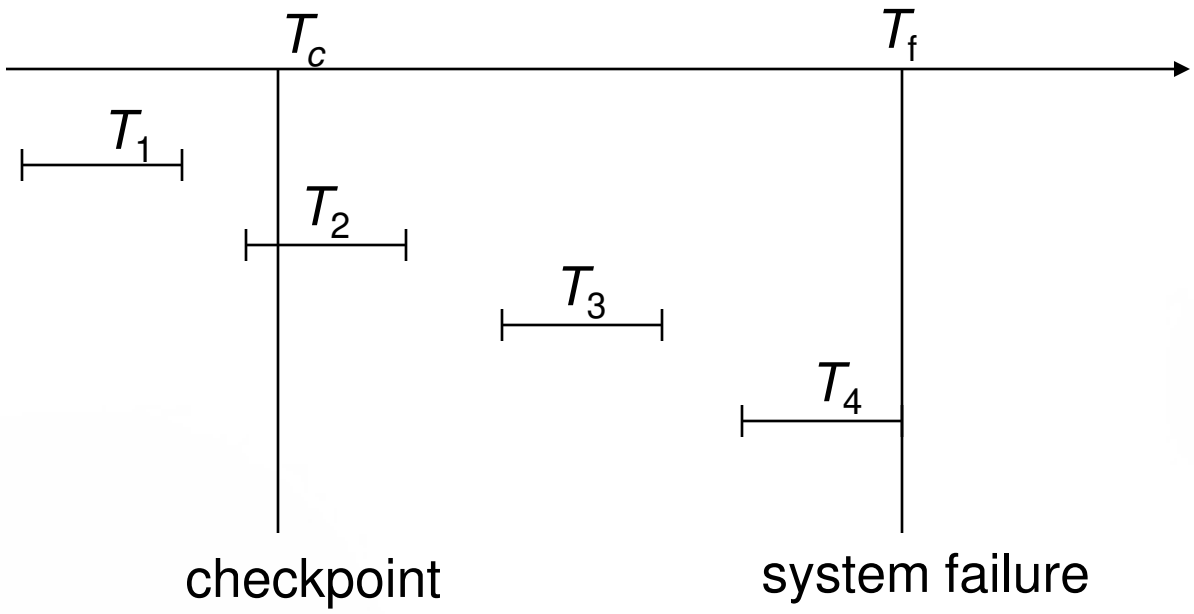
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
- All updates are stopped while doing checkpointing
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

