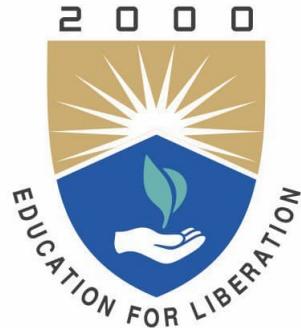


INSTITUTE OF AERONAUTICAL ENGINEERING
(AUTONOMOUS)
Dundigal, Hyderabad - 500 043



LECTURE NOTES:

DATABASE MANAGEMENT SYSTEMS(AITC05)

DRAFTED BY :
A. RAJITHA (IARE 10767)
Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
INSTITUTE OF AERONAUTICAL ENGINEERING

November 1, 2021

Contents

Contents	1
List of Figures	4
1 CONCEPTUAL MODELING	1
1.1 Introduction	1
1.1.1 Data models:	3
1.1.2 Data Base Languages:	4
1.1.2.1 Data-Manipulation Language	4
1.1.2.2 Data- Definition Language (DDL):	4
1.1.2.3 Data Base Access from Application Programs:	4
1.1.3 Transaction Management:	5
1.1.4 The Query Processor	5
1.2 Database Architecture:	6
1.2.1 Database Application Architectures:	6
1.3 Database Users and Administrators:	8
1.3.1 Data base Users and User Interfaces	8
1.3.2 Database Administrator	9
1.4 History of Database Systems:	9
1.5 Introduction to Database Design:	10
1.5.1 Data Base Design	10
1.5.2 Attributes	11
1.5.2.1 Types of attributes	11
1.6 Relationships and Relationship Sets	13
1.7 Key Constraints	15
1.8 Conceptual Design with ER Model	18
1.8.1 Conceptual Design for Large Enterprises	18
1.9 Relational Model	19
1.10 Tegrity Constraints over Relations	20
2 RELATIONAL ALGEBRA AND RELATIONAL CALCULUS	22
2.1 Introduction	22
2.1.1 Selection:	23
2.2 Set Operations	23
2.3 Examples of Algebra Queries	25
2.3.1 1.Find names of sailors who've reserved boat #103	25

2.3.2	2. Find Sailors Who've Reserved a Red or a Green Boat	26
2.3.3	Expressive Power of Relational Algebra And Calculus	27
2.4	Introduction to Nested Queries	28
2.4.1	Correlated Nested Queries	28
2.4.2	The Group By and Having Clauses	29
2.4.3	Null Values	30
2.4.4	Logical Connectives-And, or and Not	31
2.5	Outer joins	33
2.6	Complex Integrity Constraints in SQL	34
2.6.1	1. Constraints over a Single Table	34
2.6.2	Domain Constraints	35
2.6.3	3. Assertions: ICs over Several Tables	35
3	INTRODUCTION TO SCHEMA REFINEMENT	36
3.1	Introduction	36
3.1.1	Problem Related to Decomposition	37
3.1.2	Armstrong's Axioms	38
3.2	Trivial Functional Dependency	38
3.2.1	Reasoning about FDS	38
3.3	Normal forms	39
3.3.1	First normal form	39
3.3.2	Second normal form	40
3.3.3	Third normal form	40
3.4	Properties of Decompositions	42
3.4.1	Lossless Join Decompositions:	42
3.4.2	Dependency Preserving Decomposition (Intuitive)	43
3.5	Schema Refinement in Data Base Design:	44
3.5.1	Forth normal form	45
3.5.2	Inclusion Dependencies	46
4	TRANSACTION CONCEPT	47
4.1	Introduction	47
4.2	ACID Properties	49
4.3	Transaction State	49
4.4	Implementation of Atomicity And Durability	50
4.5	Serializability	52
4.5.1	Conflict Serializability	53
4.6	View Serializability	53
4.7	Recoverability	54
4.7.1	Cascading Rollbacks	54
5	CONCURRENCY CONTROL	56
5.1	Introduction	56
5.1.1	Testing For Serializability	57
5.2	Concurrency Control	58
5.3	Lock Based Protocols	59

5.4	Two-Phase Locking Protocol	60
5.4.1	Time stamp Based Protocols	61
5.5	Correctness of Timestamp-Ordering Protocol	61
5.5.1	Validation Based Protocol	62
5.6	Multiple Granularities	63
5.7	Recovery System	65
5.7.1	Recovery and Atomicity	65
5.8	Recovery with concurrent transactions	67
5.8.1	Log Record Buffering	67
5.8.2	Buffer Management	68
5.9	Advanced Recovery Techniques	68
5.10	Remote Backup Systems	70

List of Figures

1.1	The Three level of abstraction	3
1.2	The Query Processor	6
1.3	Overall System Structure	7
1.4	Two-tier and Three tier architecture	8
1.5	E-R diagrams	11
1.6	Simple attribute:	12
1.7	Composite attribute:	12
1.8	Multivalued attributes	13
1.9	One-to-one relation	14
1.10	One-to-many	14
1.11	Many-to-one	14
1.12	Many-to-many:	15
1.13	Key constraints:	15
1.14	Participation Constraints	16
1.15	Mapping Weak Entity Sets	16
1.16	Mapping hierarchical entities	17
1.17	Generalization	17
1.18	Specialization	18
1.19	Aggregation	18
1.20	An Instance of the S1 of the Students Relation	20
1.21	General Constraints	21
2.1	Relational algebra includes operator to select rows from a relation	23
2.2	Projection	23
2.3	Cross-Product	24
2.4	Cross-Product	24
2.5	Equijoin	25
2.6	Natural join	26
2.7	domain variable	27
2.8	SQL query	28
2.9	reserved boat	28
2.10	reserved a red boat.	28
2.11	reserved a red boat103.	29
2.12	reserved a red boat103.	29
2.13	reserved a red boat103.	29
2.14	The attribute list	30

2.15 sailors	31
2.16 Construct	32
2.17 Some Construct	32
2.18 all Construct	33
2.19 exists Construct	33
2.20 Joins	33
2.21 Relation loan	34
2.22 Joined Relations	34
2.23 Joined Relations1	34
2.24 Joined Relations1	34
3.1 Refinements	37
3.2 Example First Normal Form	39
3.3 Refinements	40
3.4 example of student relation	41
3.5 More on Lossless Join	42
3.6 Identifying Attributes of Entities	44
3.7 Dependencies	45
4.1 Transaction State	49
4.2 shadow database	50
4.3 Schedule1	51
4.4 Schedule2	51
4.5 Schedule3	51
4.6 Conflict serializability	52
4.7 View serializability	52
4.8 conflict serializable	53
4.9 reads and writes	53
4.10 reads and writes	54
4.11 Recoverable schedule	54
4.12 Cascadeless schedules	55
5.1 Implementation of Isolation	57
5.2 Testing For Serializability	57
5.3 Testing For Serializability	58
5.4 Conflict Serializability	58
5.5 Lock compatibility matrix	59
5.6 Use of the protocol	62
5.7 Timestamp-Ordering Protocol	62
5.8 Validation Based Protocol	63
5.9 Multiple Granularities	64
5.10 Shared and intention-exclusive	64
5.11 Shared and intention-exclusive	67
5.12 Fuzzy Check pointing	70
5.13 Remote Backup Systems	70

Chapter 1

CONCEPTUAL MODELING

Course Outcomes

After successful completion of this module, students should be able to:

CO 1	Define database, characteristics, functions of database management system and types of users to describe large sets of data.	Remember
CO 2	Compare traditional File Processing System and a Database System for constructing a database.	Understand
CO 3	Describe data models, schemas, instances, view levels and database architecture for voluminous data storage.	Remember
CO 4	Model the real world database systems using Entity Relationship Diagrams from the requirement specification.	Apply

1.1 Introduction

- **Data:** data is a collection of raw facts and figures.
- **Database:** database is a collection of interrelated data.
- **DBMS:** database is a collection of interrelated data and set of programs can access that system.

DATABASE SYSTEM APPLICATIONS:

1. Enter Price Information:
 - Sales: customers, products, purchases
 - Accounting: payments, receipts, account balance, assets.
 - Human Resources: employee records, salaries, tax deductions
 - Manufacturing: production, inventory, orders, supply chain
 - Online Retail: order tracking, customized recommendations
2. Banking and Finance: all transactions

- Credit card Transaction: generation of monthly statements.
- Finance: storing information about holdings and sales,
- 3. Universities: registration, grades
- 4. Airlines: reservations, schedules
- 5. Telecommunications: keeping records of calls made, generating monthly bills

PURPOSE OF DATABASE SYSTEMS:

In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data:

- Data redundancy and inconsistency: Multiple file formats, duplication of information in different files
- Difficulty in accessing data: Need to write a new program to carry out each new task
- Data isolation: multiple files and formats
- Integrity problems: Hard to add new constraints or change existing ones
- Atomicity of updates: Failures may leave database in an inconsistent state with partial updates carried out. Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access anomalies: Example: Two people reading a balance and updating it at the same time

VIEW OF DATA:

A database is a collection of interrelated data and 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 stored and maintained.

Data Abstraction:

Major purpose of DBMS is to provide users with abstract view of data i.e. the system hides certain details of how the data are stored and maintained. Since database system users are not computer trained, developers hide the complexity from users through 3 levels of abstraction, to simplify user's interaction with the system.

Levels of Abstraction

- Physical level of data abstraction: How the data are actually stored. This is the lowest level of abstraction which describes how data are actually stored.
- Logical level of data abstraction: This level hides what data are actually stored in the database and what relationship exists among them. Describes data stored in database, and the relationships among the data.
- View Level of data abstraction: View provides security mechanism to prevent user from accessing certain parts of database. Application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

Instances and schemas:

Instance: The collection of information stored in the database at a particular moment is called an instance of the database.

Similar to types and variables in programming languages **Schema:** the overall design of the database is called the database schema.

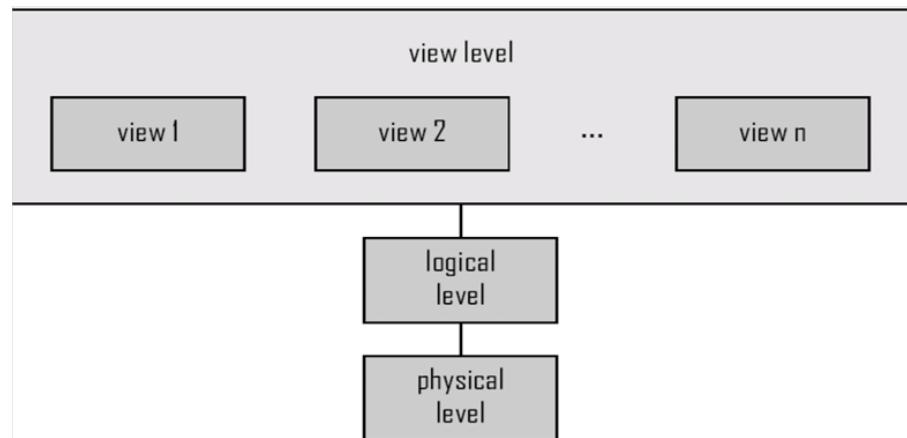


FIGURE 1.1: The Three level of abstraction

Example: The database consists of information about a set of customers and accounts and the relationship between them .Analogous to type information of a variable in a program

- Physical schema: database design at the physical level.
- Logical schema: database design at the logical level.

1.1.1 Data models:

Data Model: 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.

- 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 called known as relations.
- 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 object

- Object-Based Data Models: Object-oriented Programming (especially in Java, C++, or C#).
- 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.
- Other older models:
 1. Network Model
 2. Hierarchical Model

1.1.2 Data Base Languages:

A Database provides a DDL to specify the database schema and a DML to express database queries and updates.

1.1.2.1 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 non procedural DMLs) require user to specify what data are needed without specifying how to get those data.