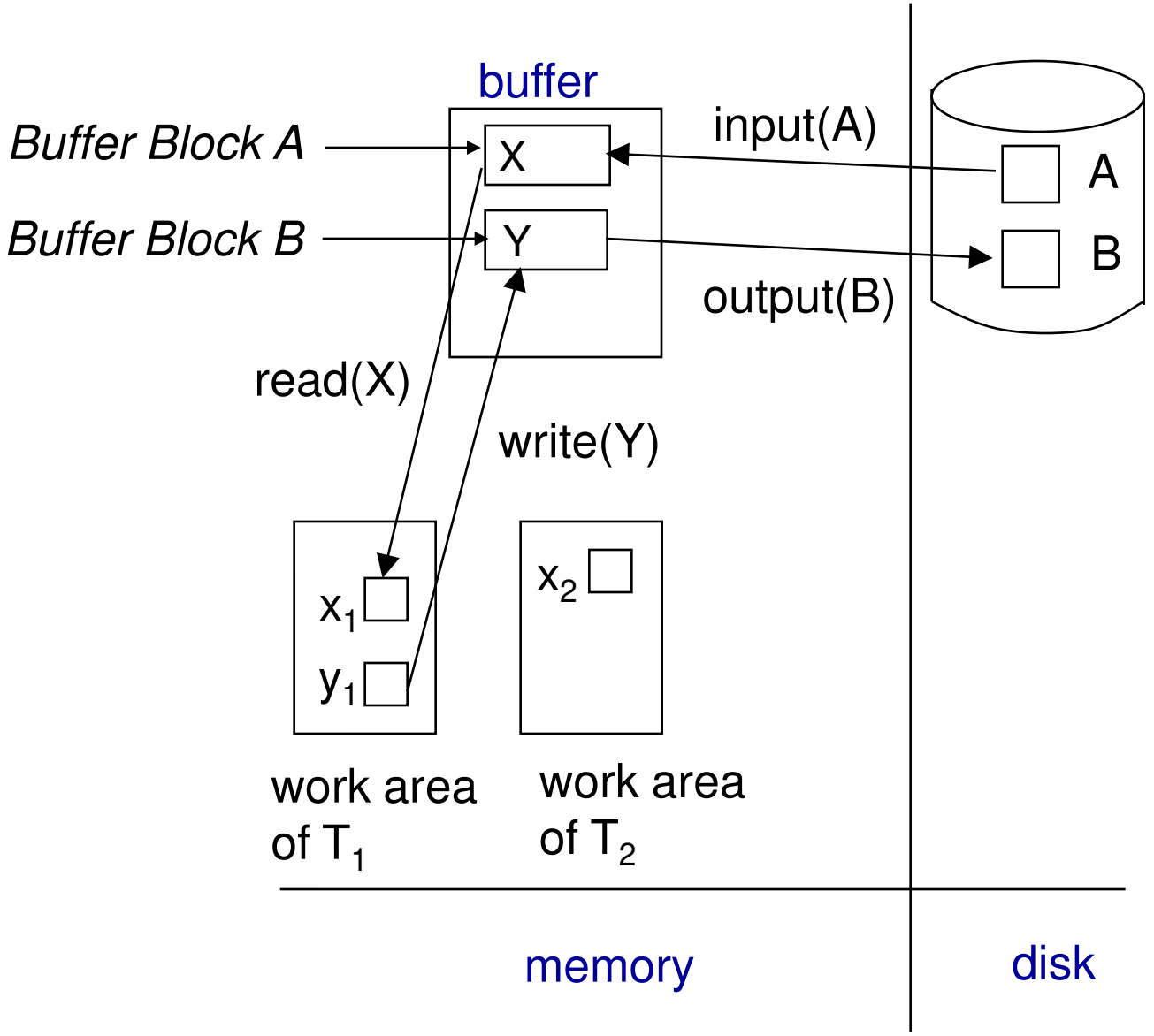
Disk Crash

- What happens if the disk crashes and the data on it is gone?

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how do we perform undo if T_1 updates A, then T_2 updates A and commits, and finally T_1 has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Example of Data Access with Concurrent transactions



Recovery Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - perform the undo by writing V_1 to X_j,
    - write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

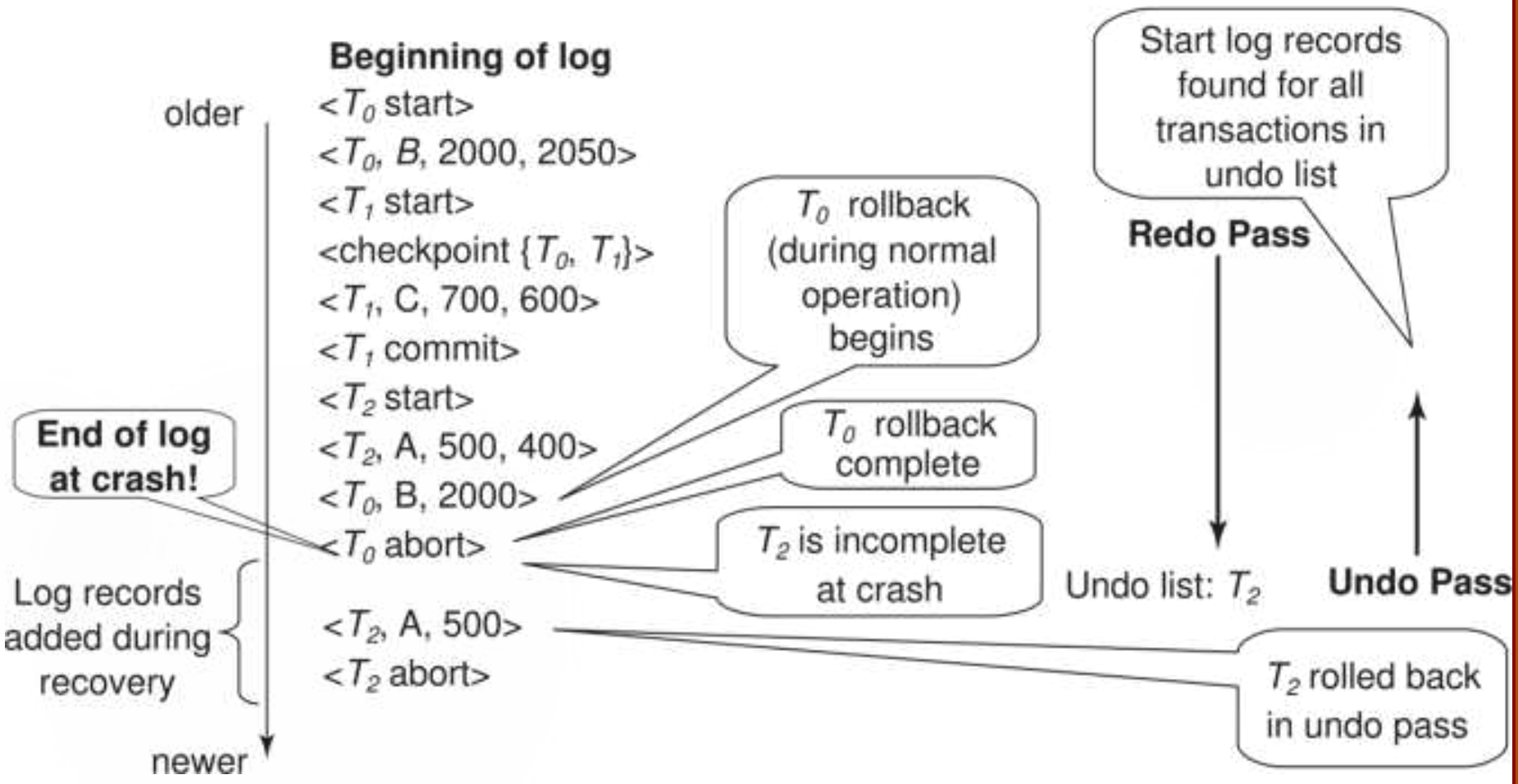
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record <T_i, X_j, V_1, V_2> is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e.,<T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**:i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     1. Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use

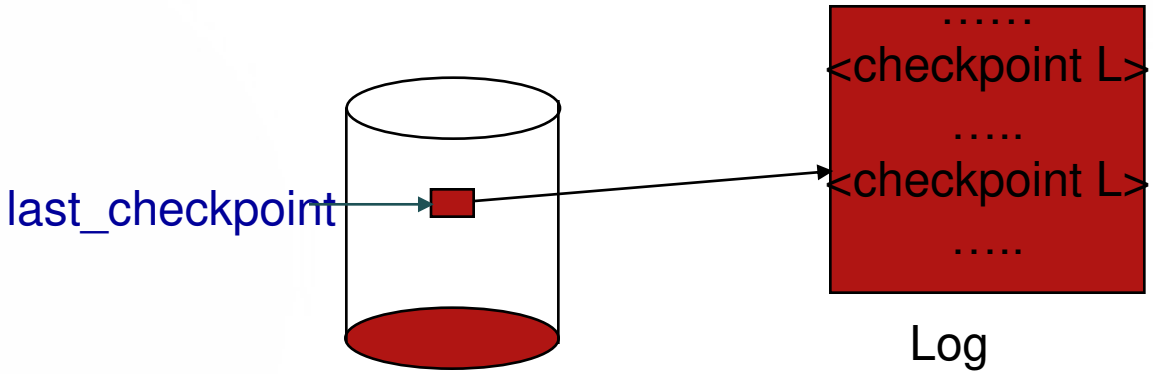
    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record **<dump>** to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
  - Similar to fuzzy checkpointing

Recovery with Early Lock Release and Logical Undo

Recovery with Early Lock Release

- Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  - Supports "logical undo"
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing

Logical Undo Logging

- Operations like B+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B+-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - Operations are called **logical operations**
  - Other examples:
    - delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

Physical Redo

- Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
  - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
  - Physical redo logging does not conflict with early lock release

Operation Logging

- Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Example: insert of (key, record-id) pair (K5, RID7) into index I9

<T1, O1, operation-begin>

….

<T1, X, 10, K5>

<T1, Y, 45, RID7>

Physical redo of steps in insert

<T1, O1, operation-end, (delete I9, K5, RID7)>

Operation Logging (Cont.)

- If crash/rollback occurs before operation completes:
  - the **operation-end** log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the **operation-end** log record is found, and in this case
  - logical undo is performed using *U*; the physical undo information for the operation is ignored.
- Redo of operation (after crash) still uses physical redo information.

Transaction Rollback with Logical Undo

Rollback of transaction T_i is done as follows:

- Scan the log backwards
  1. If a log record $<T_i, X, V_1, V_2>$ is found, perform the undo and log a al $<T_i, X, V_1>$.
  2. If a $<T_i, O_j,$ **operation-end**, $U>$ record is found
     - Rollback the operation logically using the undo information U.
       - Updates performed during roll back are logged just like during normal operation execution.
       - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record

         $<T_i, O_j,$ **operation-abort**$>$.
     - Skip all preceding log records for T_i until the record $<T_i, O_j$ **operation-begin**$>$ is found

Transaction Rollback with Logical Undo (Cont.)