1.1.2.2 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. SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc...

Example: create table account (

account number char(10),

branch name char(10),

balance integer)

In addition, the DDL statement updates the data dictionary, which contains metadata; the schema of a table is an example of metadata.

1.1.2.3 Data Base Access from Application Programs:

Application programs are programs that are used to interact with the database.

To access the database, DML Statements need to be executed form the host language. There are two ways o do this.

- By Providing an Application Program interface (set of procedures) that can be used to send DML and DDL statement to the database and retrieve the results.(ODBC and JDBC).

- By extending the host language syntax to embed DML calls within the host language program. A special character prefaces DML calls and preprocessor called the DML pre compiler ,converts the DML statements to normal procedure calls in the host language SQL: widely used non-procedural language

Example 1:

Find the name of the customer with customer-id 192-83-7465 select customer.customer name from customer

where customer.customer id = ‘192-83-7465’

Example 2:

Find the balances of all accounts held by the customer with customer-Id 192-83- 7465.

select from where

account.balance depositor, account

depositor.customer id = ‘192-83-7465’ and

depositor.account number = account.account number

Example 3:

select from where

Find the name of the customer

customer.customer name customer

customer.customer id = ‘192-83-7465’

1.1.3 Transaction Management:

- A transaction is a collection of operations that performs a single logical function in a Database application.
- Transaction-management component ensures that the database remains in a Consistent (correct) state despite system failures (e.g., power failures and operating System crashes) and transaction failures.
- Concurrency-control manager controls the interaction among the concurrent Transactions to ensure the consistency of the database.

1.1.4 The 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.

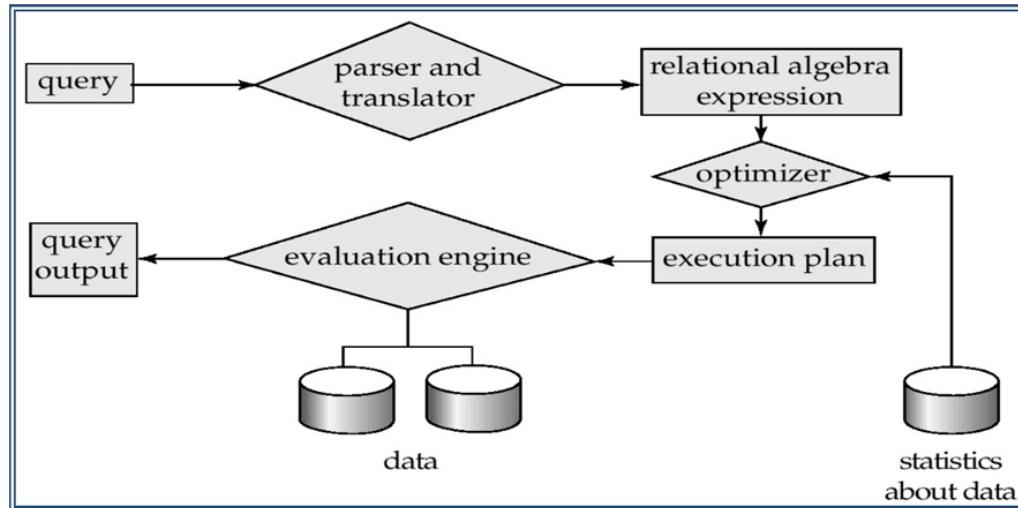


FIGURE 1.2: The Query Processor

1.2 Database Architecture:

The architecture of database systems is greatly influenced by the underlying computer system on which the database is runs:

Database system can be.

- Client-server
- Parallel (multiple processors and disks)
- Distributed

1.2.1 Database Application Architectures:

- Database applications are usually partitioned into two or three parts, as in Figure 1.8.1.. 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.

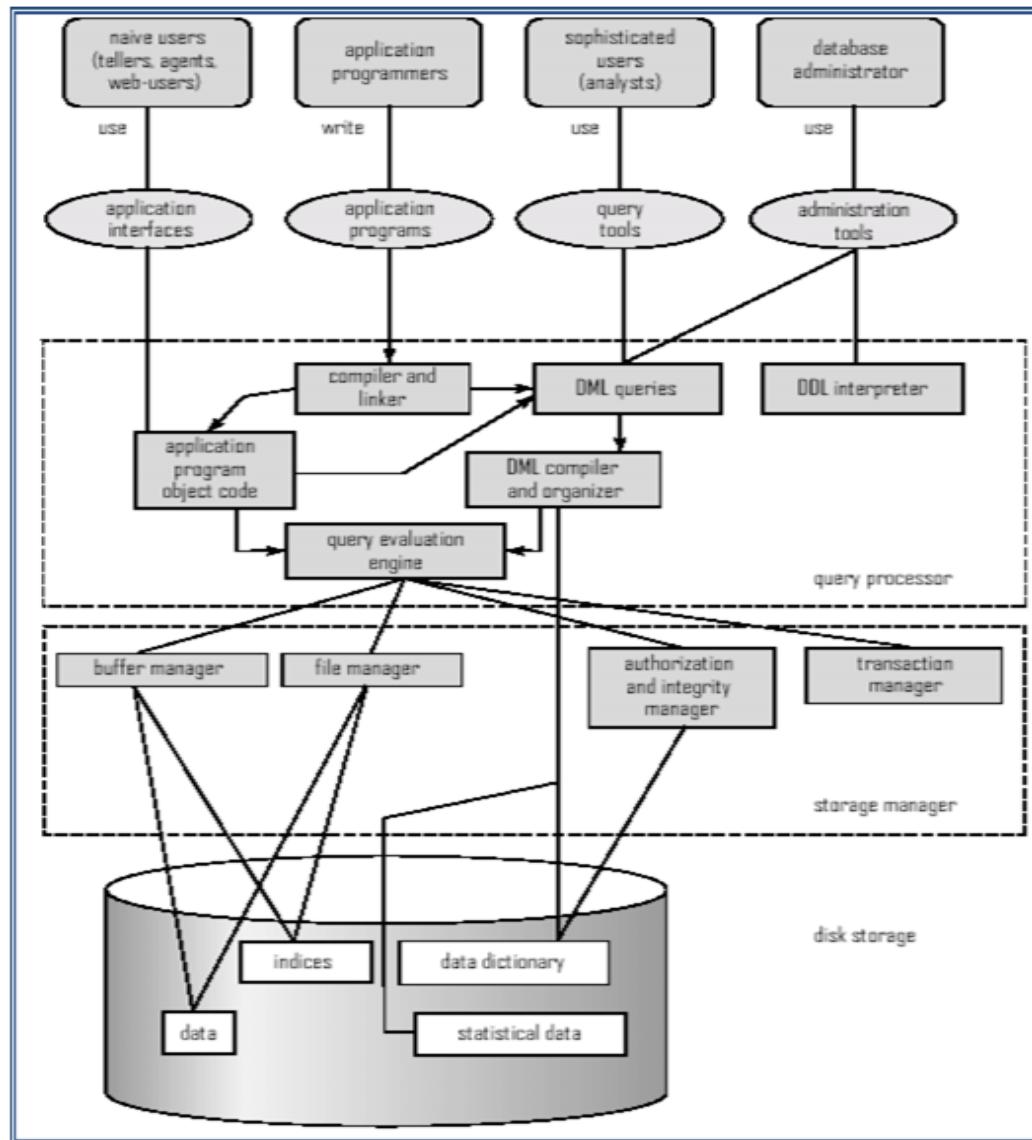


FIGURE 1.3: Overall System Structure

- 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 World Wide Web.

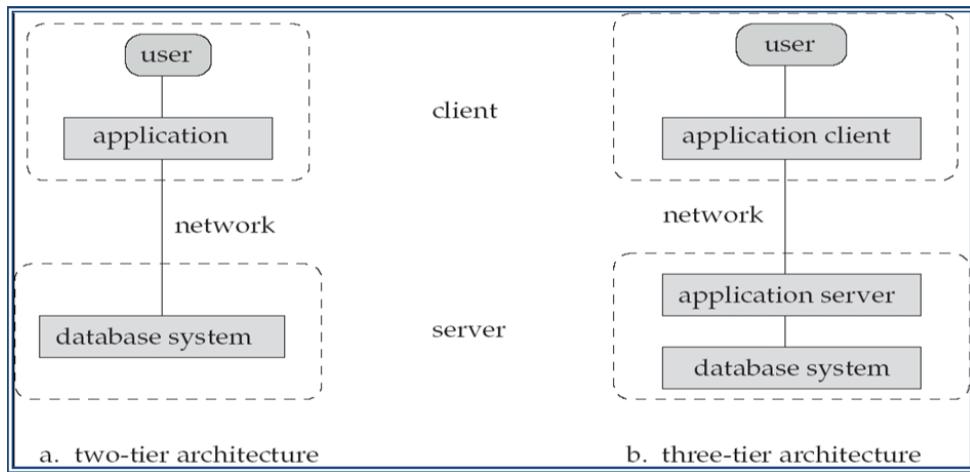


FIGURE 1.4: Two-tier and Three tier architecture

1.3 Database Users and Administrators:

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

1.3.1 Data base 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.

- **Naïve users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a clerk in the university who needs to add a new instructor to Users are differentiated by the way they expect to interact with the system department A invokes a program called New - hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), and the salary
- **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 with minimal programming effort.
- **Sophisticated users:** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
- **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.

1.3.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a data base administrator (DBA).

The functions of a DBA include:

- Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- Storage structure and access-method definition.
- Schema and physical-organization modification.

1 Routine maintenance.

2 Periodically backing up the database.

3 Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

– Monitoring jobs running on the Data base.

1.4 History of Database Systems:

- 1950s and early 1960s: Data processing using magnetic tapes for storage Tapes provide only sequential access Punched cards for input
- Late 1960s and 1970s: Hard disks allow direct access to data Network and hierarchical data models in widespread use Ted Codd defines the relational data model Would win the ACM Turing Award for this work IBM Research begins System R prototype UC Berkeley begins Ingres prototype High-performance (for the era) transaction processing
- 1980s: Research relational prototypes evolve into commercial systems SQL becomes industry standard Parallel and distributed database systems Object-oriented database systems
- 1990s: Large decision support and data-mining applications Large multi-terabyte data warehouses Emergence of Web commerce
- 2000s: XML and XQuery standards Automated database administration Increasing use of highly parallel database systems Web-scale distributed data storage systems

EXERCISES 1. List four applications you have used that most likely employed a database system to store persistent data.

2. List four significant differences between a file-processing system and a DBMS.

3. Explain the concept of physical data independence and its importance in database systems.

4. List five responsibilities of a database-management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.
5. What are the five main functions of a database administrator?
6. Explain the difference between two-tier and three-tier architectures. Which is better suited for Web applications? Why?

1.5 Introduction to Database Design:

The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

1.5.1 Data Base Design

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

(1) Requirements Analysis: The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.

(2) Conceptual Database Design: The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

(3) Logical Database Design: We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. The result is a conceptual schema, sometimes called the logical schema, in the relational data model.