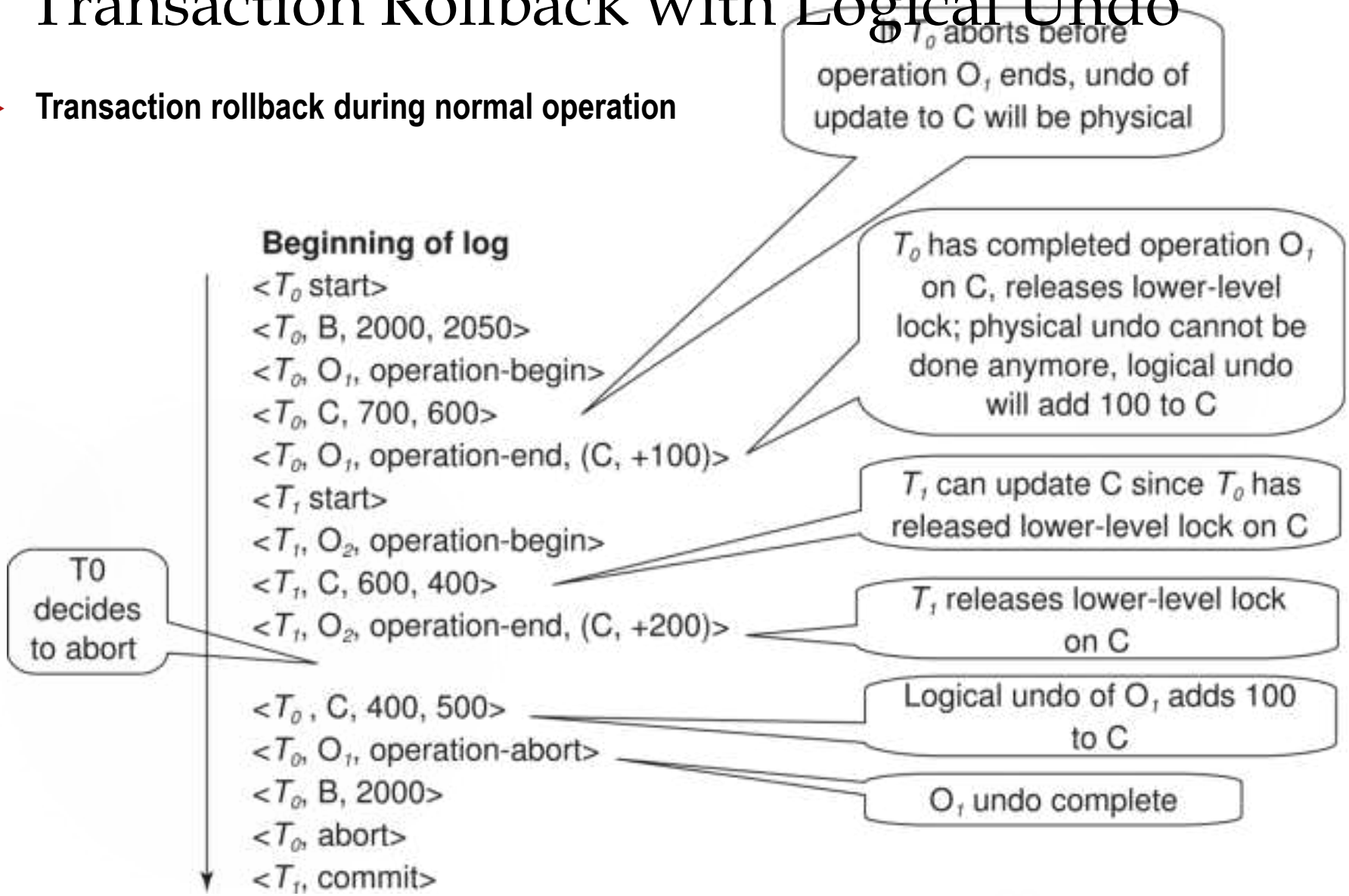
- Transaction rollback, scanning the log backwards (cont.):
  3. If a redo-only record is found ignore it
  4. If a <T_i, O_j, **operation-abort**> record is found:
     - skip all preceding log records for T_i until the record <T_i, O_j, **operation-begin**> is found.
  5. Stop the scan when the record <T_i, start> is found
  6. Add a <T_i, **abort**> record to the log

Some points to note:

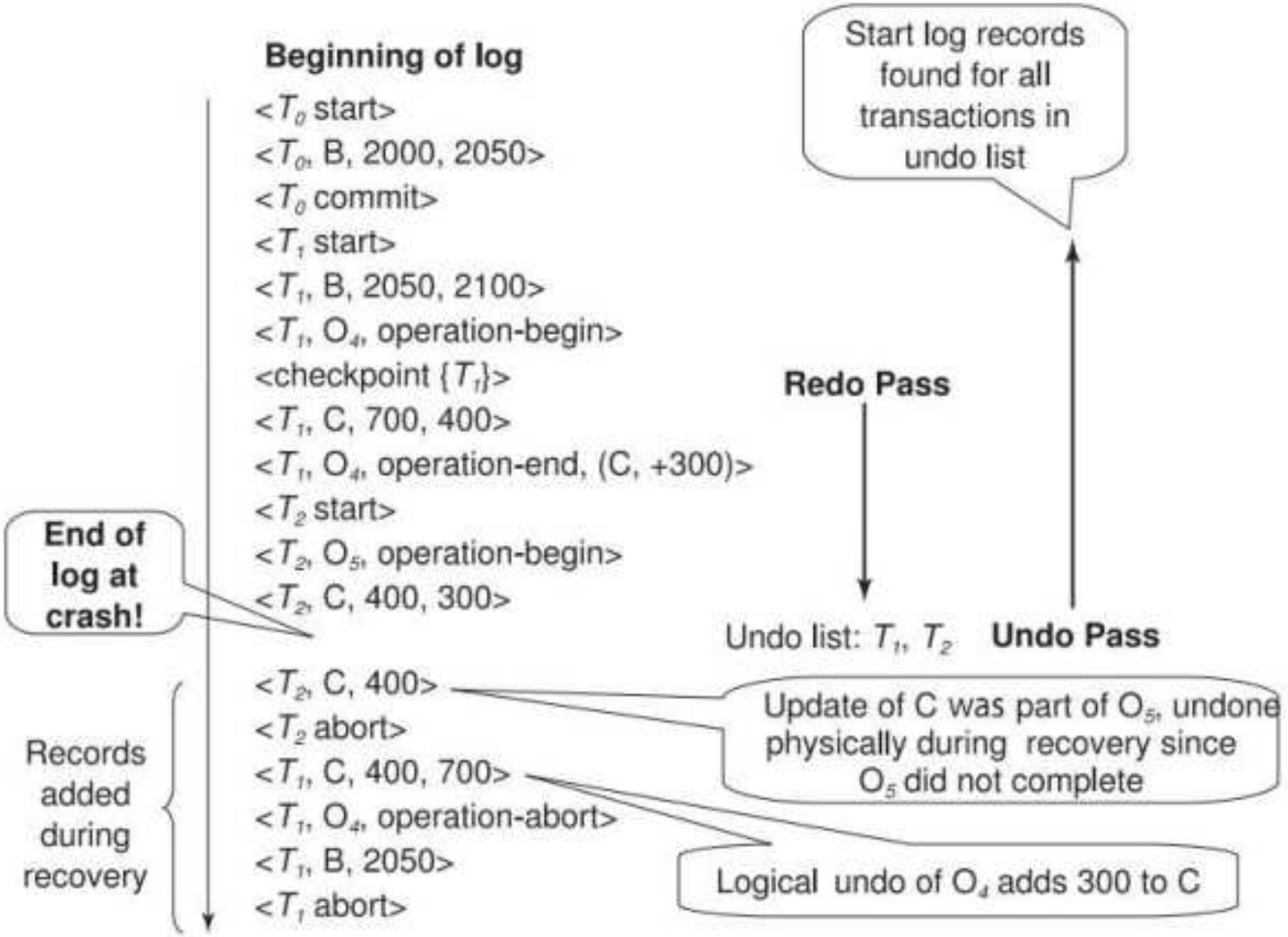
- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback with Logical Undo