Beyond ER Design The ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis. Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described below:

(4) Schema Refinement: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems,

(5) Physical Database Design: In this step we must consider typical expected workloads that our

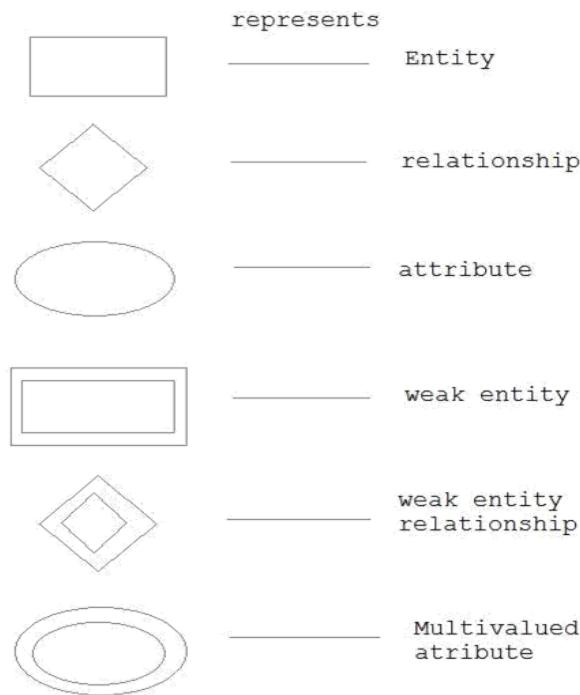


FIGURE 1.5: E-R diagrams

database must support and further refine the database design to ensure that it meets desired performance criteria

(6) Security Design: In this step, we identify different user groups and different roles played by various users (e.g., the development team for a product, the customer support representatives, the product manager).

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

1.5.2 Attributes

Entities are represented by means of their properties, called attributes. All attributes have values. For example, a student entity may have name, class, age as attributes.

There exist a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

1.5.2.1 Types of attributes

Simple attribute: Simple attributes are atomic values, which cannot be divided further. For example, student's phone-number is an atomic value of 10 digits.

Attributes are properties of entities. Attributes are represented by means of eclipses. Every eclipse

represents one attribute and is directly connected to its entity (rectangle).

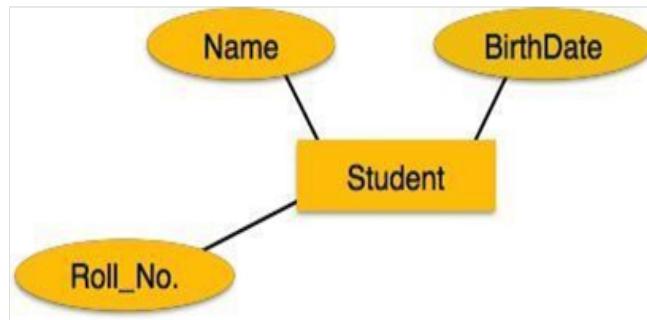


FIGURE 1.6: Simple attribute:

Composite attribute:

Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first name and last name.

If the attributes are composite, they are further divided in a tree like structure. Every node is then connected to its attribute. That is composite attributes are represented by eclipses that are connected with an eclipse

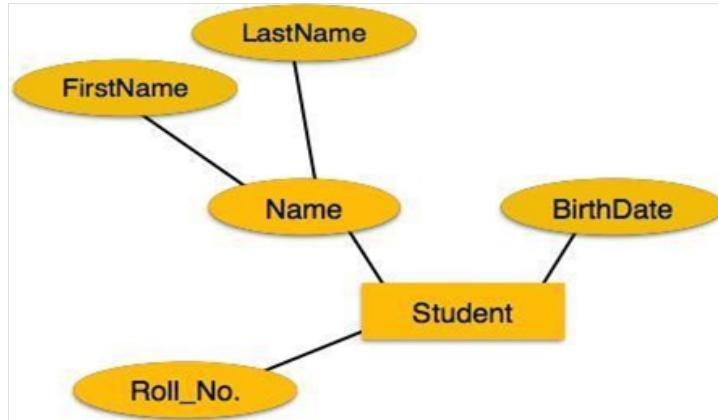


FIGURE 1.7: Composite attribute:

Single-valued attribute:

Single valued attributes contain on single value. For example: Social Security Number.

Multi-value attribute:

Multi-value attribute may contain more than one values. For example, a person can have more than one phone numbers, email addresses etc

Multivalued attributes are depicted by double eclipse

Derived attribute:

Derived attributes are attributes, which do not exist physical in the database, but their values are

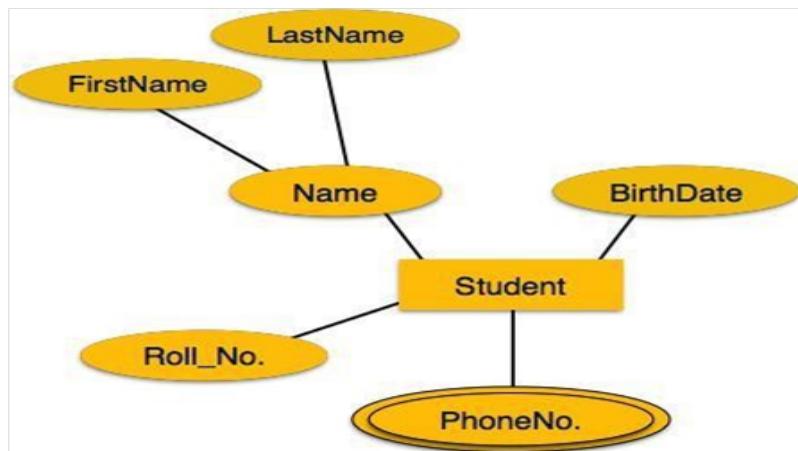


FIGURE 1.8: Multivalued attributes

derived from other attributes presented in the database. For example, average salary in a department should be saved in database instead it can be derived. For another example, age can be derived from data of birth

1.6 Relationships and Relationship Sets

The association among entities is called relationship. For example, employee entity has relation work

s at with department. Another example is for student who enrolls in some course. Here, Works at and Enrolls are called relationship.

Relationship Set:

Relationship of similar type is called relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

Degree of relationship

The number of participating entities in a relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

Mapping Cardinalities:

Cardinality defines the number of entities in one entity set which can be associated to the number of entities of other set via relationship set

• **One-to-one:** one entity from entity set A can be associated with at most one entity of entity set B and vice versa.

• **One-to-many:** One entity from entity set A can be associated with more than one entities of entity set B but from entity set B one entity can be associated with at most one entity.

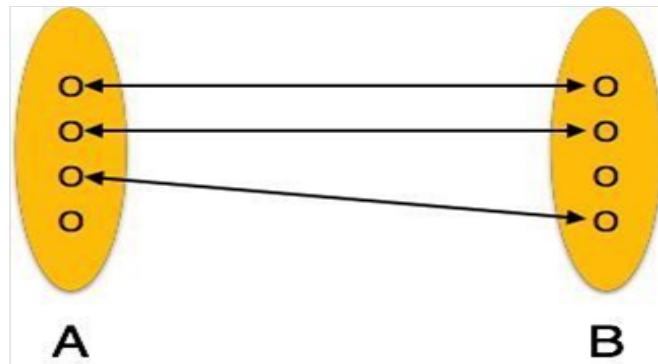


FIGURE 1.9: One-to-one relation

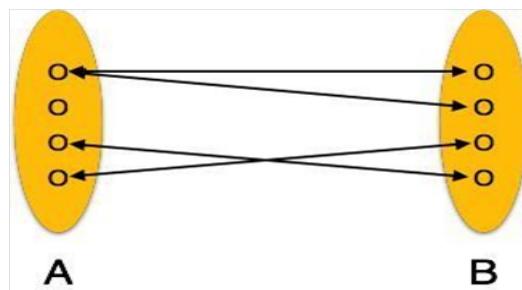


FIGURE 1.10: One-to-many

- **Many-to-one:** More than one entities from entity set A can be associated with at most one entity of entity set B but one entity from entity set B can be associated with more than one entity from entity set A.

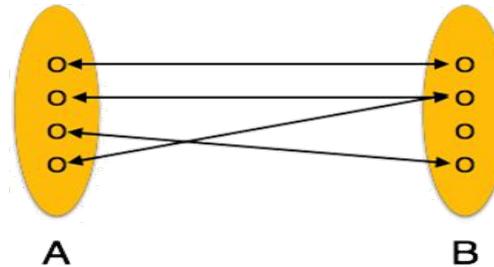


FIGURE 1.11: Many-to-one

- **Many-to-many:** one entity from A can be associated with more than one entity from B and vice versa.
- An entity is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy department.
- A attribute is a property of an entity.

ADDITIONAL FEATURES OF ER MODEL

- 1 Key Constraints
- 2 Participation Constraints

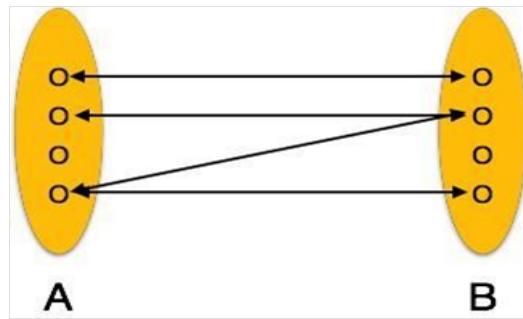


FIGURE 1.12: Many-to-many:

- 3 Weak Entities
- 4 Class Hierarchies
- 5 Aggregation

1.7 Key Constraints

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called key for that relation. If there are more than one such minimal subsets, these are called candidate keys. Key constraints forces that:

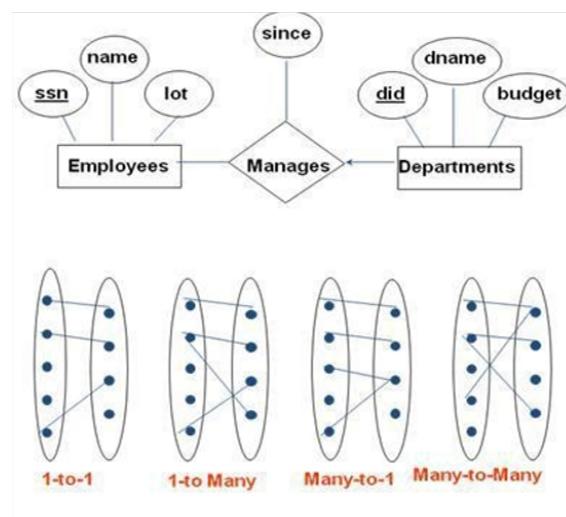


FIGURE 1.13: Key constraints:

- in a relation with a key attribute, no two tuples can have identical value for key attributes.
- key attribute can not have NULL values.

Key constraints are also referred to as Entity Constraints

WEAK ENTITY SETS

- * An entity set that does not have a primary key is referred to as a weak entity set.

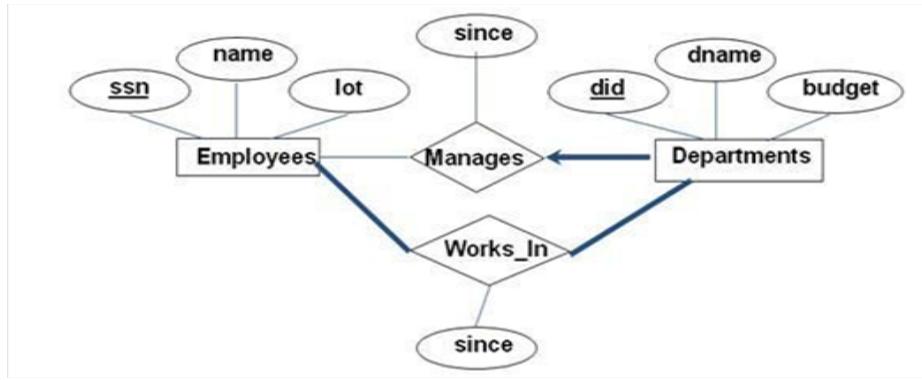


FIGURE 1.14: Participation Constraints

* The existence of a weak entity set depends on the existence of a identifying entity set it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set Identifying relationship depicted using a double diamond The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set. The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator depict a weak entity set by double rectangles.

Under line the discriminator of a weak entity set with a dashed line.

More Weak Entity Set Examples In a university, a course is a strong entity and a course offering can be modeled as a weak entity The discriminator of course offering would be semester (including year) and section number (if there is more than one section). If we model course offering as a strong entity we would model course number as an attribute. Then the relationship with course would be implicit in the course number attribute.

A weak entity sets is one which does not have any primary key associated with it. • Create table

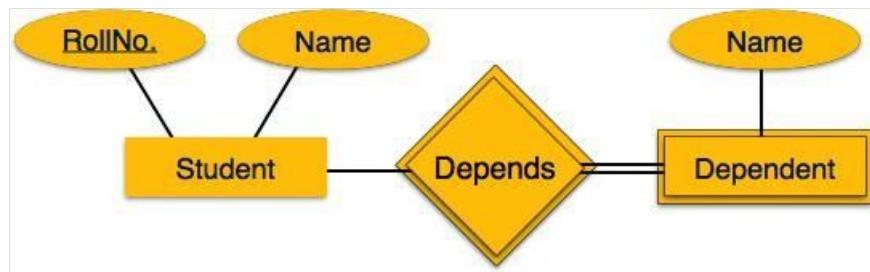


FIGURE 1.15: Mapping Weak Entity Sets

for weak entity set

- Add all its attributes to table as field
- Add the primary key of identifying entity set
- Declare all foreign key constraints

CLASS HIERARCHIES

- Classifying the entities in an entity set into sub classes

- ER specialization or generalization comes in the form of hierarchical entity sets.

Mapping process (Algorithm):

- Create tables for all higher level entities

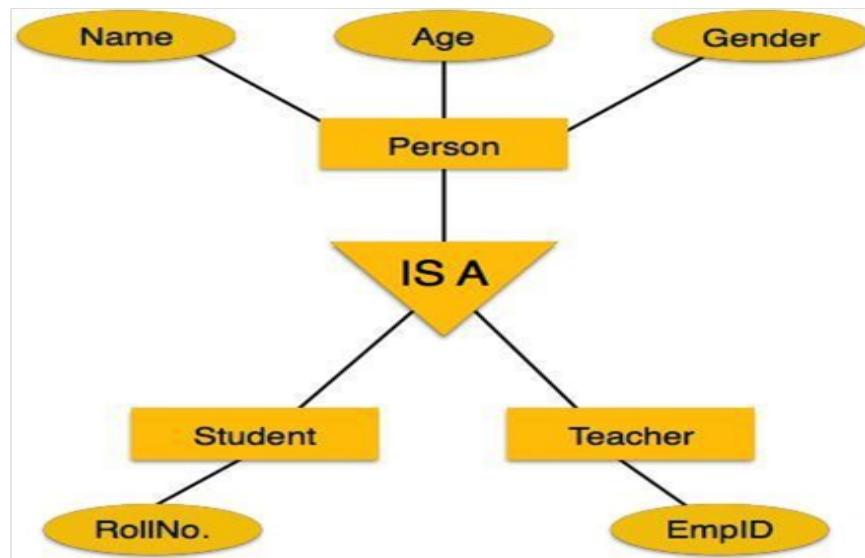


FIGURE 1.16: Mapping hierarchical entities

- Create tables for lower level entities
- Add primary keys of higher level entities in the table of lower level entities
- In lower level tables, add all other attributes of lower entities.
- Declare primary key of higher level table the primary key for lower level table
- Declare foreign key constraints

Generalization As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called Generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For an example, pigeon, house sparrow, crow and dove all can be generalized as Birds. **Specialization**

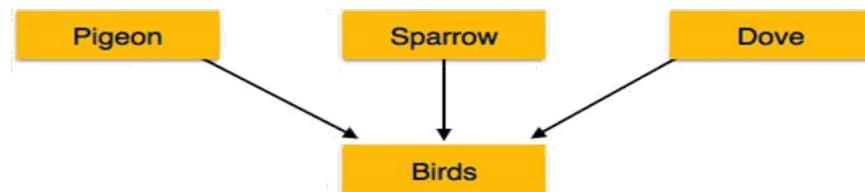


FIGURE 1.17: Generalization

Specialization is a process, which is opposite to generalization, as mentioned above. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group Person for example. A person has name, date of birth, gender etc. These properties are common in all persons, human beings. But in a company, a person can be identified as employee, employer, customer or vendor based on what role do they play in company.

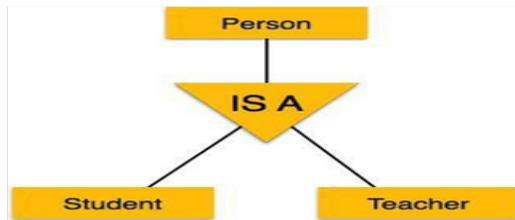


FIGURE 1.18: Specialization

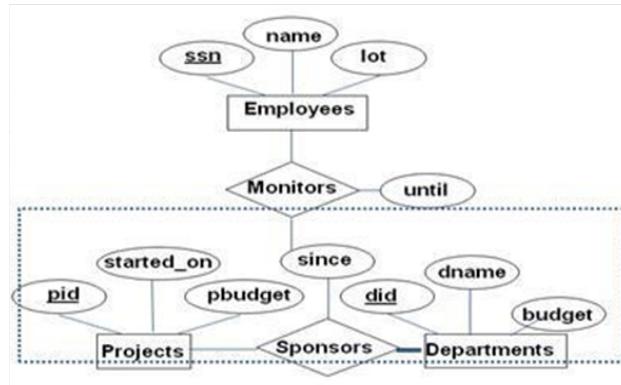


FIGURE 1.19: Aggregation

1.8 Conceptual Design with ER Model

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

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

Issues involved in making these choices.

- Entity versus Attribute
- Entity Versus Relationship
- Binary versus Ternary Relationship
- Aggregation versus Ternary Relationships

1.8.1 Conceptual Design for Large Enterprises

- The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams.
- For a large enterprise, the design may require the efforts of more than one designer and span data

and application code used by a number of user groups.

- Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.
- An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent.
- The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

1.9 Relational Model

Codd proposed the relational data model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database field and largely supplanted these earlier models.

Today, the relational model is by far the dominant data model and is the foundation for the leading DBMS products, including IBM's DB2 family, Microsoft's Access and SQL-Server, FoxBASE, and Paradox.

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

Introduction to the relational model The main construct for representing data in the relational model is a relation. A relation consists of a relation schema and a relation instance. The relation instance is a table, and the relation

schema describes the column heads for the table. We first describe the relation schema and then the relation instance.

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

We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

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

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

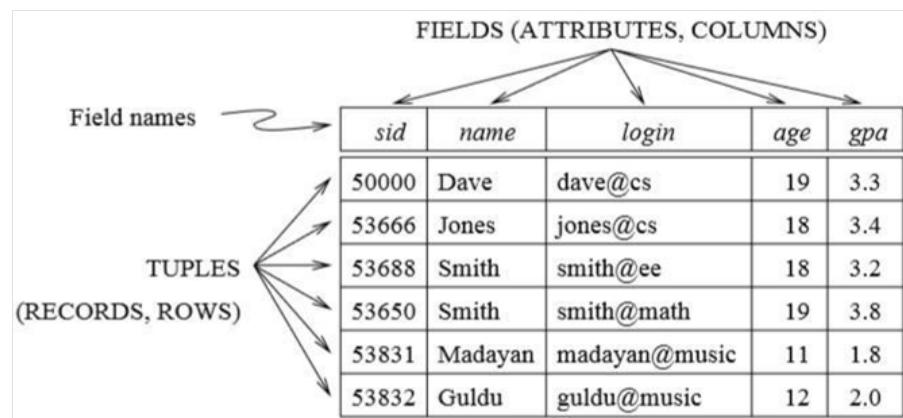


FIGURE 1.20: An Instance of the S1 of the Students Relation

1.10 Integrity Constraints over Relations

- IC: condition that must be true for any instance of the database; e.g., domain constraints.
- ICs are specified when schema is defined.
- ICs are checked when relations are modified.
- A legal instance of a relation is one that satisfies all specified ICs.
- DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.

Key constraints

- Candidate Key
- Primary Key
- Super Key

Foreign Key Constraints

- Specifying Key constraints in SQL General Constraints

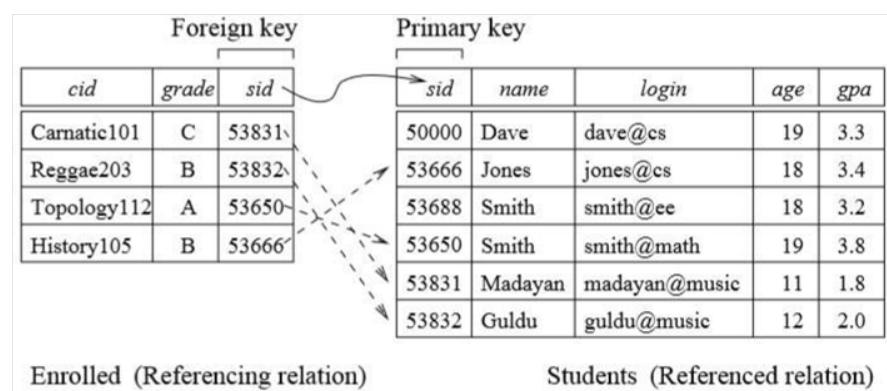


FIGURE 1.21: General Constraints

Chapter 2

RELATIONAL ALGEBRA AND RELATIONAL CALCULUS

Course Outcomes

After successful completion of this module, students should be able to:

CO 5	Define the relational data model, its constraints and keys to maintain integrity of data.	Remember
CO 6	Define the concept of Relational Algebra and Relational Calculus from set theory to represent queries.	Remember
CO 7	Build queries in Relational Algebra and Relational Calculus to retrieve desired information..	Apply

2.1 Introduction

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.

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.

2.1.1 Selection:

Relational algebra includes operator to select rows from a relation (σ). **Projection**

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

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

sname	rating
yuppy	9
rusty	10

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

FIGURE 2.1: Relational algebra includes operator to select rows from a relation

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

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

age
35.0
55.5

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

FIGURE 2.2: Projection

2.2 Set Operations

Cross-Product

- Each row of S1 is paired with each row of R1.
- Result schema has one field per field of S1 and R1, with field names inherited if possible.
- Conflict: Both S1 and R1 have a field called sid.

- All of these operations take two input relations, which must be union-compatible:
 - Same number of fields.
 - 'Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$			
sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 - S2$			
sid	sname	rating	age
22	dustin	7	45.0

$S1 \cap S2$			
sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

FIGURE 2.3: Cross-Product

- Each row of $S1$ is paired with each row of $R1$.
- *Result schema* has one field per field of $S1$ and $R1$, with field names 'inherited' if possible.
 - *Conflict*: Both $S1$ and $R1$ have a field called *sid*.

(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 2.4: Cross-Product

Renaming

Renaming operator :

$$\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 * R1)$$

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.

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.

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:

$$R \ltimes S = \sigma_c(R * S)$$

EQUIJOIN

A common special case of the join operation 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 $S1 \bowtie_{R.sid=S.sid} R1$. Notice that only one field called *sid* appears in the result.

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

$S1 \bowtie_{R.sid=S.sid} R1$

FIGURE 2.5: Equijoin

NATURAL JOIN

A further special case of the join operation $R ./ S$ 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 $S1 ./ R.sid=S.sid R1$ is actually a natural join and can simply be denoted as $S1 ./ R1$, since the only common field is sid. If the two relations have no attributes in common, $S1 ./ R1$ is simply the cross-product.