- **Transaction rollback during normal operation**



Failure Recovery with Logical Undo



Transaction Rollback: Another Example

- Example with a complete and an incomplete operation

<T1, start>

<T1, O1, operation-begin>

….

<T1, X, 10, K5>

<T1, Y, 45, RID7>

<T1, O1, operation-end, (delete I9, K5, RID7)>

<T1, O2, operation-begin>

<T1, Z, 45, 70>

← T1 Rollback begins here

<T1, Z, 45> ← redo-only log record during physical undo (of incomplete O2)

<T1, Y, .., ..> ← Normal redo records for logical undo of O1

…

<T1, O1, operation-abort> ← What if crash occurred immediately after this?

<T1, abort>

Recovery Algorithm with Logical Undo

Basically same as earlier algorithm, except for changes described earlier for transaction rollback

1. (**Redo phase**): Scan log forward from last < **checkpoint** *L*> record till end of log
    1. **Repeat history** by physically redoing all updates of all transactions,
    2. Create an undo-list during the scan as follows
        - *undo-list* is set to *L* initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

    This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

    Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

Recovery with Logical Undo (Cont.)

Recovery from system crash (cont.)

2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*.
   - Log records of transactions being rolled back are processed as described earlier, as they are found
     - Single shared scan for all transactions being undone
   - When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
   - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*
- This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

ARIES Recovery Algorithm

ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The recovery algorithm we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the recovery algorithm described earlier, ARIES
  1. Uses **log sequence number (LSN)** to identify log records
     - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
     - More coming up on each of the above …

ARIES Optimizations

- **Physiological redo**
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - Physiological redo can log just the record deletion
        - Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - But extra actions are required for recovery
        - Treated as a media failure

ARIES Data Structures

- ARIES uses several data structures
  - Log sequence number (LSN) identifies each log record
    - Must be sequentially increasing
    - Typically an offset from beginning of log file to allow fast access
      - Easily extended to handle multiple log files
  - Page LSN
  - Log records of several different types
  - Dirty page table

ARIES Data Structures: Page LSN

- Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the page, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page
  - To flush page to disk, must first S-latch page
    - Thus page state on disk is operation consistent
      - Required to support physiological redo
  - PageLSN is used during recovery to prevent repeated redo
    - Thus ensuring idempotence

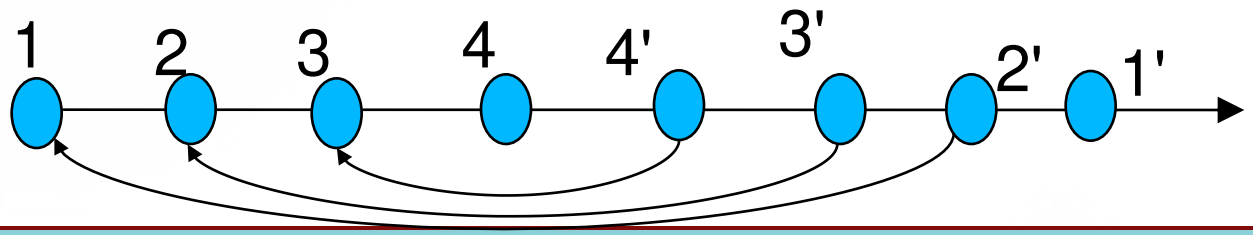
ARIES Data Structures: Log Record

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

  - LSN in log record may be implicit
- Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
  - Serves the role of operation-abort log records used in earlier recovery algorithm
  - Has a field UndoNextLSN to note next (earlier) record to be undone
    - Records in between would have already been undone
    - Required to avoid repeated undo of already undone actions