2.3 Examples of Algebra Queries

2.3.1 1.Find names of sailors who've reserved boat #103

Find names of sailors who've reserved a red boat Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname} = (\sigma_{bid=103} Reserves) \bowtie sailors$$

<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>sno</th><th>pno</th></tr> </thead> <tbody> <tr><td>s1</td><td>p1</td></tr> <tr><td>s1</td><td>p2</td></tr> <tr><td>s1</td><td>p3</td></tr> <tr><td>s1</td><td>p4</td></tr> <tr><td>s2</td><td>p1</td></tr> <tr><td>s2</td><td>p2</td></tr> <tr><td>s3</td><td>p2</td></tr> <tr><td>s4</td><td>p2</td></tr> <tr><td>s4</td><td>p4</td></tr> </tbody> </table> <p style="margin-top: 10px;">A</p>	sno	pno	s1	p1	s1	p2	s1	p3	s1	p4	s2	p1	s2	p2	s3	p2	s4	p2	s4	p4	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>pno</th></tr> </thead> <tbody> <tr><td>p2</td></tr> </tbody> </table> <p style="margin-top: 10px;">B</p>	pno	p2	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>pno</th></tr> </thead> <tbody> <tr><td>p2</td></tr> <tr><td>p4</td></tr> </tbody> </table> <p style="margin-top: 10px;">B2</p>	pno	p2	p4	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>pno</th></tr> </thead> <tbody> <tr><td>p1</td></tr> <tr><td>p2</td></tr> <tr><td>p4</td></tr> </tbody> </table> <p style="margin-top: 10px;">B3</p>	pno	p1	p2	p4
sno	pno																															
s1	p1																															
s1	p2																															
s1	p3																															
s1	p4																															
s2	p1																															
s2	p2																															
s3	p2																															
s4	p2																															
s4	p4																															
pno																																
p2																																
pno																																
p2																																
p4																																
pno																																
p1																																
p2																																
p4																																
<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>sno</th></tr> </thead> <tbody> <tr><td>s1</td></tr> <tr><td>s2</td></tr> <tr><td>s3</td></tr> <tr><td>s4</td></tr> </tbody> </table> <p style="margin-top: 10px;">A/B1</p>	sno	s1	s2	s3	s4	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>sno</th></tr> </thead> <tbody> <tr><td>s1</td></tr> <tr><td>s2</td></tr> <tr><td>s3</td></tr> <tr><td>s4</td></tr> </tbody> </table> <p style="margin-top: 10px;">A/B2</p>	sno	s1	s2	s3	s4	<table border="1" style="border-collapse: collapse; width: 100px; height: 100px;"> <thead> <tr><th>sno</th></tr> </thead> <tbody> <tr><td>s1</td></tr> <tr><td>s4</td></tr> </tbody> </table> <p style="margin-top: 10px;">A/B3</p>	sno	s1	s4																	
sno																																
s1																																
s2																																
s3																																
s4																																
sno																																
s1																																
s2																																
s3																																
s4																																
sno																																
s1																																
s4																																

FIGURE 2.6: Natural join

2.3.2 2. Find Sailors Who've Reserved a Red or a Green Boat

Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho(Temboats, (\sigma_{color=redBoats}) \cup ((\sigma_{color=greenBoats})) \pi_{sname}(Temboats \times Reserves \times Sailors)$$

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative.

Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC).

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 ;

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.

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 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 $>, <, =, \geq, \leq, 6=$ and let X and Y be domain variables. An atomic formula in DRC is one of the following:

2.3.3 Expressive Power of Relational Algebra And Calculus

It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called unsafe.

Example

It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.

Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

- $\langle x_1, x_2, \dots, x_n \rangle \in Rel$, where Rel is a relation with n attributes; each x_i , $1 \leq i \leq n$ is either a variable or a constant.
- $X \text{ op } Y$
- $X \text{ op } \text{constant}$, or $\text{constant op } X$

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
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

FIGURE 2.7: domain variable

THE FORM OF A BASIC SQL QUERIES

- 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.

- `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.

	<u>sid</u>	<u>bid</u>	<u>day</u>
R1	22	101	10/10/96
	58	103	11/12/96

	<u>sid</u>	<u>sname</u>	<u>rating</u>	<u>age</u>
S1	22	dustin	7	45.0
	31	lubber	8	55.5
	58	rusty	10	35.0

	<u>sid</u>	<u>sname</u>	<u>rating</u>	<u>age</u>
S2	28	yuppy	9	35.0
	31	lubber	8	55.5
	44	guppy	5	35.0
	58	rusty	10	35.0

FIGURE 2.8: SQL query

```

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

FIGURE 2.9: reserved 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' ) )
    
```

FIGURE 2.10: reserved a red boat.

2.4 Introduction to Nested Queries

2.4.1 Correlated Nested Queries

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).

(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 )
```

FIGURE 2.11: reserved a red boat103.

SET-COMPARISON OPERATORS

SQL also supports op ANY and op ALL, whereop is one of the arithmetic comparison operators

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' )
```

FIGURE 2.12: reserved a red boat103.

Find the sailors with the highest rating.

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

FIGURE 2.13: reserved a red boat103.

2.4.2 The Group By and Having Clauses

Motivation for Grouping

So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply

them to each of several groups of tuples.

- consider: Find the age of the youngest sailor for each rating level. In general, we don't know how many rating levels exist, and what the rating values for these levels are Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (i):
 - (i) attribute names
 - (ii)terms with aggregate operations (e.g., MIN (S.age)).

Find the sailors with the highest rating.

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

FIGURE 2.14: The attribute list

The attribute list (i) must be a subset of grouping-list. Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, ‘unnecessary’ fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group!
- In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age 18, for each rating with at least 2 such sailors

1. HAVING clause can also contain a subquery.

2. Aggregate operations cannot be nested! WRONG: Examples:

(Q1).Find the number of depositors for each branch.

(Q2).Find the names of all branches where the average account balance is more than 1,200.

2.4.3 Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse's name).

Sailors instance:

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
96	frodo	3	25.5

*SELECT S.rating, MIN(S.age)
AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1*

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

FIGURE 2.15: sailors

- SQL provides a special value null for such situations. The presence of null complicates many issues. E.g.: Special operators needed to check if value is/is not null. Is rating \neq 8 true or false when rating is equal to null?

COMPARISON USING NULL VALUES

It is possible for tuples to have a null value, denoted by null, for some of their attributes. null signifies an unknown value or that a value does not exist. The predicate is null can be used to check for null values.

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.

Example: Find all loan number which appear in the loan relation with null values for amount.

select loan number

from loan

where amount is null

The result of any arithmetic expression involving null is null

Example: $5 + \text{null}$ returns null However, aggregate functions simply ignore nulls

Any comparison with null returns unknown

2.4.4 Logical Connectives-And, or and Not

Three-valued logic using the truth value unknown:

OR:

(unknown or true) = true, (unknown or false) = unknown (unknown or unknown) = unknown

AND:

(true and unknown) = unknown, (false and unknown) = false,

(unknown and unknown) = unknown

NOT:

(not unknown) = unknown

P is unknown evaluates to true if predicate P evaluates to unknown

- All aggregate operations except count(*) ignore tuples with null values on the aggregated attributes.

IMPACT ON SQL CONSTRUCTS In

Construct some Construct

Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
  from borrower
 where customer_name in (select customer_name
                           from depositor)
```

■ Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
  from borrower
 where customer_name not in (select customer_name
                               from depositor)
```

FIGURE 2.16: Construct

All Construct Exists Construct Find all customers who have an account at all branches located

Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch_name
  from branch as T, branch as S
 where T.assets > S.assets and
       S.branch_city = 'Brooklyn'
```

■ Same query using > some clause

```
select branch_name
  from branch
 where assets > some
       (select assets
        from branch
       where branch_city = 'Brooklyn')
```

FIGURE 2.17: Some Construct

in Brooklyn. absence of duplicate tuples the unique construct tests whether a sub query has any duplicate tuples in its result.

Find all customers who have at most one account at the Perryridge branch.

```
select branch_name
  from branch
where assets > all
    (select assets
      from branch
      where branch_city = 'Brooklyn')
```

FIGURE 2.18: all Construct

```
select distinct S.customer_name
  from depositor as S
where not exists (
  (select branch_name
    from branch
    where branch_city = 'Brooklyn')
  except
  (select R.branch_name
    from depositor as T, account as R
    where T.account_number = R.account_number and
      S.customer_name = T.customer_name ))
```

FIGURE 2.19: exists Construct

2.5 Outer joins

Join operations take two relations and return as a result another relation. These additional operations are typically used as subquery expressions in the from clause

Join types	Join Conditions
inner join left outer join right outer join full outer join	natural on <predicate> using (A ₁ , A ₁ , ..., A _n)

FIGURE 2.20: Joins

Join condition defines which tuples in the two relations match, and what attributes are present in the result of the join. **Join type** defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated. Joined Relations - Datasets for Examples

Joined Relations – Examples

loan inner join borrower on loan.loan number = borrower.loan number

Joined Relations – Examples loan natural inner join borrower

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	Hayes	L-155
<i>loan</i>			<i>borrower</i>	

FIGURE 2.21: Relation loan

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

FIGURE 2.22: Joined Relations

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

FIGURE 2.23: Joined Relations1

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	null	null

FIGURE 2.24: Joined Relations1

2.6 Complex Integrity Constraints in SQL

2.6.1 1. Constraints over a Single Table

We can specify complex constraints over a single table using table constraints, which have the form CHECK conditional-expression.

example, to ensure that rating must be an integer in the range 1 to 10, use:

```
CREATE TABLE Sailors ( sid INTEGER,
```

```
sname CHAR(10),
```

```
rating INTEGER, age REAL, PRIMARY KEY (sid),
```

```
CHECK ( rating >= 1 AND rating <= 10 )
```

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

```
CREATE TABLE Reserves ( sid INTEGER, bid INTEGER, day DATE, FOREIGN KEY (sid)
```

```
REFERENCES Sailors FOREIGN KEY (bid) REFERENCES Boats CONSTRAINT
```

noInterlakeRes CHECK (‘Interlake’ >< (SELECT B.bname FROM Boats B WHERE B.bid = Reserves.bid))

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

2.6.2 Domain Constraints

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

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

INTEGER is the base type for the domain ratingval, and every ratingval value must be of this type. The optional DEFAULT keyword is used to associate a default value with a domain

2.6.3 3. Assertions: ICs over Several Tables

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold only if the associated table is nonempty. SQL supports the creation of assertions, which are constraints not associated with any one table.

Example: enforce the constraint that the number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE Sailors ( sid INTEGER,
sname CHAR(10), rating INTEGER, age REAL,
PRIMARY KEY (sid),
CHECK ( rating >=1AND rating <= 10) CHECK ((SELECT COUNT (S.sid) FROM Sailors S )
+(SELECT COUNT (B.bid) FROM Boats B ) <100 ))
```