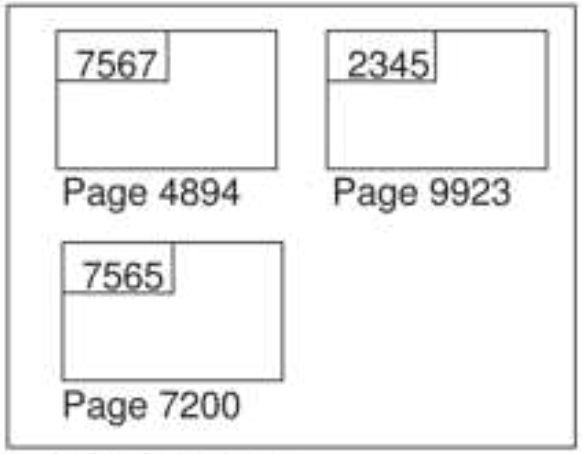
LSN	TransID	UndoNextLSN	RedoInfo



ARIES Data Structures: DirtyPage Table

- **DirtyPageTable**
  - List of pages in the buffer that have been updated
  - Contains, for each such page
    - **PageLSN** of the page
    - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
      - Set to current end of log when a page is inserted into dirty page table (just before being updated)
      - Recorded in checkpoints, helps to minimize redo work

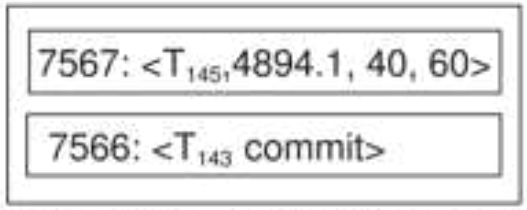
ARIES Data Structures



Database Buffer

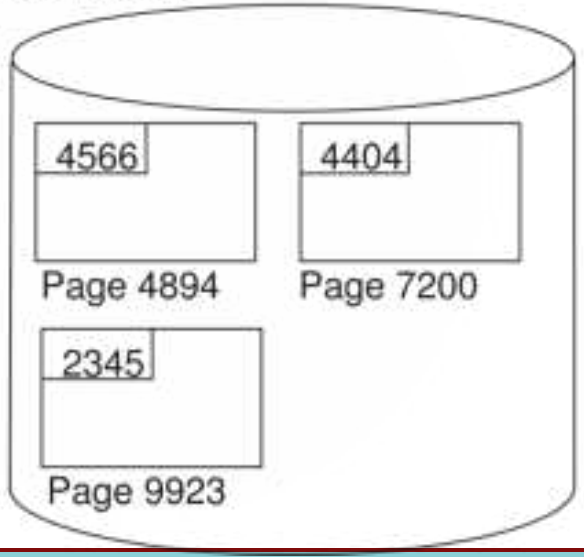
PageID	PageLSN	RecLSN
4894	7567	7564
7200	7565	7565

Dirty Page Table

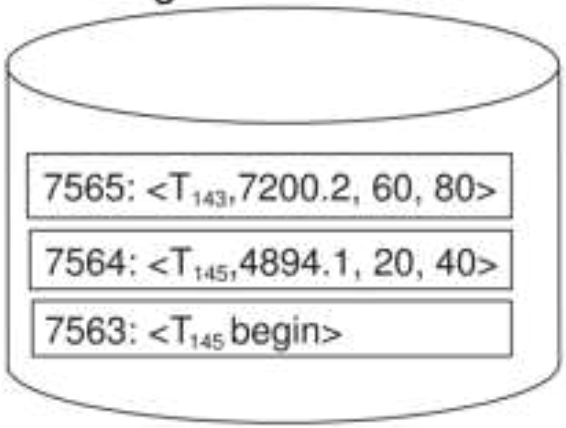


Log Buffer (PrevLSN and UndoNextLSN fields not shown)

Stable data



Stable log



ARIES Data Structures: Checkpoint Log

- **Checkpoint log record**
  - Contains:
    - DirtyPageTable and list of active transactions
    - For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time
    - Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead
  - can be done frequently