The best solution is to create an assertion, as follows:

```
CREATE ASSERTION smallClub CHECK ((SELECT COUNT (S.sid) FROM Sailors S )
+(SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

Chapter 3

INTRODUCTION TO SCHEMA REFINEMENT

Course Outcomes

After successful completion of this module, students should be able to:

CO 8	Demonstrate the use of SQL for database creation and maintenance.	Understand
CO 9	Make Use of SQL queries for data aggregation, calculations, views, sub-queries, embedded queries manipulation.	Apply
CO 10	Illustrate the definition of Functional Dependencies, Inference rules and minimal sets of FD's to maintain data integrity.	Understand

3.1 Introduction

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.

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.

DECOMPOSITIONS

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural.

Functional dependencies (ICs) can be used to identify such situations and to suggest refinements to the schema.

	<i>rating</i>	<i>hourlywages</i>		
	8	10		
	5	7		
<i>ssn</i>	<i>Name</i>	<i>lot</i>	<i>rating</i>	<i>hours worked</i>
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

FIGURE 3.1: Refinements

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 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)

3.1.1 Problem 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?

FUNCTIONAL DEPENDENCIES Functional dependency (FD) is set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by arrow sign (\rightarrow), that is $X \rightarrow Y$, where X functionally determines Y. The left hand side attributes determines the values of attributes at right hand side.

3.1.2 Armstrong's Axioms

If F is set of functional dependencies then the closure of F, denoted as F^+ , is the set of all functional dependencies logically implied by F. Armstrong's Axioms are set of rules, when applied repeatedly generates closure of functional dependencies.

Reflexive rule: If alpha is a set of attributes and beta is subset of alpha, then alpha holds beta.

Augmentation rule: if $a \rightarrow b$ holds and y is attribute set, then $ay \rightarrow by$ also holds. That is adding attributes in dependencies, does not change the basic dependencies.

Transitivity rule: Same as transitive rule in algebra, if $a \rightarrow b$ holds and $b \rightarrow c$ holds then $a \rightarrow c$ also hold. $a \rightarrow b$ is called as a functionally determines b.

3.2 Trivial Functional Dependency

- Trivial: If an FD $X \rightarrow Y$ holds where Y subset of X, then it is called a trivial FD. Trivial FDs are always hold.
- Non-trivial: If an FD $X \rightarrow Y$ holds where Y is not subset of X, then it is called non-trivial FD.
- Completely non-trivial: If an FD $X \rightarrow Y$ holds where $X \cap Y = \emptyset$, is said to be completely non-trivial FD.

3.2.1 Reasoning about FDS

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 $ssn \rightarrow did$ holds, since ssn is the key, and FD $did \rightarrow lot$ 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 $ssn \rightarrow lot$ also holds on Workers. We say that an FD f is implied by a given setFof 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.

3.3 Normal forms

- Definition Normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency.
- The normal forms based on FDs are first 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 definition of the relational model.
- Although some of the newer database systems are relaxing this requirement 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

3.3.1 First normal form

- A relation schema is said to be in first normal form if the attributes values in the relation are atomic, i.e there should be no repeated values in a particular column
- A attribute is said to be value atomic value if it contains only a single.

Emp_id	Emp_section_id	Emp_name	Emp_address	dependents
0012	575	Manideep	Hyderabad	Father, Mother,Brother
0013	572	Bhaskar reddy	Delhi	Wife, Mother, Son
0014	5A0	Priyanka	Bangalore	Brother, Sister
0015	5B8	Anusha reddy	Hyderabad	Sister, Mother

FIGURE 3.2: Example First Normal Form

Here,The column dependents have non atomic values, In order to convert this relation in INF,we have to convert these non atomic values to atomic values

The relation employee is in 1NF since the column dependents have atomic value But the other attributes i.e. emp id, emp section id, emp name, emp address are all repeating and forming a group called repeated groups.

Emp_id	Emp_section_id	Emp_name	Emp_address	Dependents
0012	575	Manideep	Hyderabad	Father,
0012	575	Manideep	Hyderabad	Mother
0012	575	Manideep	Hyderabad	Brother
0013	572	Bhaskar reddy	Delhi	Wife
0013	572	Bhaskar reddy	Delhi	Mother
0013	572	Bhaskar reddy	Delhi	Son
0014	5A0	Priyanka	Bangalore	Brother
0014	5A0	Priyanka	Bangalore	Sister
0015	5B8	Anusha reddy	Hyderabad	Sister
0015	5B8	Anusha reddy	Hyderabad	Mother

FIGURE 3.3: Refinements

3.3.2 Second normal form

- A relation is said to be in 1NF and every non Key attribute is fully functionally dependent on primary key attribute
- If any one of the following conditions are satisfied then a relation(which is in 1NF) is in 2NF

Rules:

1. There should be only one attribute associated with primary key
2. There must be no non key attributes in the relation

Example:

- Student(student id,class id,name,course,time)
- (student id,class id,)is the primary key,
- A student can attend different course in different classes at different times.

The above relation is not in 2NF, as the name of the student can be determined by student id. there , a non key attribute(name) is functionally depend on a part of key (student id)

3.3.3 Third normal form

A relation R in 3NF if and only if it is in 2NF and every non key column does not depend on another non key column

- All nonprime attributes of R must be non-transitively functionally dependent on a key of the relation
- Relation R with FDs F is in 3NF if, for all X A in
1 A X (called a trivial FD), or

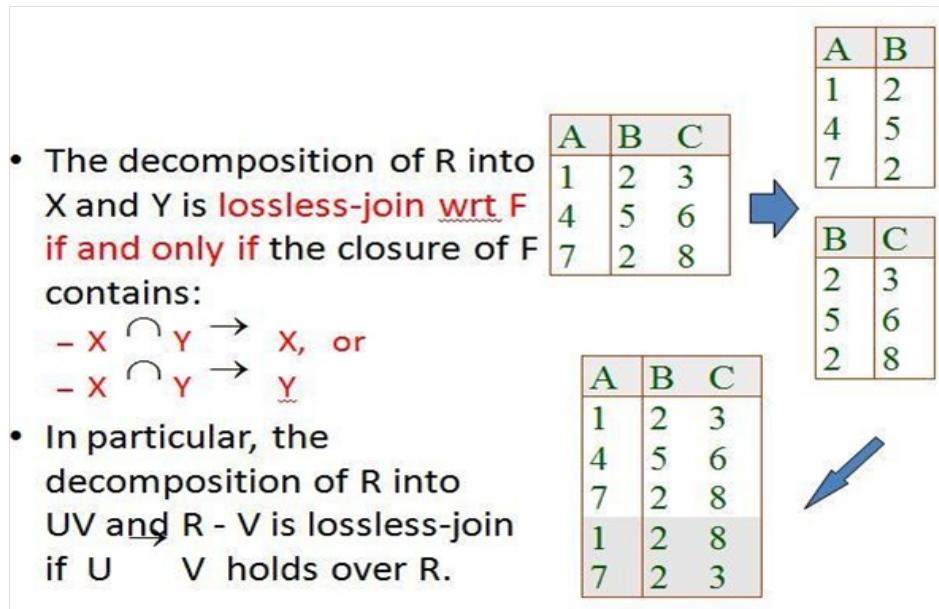


FIGURE 3.4: example of student relation

2 X contains a key for R, or

3 A is part of some key for R.

- Minimality of a key is crucial in third condition above

• If R is in BCNF, obviously in 3NF.

• If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no “good” decomp, or performance considerations).

4 Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.

SUPPLIER (SNAME, STREET, CITY, STATE, TAX) SNAME → STREET, CITY, STATE STATE → TAX (non key → non key) SNAME → STATE → TAX (transitive FD)

- solution: decompose the relation SUPPLIER2 (SNAME, STREET, CITY, STATE) TAXINFO (STATE, TAX)

Boyce-Codd NORMAL FORM (BCNF)

- Reln R with FDs F is in BCNF if, for all X A in

1 A X (called a trivial FD), or

2 X contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.

3 No dependency in R that can be predicted using FDs alone.

4 If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.

5 If example relation is in BCNF, the 2 tuples must be identical (since x is a key).

3.4 Properties of Decompositions

DECOMPOSITION OF A RELATION SCHEME

- Suppose that relation R contains attributes A₁ ... A_n. A decomposition of R consists of replacing R by two or more relations such that:
 - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
 - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

3.4.1 Lossless Join Decompositions:

- Decomposition of R into X and Y is lossless join w.r.t. a set of FDs F if, for every instance r that satisfies F:
 - It is always true that In general, the other direction does not hold! If it does, the decomposition is lossless-join.
 - Definition extended to decomposition into 3 or more relations in a straightforward way.
 - It is essential that all decompositions used to deal with redundancy be lossless!

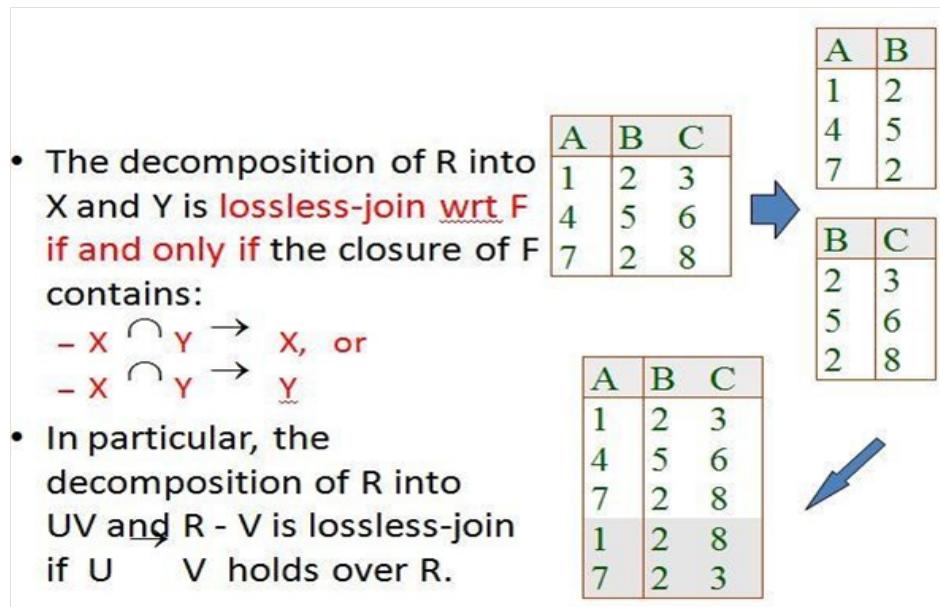


FIGURE 3.5: More on Lossless Join

3.4.2 Dependency Preserving Decomposition (Intuitive)

If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. (Avoids Problem (3).)

Projection of set of FDs F: If R is decomposed into X, ... projection of F onto X (noted F_X) is the set of FDs UV in F^+ (closure of F) such that U, V are in X. Decomposition of R into X and Y is dependency preserving if $(F_X \cup F_Y) = F^+$ i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+ . Important to consider F^+ , not F, in this definition:

ABC, A B, B C, C A, decomposed into AB and BC. Is this dependency preserving? Is C A preserved????? Dependency preserving does not imply lossless join: ABC, A B, decomposed into AB and BC. And vice-versa

Decomposition into BCNF Consider relation R with FDs F. If X Y violates BCNF, decompose R into R - Y and XY. Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

- e.g., CSJDPQV, key C, JP C, SD P, JS
- To deal with SD P, decompose into SDP, CSJDQV.
- To deal with JS, decompose CSJDQV into JS and CJDQV

In general, several dependencies may cause violation of BCNF. The order in which we deal with them could lead to very different sets of relations

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
- e.g., CSZ, CS Z, Z C
- Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP C, SD P and JS). However, it is a lossless join decomposition. In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition. JPC tuples stored only for checking FD! (Redundancy)

Decomposition into 3NF

Obviously, the algorithm for lossless join decompose into BCNF can be used to obtain a lossless join decompose into 3NF (typically, can stop earlier).

To ensure dependency preservation, one idea:

If X Y is not preserved, add relation XY.

Problem is that XY may violate 3NF! e.g., consider the addition of CJP to 'preserve' JP C. What if we also have JC? Refinement: Instead of the given set of FDs F, use a minimal cover for F.

3.5 Schema Refinement in Data Base Design:

Constraints on an Entity Set

Consider the Hourly Emps relation again. The constraint that attribute ssn is a key can be expressed as an FD: { ssn } > { ssn, name, lot, rating, hourly wages, hours worked }

For brevity, we will write this FD as S -> SNLRWH, using a single letter to denote each attribute. In addition, the constraint that the hourly wages attribute is determined by the rating attribute is an FD: R -> W

Constraints on a Relationship Set

The previous example illustrated how FDs can help to refine the subjective decisions made during ER design, but one could argue that the best possible ER diagram would have led to the same null set of relations. Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

Identifying Attributes of Entities

In particular, it shows that attributes can easily be associated with the ‘wrong’ entity set during ER design. The ER diagram shows a relationship set called Works_In that is similar to the Works_In relationship set. Using the key constraint, we can translate this ER diagram into two relations:

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

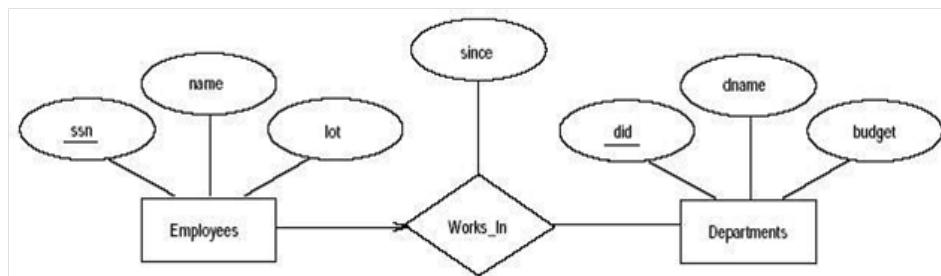


FIGURE 3.6: Identifying Attributes of Entities

Identifying Entity Sets Let Reserves contain attributes S, B, and D as before, indicating that sailor S has a reservation for boat B on day D.

In addition, let there be an attribute C denoting the credit card to which the reservation is charged. Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD

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.

There are three points to note here:

The relation schema CTB is in BCNF; thus we would not consider decomposing it further if we

<i>course</i>	<i>teacher</i>	<i>book</i>
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

FIGURE 3.7: Dependencies

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 Physics 101 is recorded once per potential teacher.

The redundancy can be eliminated by decomposing CTB into CT and CB. 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 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. Three of the additional rules involve only MVDs:

MVD Complementation: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R \rightarrow XY$

MVD Augmentation: If $X \rightarrow\rightarrow Y$ and $W > Z$, then $WX \rightarrow\rightarrow YZ$