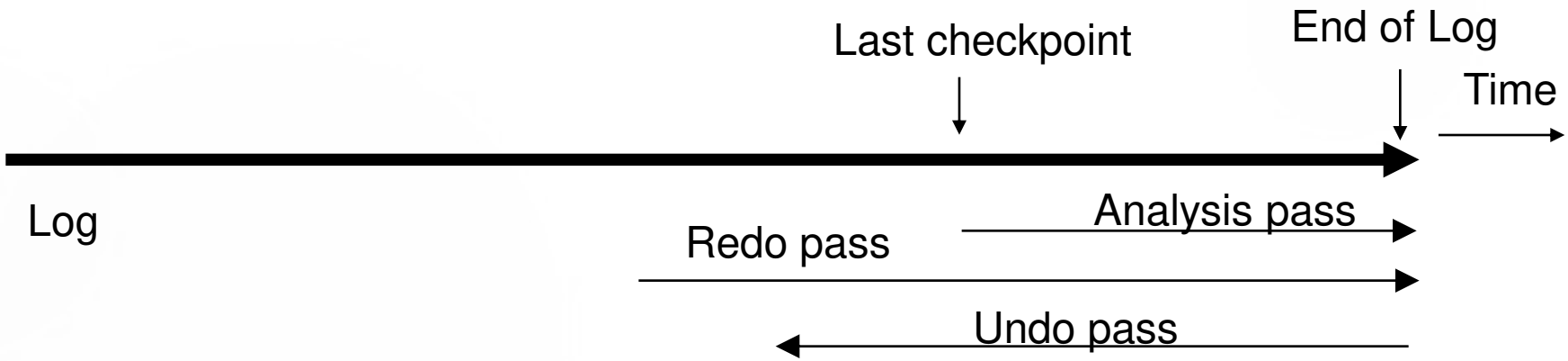
ARIES Recovery Algorithm

ARIES recovery involves three passes

- Analysis pass: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- Redo pass:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- Undo pass:
  - Rolls back all incomplete transactions
    - Transactions whose abort was complete earlier are not undone
      - Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required

Aries Recovery: 3 Passes

- Analysis, redo and undo passes
- Analysis determines where redo should start
- Undo has to go back till start of earliest incomplete transaction



ARIES Recovery: Analysis

Analysis pass

- Starts from last complete checkpoint log record
  - Reads DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in undo-list from checkpoint log record
- Scans forward from checkpoint
- .. Cont. on next page …

ARIES Recovery: Analysis (Cont.)

Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to undo-list
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo
- At end of analysis pass:
  - RedoLSN determines where to start redo pass
  - RecLSN for each page in DirtyPageTable used to minimize redo work
  - All transactions in undo-list need to be rolled back

ARIES Redo Pass

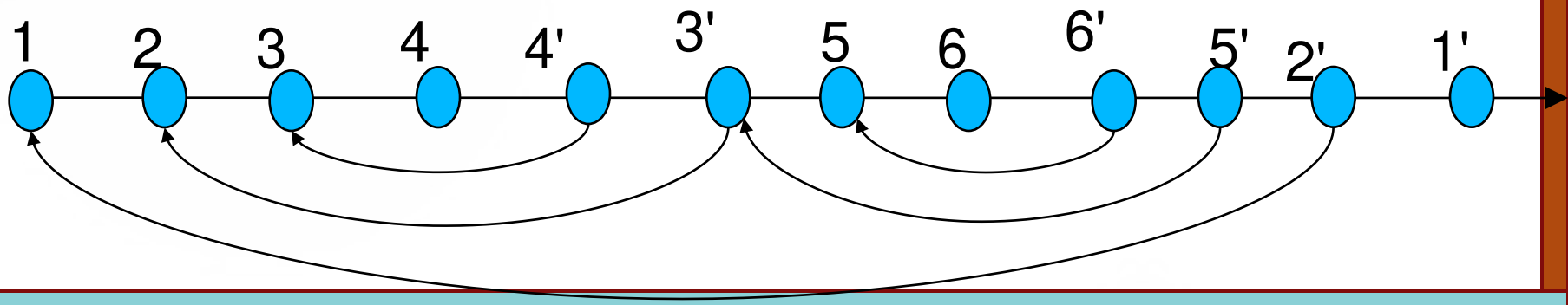
Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

  NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk!

ARIES Undo Actions

- When an undo is performed for an update log record
  - Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).
    - CLR for record *n* noted as *n*' in figure below
  - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
    - Arrows indicate UndoNextLSN value
- ARIES supports partial rollback
  - Used e.g. to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback

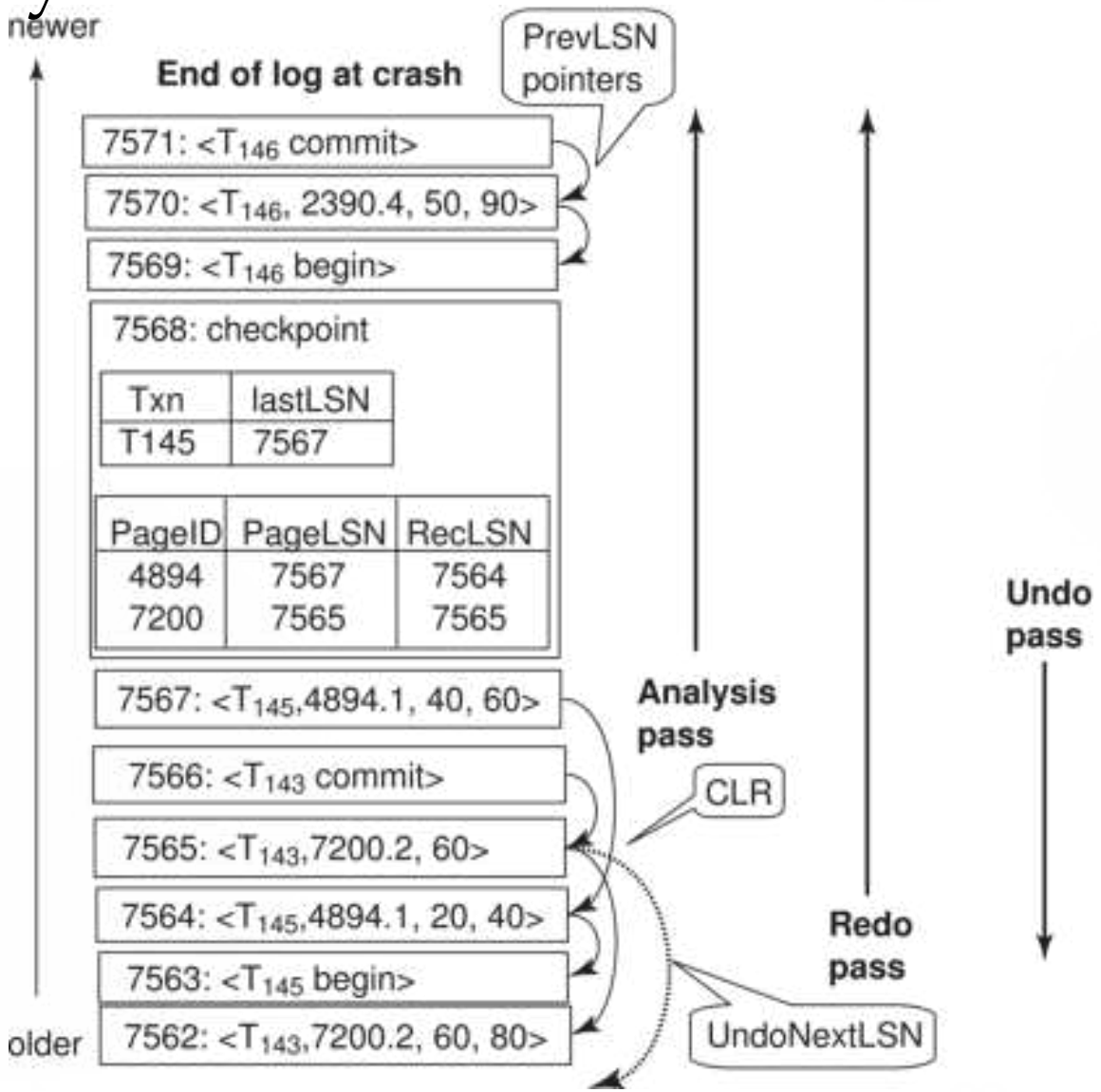


ARIES: Undo Pass

Undo pass:

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
      - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
        - All intervening records are skipped since they would have been undone already
- Undos performed as described earlier

Recovery Actions in ARIES



Other ARIES Features

- Recovery Independence
  - Pages can be recovered independently of others
    - E.g. if some disk pages fail they can be recovered from a backup while other pages are being used
- Savepoints:
  - Transactions can record savepoints and roll back to a savepoint
    - Useful for complex transactions
    - Also used to rollback just enough to release locks on deadlock

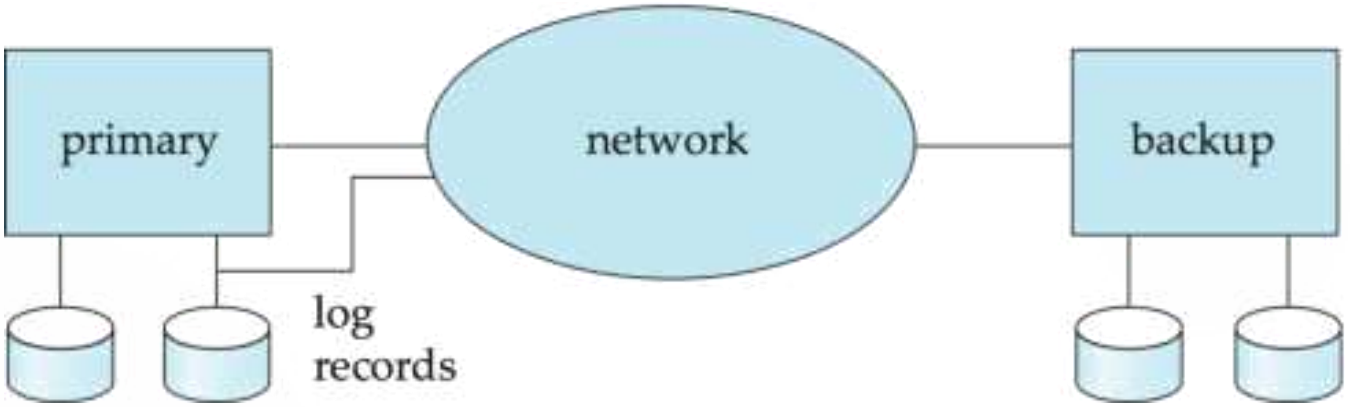
Other ARIES Features (Cont.)

- Fine-grained locking:
  - Index concurrency algorithms that permit tuple level locking on indices can be used
    - These require logical undo, rather than physical undo, as in earlier recovery algorithm
- Recovery optimizations: For example:
  - Dirty page table can be used to prefetch pages during redo
  - Out of order redo is possible:
    - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
    - Meanwhile other log records can continue to be processed

Remote Backup Systems

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe:** commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

End of Recovery Systems (Unit-IV)