MVD Transitivity: If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then $X \rightarrow\rightarrow (Z \rightarrow Y)$

3.5.1 Forth normal form

R is said to be in fourth normal form (4NF) if for every MVD $X \rightarrow\rightarrow Y$ that holds over R, one of the following statements is true:

- Y subset of X or $XY = R$, or
- X is a super key.

JOIN DEPENDENCIES

A join dependency is a further generalization of MVDs. A join dependency (JD) $\infty \{ R_1, \dots, R_n \}$ is said to hold over a relation R if R_1, \dots, R_n is a lossless join decomposition of R.

An MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\infty \{ XY, X(R - Y) \}$. As an example, in the CTB relation, the MVD $C \rightarrow\rightarrow T$ can be expressed as the join dependency

$\infty \{ CT, CB \}$ Unlike FDs and

FIFTH NORMAL FORM

A relation schema R is said to be in fth normal form (5NF) if for every JD $\infty R_1, \dots, R_n$ 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 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.

3.5.2 Inclusion Dependencies

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about. In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design the main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency. Most inclusion dependencies in practice are key-based, that is, involve only keys.

Chapter 4

TRANSACTION CONCEPT

Course Outcomes

After successful completion of this module, students should be able to:

CO 11	Apply normalization techniques to normalize a database .	Apply
CO 12	State the concepts of transaction, states and ACID properties in data manipulation.	Remember
CO 13	Demonstrate concurrency control protocols to preserve the database in a consistent state.	Understand
CO 14	Illustrate the problems of data management in a concurrent environment by using recovery techniques to recover the lost data.	Understand

4.1 Introduction

A Transaction is a unit of program execution that accesses and possibly updates various data items.

Example transaction to transfer \$50 from account A to account B:

1. read(A)
2. A:=A-50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)

Two main issues to deal with:

Failures of various kinds, such as hardware failures and system crashes Concurrent execution of

multiple transactions

Example of Fund Transfer Transaction to transfer \$50 from account A to account B:

1. read(A)
2. A:=A-50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)

Atomicity requirement if the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state Failure could be due to software or hardware the system should ensure that updates of a partially executed transaction are not reflected in the database

Durability requirement once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer Transaction to transfer \$50 from account A to account B:

1. read(A)
2. A:=A-50
3. write(A)
4. read(B)
5. B:=B+50
6. write(B)

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction In general, consistency requirements include Explicitly specified integrity constraints such as primary keys and foreign keys Implicit integrity constraints Example sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand A transaction must see a consistent database. During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully the database must be consistent Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer Isolation requirement if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).

T1 T2

1. read(A)
2. A:=A-50
3. write(A)
read(A), read(B), print(A+B)
4. read(B)
5. B:=B+50
6. write(B)

Isolation can be ensured trivially by running transactions serially that is, one after the other. However, executing multiple transactions concurrently has significant benefits.

4.2 ACID Properties

Atomicity Either all operations of the transaction are properly reflected in the database or none are.

consistency Execution of a transaction in isolation preserves the consistency of the database.

Isolation Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.

Durability After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

4.3 Transaction State

- Active – the initial state; the transaction stays in this state while it is executing
- Partially committed – after the final statement has been executed.
- Failed – after the discovery that normal execution can no longer proceed.
- Aborted – after the transaction has been rolled back and the database restored to its State prior to the start of the transaction. Two options after it has been aborted: restart the transaction can be done only if no internal logical error kill the transaction
- Committed – after successful completion.

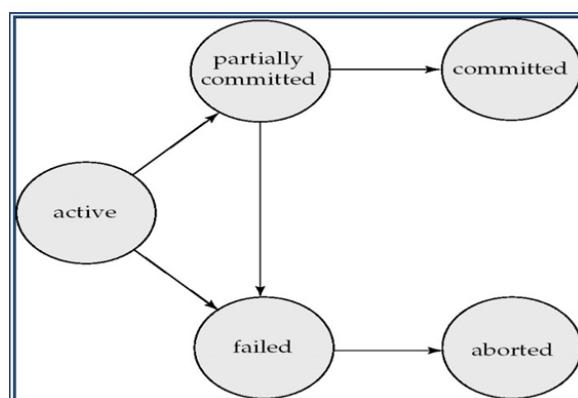


FIGURE 4.1: Transaction State

4.4 Implementation of Atomicity And Durability

The recovery-management component of a database system implements the support for atomicity and durability. Example of the shadow-database scheme: all updates are made on a shadow copy of the database db pointer is made to point to the updated shadow copy after the transaction reaches partial commit and all updated pages have been flushed to disk. Db pointer always points to the current consistent copy of the database. In case transaction fails, old consistent copy pointed to by db pointer can be used, and the shadow copy can be deleted. The shadow-database scheme:

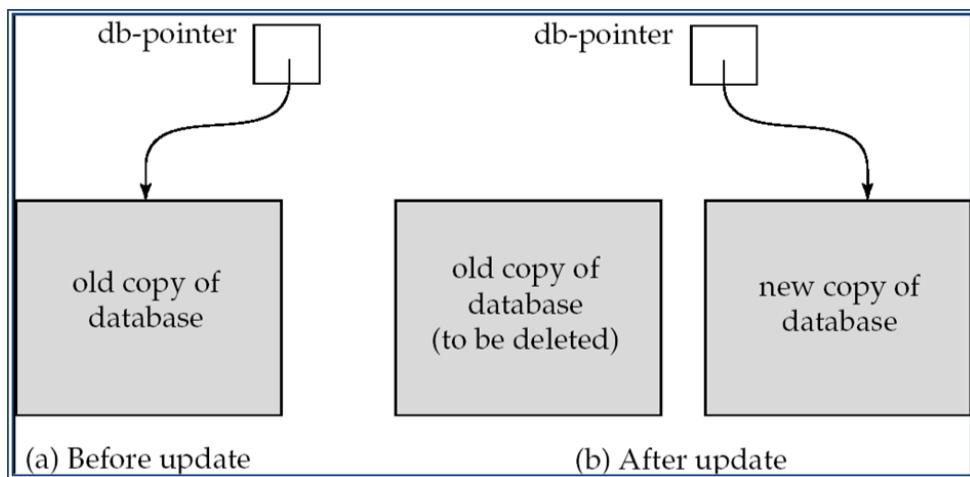


FIGURE 4.2: shadow database

Assumes that only one transaction is active at a time. Assumes disks do not fail Useful for text editors, but extremely inefficient for large databases (why?) Variant called shadow paging reduces copying of data, but is still not practical for large databases does not handle concurrent transactions

CONCURRENT EXECUTIONS

Multiple transactions are allowed to run concurrently in the system. Advantages are: Increased processor and disk utilization, leading to better transaction throughput

Example 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.

Schedule – 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 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 T1 transfer \$50 from A to B, and T2 transfer 10% of the balance from A to B. A serial schedule in which T1 is followed by T2 :

T_1	T_2
<pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) </pre>	<pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) </pre>

FIGURE 4.3: Schedule1

T_1	T_2
<pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) </pre>	<pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) </pre>

FIGURE 4.4: Schedule2

T_1	T_2
<pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) </pre>	<pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) </pre>

FIGURE 4.5: Schedule3

Let T1 and T2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1.

4.5 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:

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

FIGURE 4.6: Conflict serializability

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

FIGURE 4.7: View serializability

Simplified views 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 lid and elk of transactions T_i and T_j Respectively, conflict if and only if there exists some item Q accessed by both li and lj, and at least one of these instructions wrote Q.

1. $l_i = \text{read}(Q), l_j = \text{read}(Q)$. l_i and l_j don't conflict.
2. $l_i = \text{read}(Q), l_j = \text{write}(Q)$. They conflict.
3. $l_i = \text{write}(Q), l_j = \text{read}(Q)$. They conflict
4. $l_i = \text{write}(Q), l_j = \text{write}(Q)$. They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_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.

4.5.1 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. Schedule 3 can be transformed into Schedule 6, a serial schedule where T2 follows T1, by series

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

FIGURE 4.8: conflict serializable

of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable. Example of a schedule that is not conflict serializable: We are unable to swap instructions in the above schedule to obtain either the serial schedule $> T_3, T_4 >$, or the serial schedule $< T_4, T_3 >$.

4.6 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 Quid in schedule S, transaction Ti reads the initial value of Q, then in schedule S' also transaction Ti must read the initial value of Q.

If in schedule S transaction Ti executes read(Q), and that value was produced by transaction Tj (if any), then in schedule S' also transaction Ti must read the value of Q that was produced by the same write(Q) operation of transaction Tj . 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.

A schedule S is view serializable if it is view equivalent to a serial schedule. Every conflict

T_3	T_4	T_6
read(Q)		
write(Q)	write(Q)	write(Q)

FIGURE 4.9: reads and writes

serializable schedule is also view serializable. Below is a schedule which is view-serializable but not conflict serializable.

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has blind writes. Other Notions of Serializability

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

FIGURE 4.10: reads and writes

The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it. Determining such equivalence requires analysis of operations other than read and write.

4.7 Recoverability

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 appears before the commit operation of T_j . The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read. 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.

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

FIGURE 4.11: Recoverable schedule

4.7.1 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)

If T_{10} fails, T_{11} and T_{12} must also be rolled back. Can lead to the undoing of a significant amount

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

FIGURE 4.12: Cascadeless schedules

of work **Cascadeless schedules** — cascading rollbacks cannot occur; 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.

Chapter 5

CONCURRENCY CONTROL

Course Outcomes

After successful completion of this module, students should be able to:

CO 15	Describe disk storage devices, file organization to select efficient data storage.	Remember
CO 16	Apply indexing ,hashing techniques to access the records from the file effectively.	Apply
CO 17	Compare between indexing and hashing for efficient search process .	Understand
CO 18	Design a full real size database system for an industry or business scenario.	Analyze

5.1 Introduction

A database must provide a mechanism that will ensure that all possible schedules are either conflict or view serializable, and are recoverable and preferably cascade less A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency Are serial schedules recoverable/cascade less? Testing a schedule for serializability after it has executed is a little too late! Goal – to develop concurrency control protocols that will assure serializability.

Implementation of Isolation

Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, 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. Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$	$\text{read}(A)$ $temp := A * 0.1$ $A := A - temp$ $\text{write}(A)$ $\text{read}(B)$ $B := B + temp$ $\text{write}(B)$

FIGURE 5.1: Implementation of Isolation

5.1.1 Testing For Serializability

- Consider some schedule of a set of transactions T_1, T_2, \dots, 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

Test for Conflict Serializability A schedule is conflict serializable if and only if its precedence

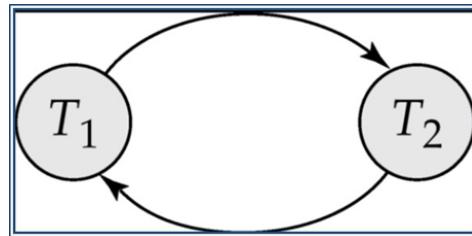


FIGURE 5.2: Testing For Serializability

graph is acyclic. 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. This is a linear order consistent with the partial order of the graph. For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$. Are there others?

Example Schedule (Schedule A) + Precedence Graph				
T_1	T_2	T_3	T_4	T_5
read(Y) read(Z)	read(X)			
	read(Y) write(Y)			
read(U)		write(Z)		
read(U) write(U)			read(Y) write(Y) read(Z) write(Z)	read(V) read(W) read(W)

```

graph TD
    T1((T1)) --> T2((T2))
    T2((T2)) --> T3((T3))
    T3((T3)) --> T4((T4))
    T4((T4)) --> T5((T5))
    T1((T1)) --> T1((T1))
    T2((T2)) --> T2((T2))
    T3((T3)) --> T3((T3))
  
```

FIGURE 5.3: Testing For Serializability

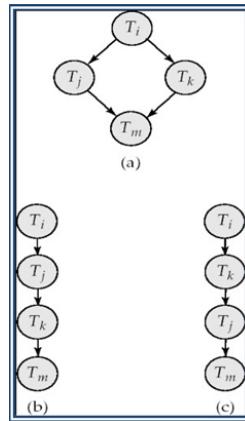


FIGURE 5.4: Conflict Serializability

5.2 Concurrency Control

Concurrency Control vs. Serializability Tests

Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascade less. Concurrency control protocols generally do not examine the precedence graph as it is being created. Instead a protocol imposes a discipline that avoids non serializable schedules. Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur. Tests for serializability help us understand why a concurrency control protocol is correct.

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. Example. 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-92 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. 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);

5.3 Lock Based Protocols

A lock is a mechanism to control concurrent access to a data item

Fig: Lock-compatibility matrix

Data items can be locked in two modes: 1. Exclusive (X) mode. Data item can be both read as

	S	X
S	true	false
X	false	false

FIGURE 5.5: Lock compatibility matrix

well as written. X-lock is requested using lock-X instruction.

2. Shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example: if a transaction performing locking:

T2: lock-S (A);
 Read (A);
 Unlock (A);
 Lock-S (B);
 Read (B);
 Unlock (B);
 Display (A+B)

Locking as above is not sufficient to guarantee serializability — if A and B get updated in- between the read of A and B, the displayed sum would be wrong.

- A locking protocol is a set of rules followed by all transactions while requesting and Releasing locks. Locking protocols restrict the set of possible schedules. Pitfalls of Lock- Based Protocols Consider the partial schedule Neither T3 nor T4 can make progress Executing lock-S (B) causes T4 to wait for T3 to release its lock on B, while executing lock-X (A) Causes T3 to wait for T4 to release its lock on A. Such a situation is called a deadlock. To handle a Deadlock one of T3 or T4 must be rolled back and its locks released. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

Starvation is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. The same transaction is repeatedly rolled back due to deadlocks. Concurrency control manager can be designed to prevent starvation.

5.4 Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules. Phase 1: Growing Phase

1. Transaction may obtain locks
2. Transaction may not release locks Phase 2: Shrinking Phase
3. Transaction may release locks
4. Transaction may not obtain locks

The protocol assures serializability It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock). All locks are released after commit or abort

Implementation of Locking

A lock manager can be implemented as a separate process to which transactions send lock And unlock requests the lock manager replies to a lock request by sending a lock grant Messages (or a message asking the transaction to roll back, in case of a deadlock).The Requesting transaction waits until its request is answered the lock manager maintains a data- Structure called a lock table to record granted locks and pending requests the lock table is Usually implemented as an in-memory hash table indexed on the name of the data item being Locked.

Two-phase locking does not ensure freedom from deadlocks

- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.
- Rigorous two-phase locking is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

5.4.1 Time stamp Based Protocols

Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp $TS(T_i)$, a new transaction T_j is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.

The protocol manages concurrent execution such that the time-stamps determine the serializability order. In order to assure such behavior, the protocol maintains for each data Q two timestamp values:

$W\text{-timestamp}(Q)$ is the largest time-stamp of any transaction that executed $\text{write}(Q)$ successfully.

$R\text{-timestamp}(Q)$ is the largest time-stamp of any transaction that executed $\text{read}(Q)$ successfully. The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

Suppose a transaction T_i issues a $\text{read}(Q)$

If $TS(T_i) \leq W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already over written. Hence, the read operation is rejected, and T_i is rolled back.

If $TS(T_i) \leq R\text{-timestamp}(Q)$, then the read operation is executed, and $R\text{-timestamp}(Q)$ is set to $\max(R\text{-timestamp}(Q), TS(T_i))$.

Suppose that transaction T_i issues $\text{write}(Q)$.

If $TS(T_i) \leq R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and T_i is rolled back.

If $TS(T_i) \leq W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q .

Hence, this write operation is rejected, and T_i is rolled back. Otherwise, the write operation is executed, and $W\text{-timestamp}(Q)$ is set to $TS(T_i)$.

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

5.5 Correctness of Timestamp-Ordering Protocol

The time stamp ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form $T_i \rightarrow T_j$. Thus, there will be no cycles in the precedence graph. Timestamp protocol ensures freedom from deadlock as no transaction ever waits. But the schedule may not be cascade-free, and may not even be recoverable.

Example Use of the Protocol				
A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5				
T_1	T_2	T_3	T_4	T_5
read(1)	read(1)			read(3)
read(2)	read(3) abort	write(1) write(2)	write(2) abort	read(2)
		write(2)		write(1) write(2)

FIGURE 5.6: Use of the protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

FIGURE 5.7: Timestamp-Ordering Protocol

Thomas Write Rule Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances. When T_i attempts to write data item Q , if $TS(T_i) \leq TS(Q)$, then T_i is attempting to write an obsolete value of Q . Rather than rolling back T_i as the timestamp ordering protocol would have done, this {write} operation can be ignored. Otherwise this protocol is the same as the timestamp ordering protocol.

1 Thomas' Write Rule allows greater potential concurrency.

2 Allows some view-serializable schedules that are not conflict-serializable.

5.5.1 Validation Based Protocol

Execution of transaction T_i is done in three phases. 1. Read and execution phase: Transaction T_i writes only to temporary local variables

2. Validation phase: Transaction T_i performs a “validation test” to determine if local variables can be written without violating serializability.

3. Write phase: If T_i is validated, the updates are applied to the ollied back. Database; otherwise,

T_i is

The three phases of concurrently executing transactions can be interleaved, but each Transaction must go through the three phases in that order. Assume for simplicity that the validation and write phase occur together, atomically and serially i.e., only one transaction executes validation/write at a time. Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation. Each transaction

T_i has 3 timestamps

- $\text{Start}(T_i)$: the time when T_i started its execution
- $\text{Validation}(T_i)$: the time when T_i entered its validation phase
- $\text{Finish}(T_i)$: the time when T_i finished its write phase Serializability order is determined by timestamp given at validation time, to increase concurrency.

- Example of schedule produced using validation

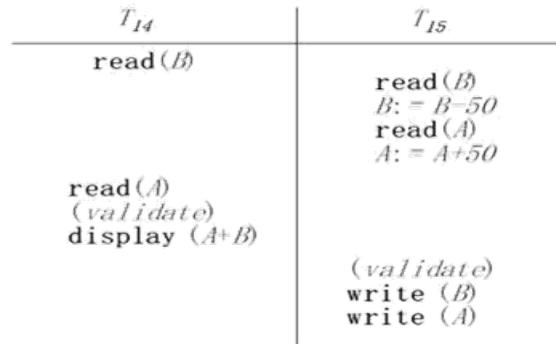


FIGURE 5.8: Validation Based Protocol

5.6 Multiple Granularities

Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones Can be represented graphically as a tree (but don't confuse with tree-locking protocol) When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendants in the same mode.

Granularity of locking (level in tree where locking is done): fine granularity (lower in tree): high concurrency, high locking overhead coarse granularity (higher in tree): low locking overhead, low concurrency The levels, starting from the coarsest (top) level are

- * database
- * area
- * file
- * record

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

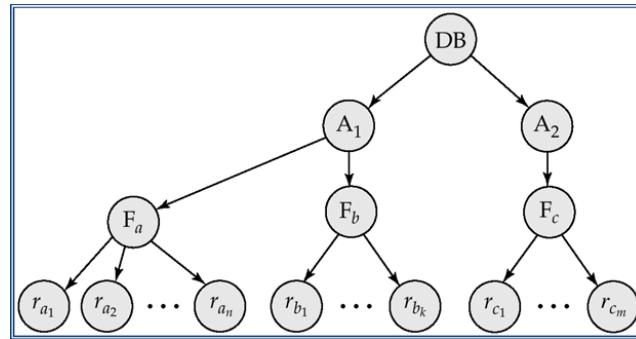


FIGURE 5.9: Multiple Granularities

Intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks

Shared and intention-exclusive (SIX): the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks. Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Intention-shared (IS): indicates explicit locking at a lower level of the tree but only with shared locks.

Intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks

Shared and intention-exclusive (SIX): the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks. Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
S IX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

FIGURE 5.10: Shared and intention-exclusive

5.7 Recovery System

Failure classification

To see where the problem has occurred we generalize the failure into various categories, as follows:

Transaction failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

- Logical errors: where a transaction cannot complete because of it has some code error or any internal error condition

System errors: where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

System crash

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk failure:

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

5.7.1 Recovery and Atomicity

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state. Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all database modifications made by T_i or none at all. Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications has been made but before all of them are made. To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself. We study two approaches:

Log-based recovery and shadow-paging. We assume (initially) that transactions run serially, that are one after the other. Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and

durability despite failures.

Recovery algorithms have two parts

Actions taken during normal transaction processing to ensure enough information exists to recover from failures. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Log-based recovery • Log is kept on stable storage. The log is a sequence of log records, and maintains a record of update activities on the database.

- When transaction T_i starts, it registers itself by writing a $\langle T_i \text{ start} \rangle$ log record Before T_i executes write(X), a log record $\langle T_i, X, V_1, V_2 \rangle$ is written,
- Where V_1 is the value of X before the write and V_2 is the value to be written to X .
- Log record notes that T_i has performed a write on data item $A_x X_j$ had value V_1 before the write, and will have value V_2 after the write.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

- Deferred database modification
- Immediate database modification
- Deferred Database Modification

The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.

- Immediate Database Modification

The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued since undoing may be needed, update logs must have both old value and new value. Update log record must be written before database item is written. We assume that the log record is output directly to stable storage. Can be extended to postpone log record output, so long as prior to execution of an output (B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage.

Checkpoints

Problems in recovery procedure:

- searching the entire log is time-consuming
- We might unnecessarily redo transactions which have already output their updates to the database. Streamline recovery procedure by periodically performing check pointing. Output all log records currently residing in main memory onto stable storage. Output all modified buffer blocks to the disk. Write a log record $\langle \text{checkpoint} \rangle$ onto stable storage.

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 $\langle \text{checkpoint} \rangle$ record. Continue scanning backwards till a record $\langle T_i \text{ start} \rangle$ is found. Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.

For all transactions (starting from T_i or later) with no $\langle T_i \text{ commit} \rangle$, execute undo (T_i). (Done

only in case of immediate modification.) Scanning forward in the log, for all transactions starting from T_i or later w $\langle T_i \text{ commit} \rangle$, execute redo (T_i).

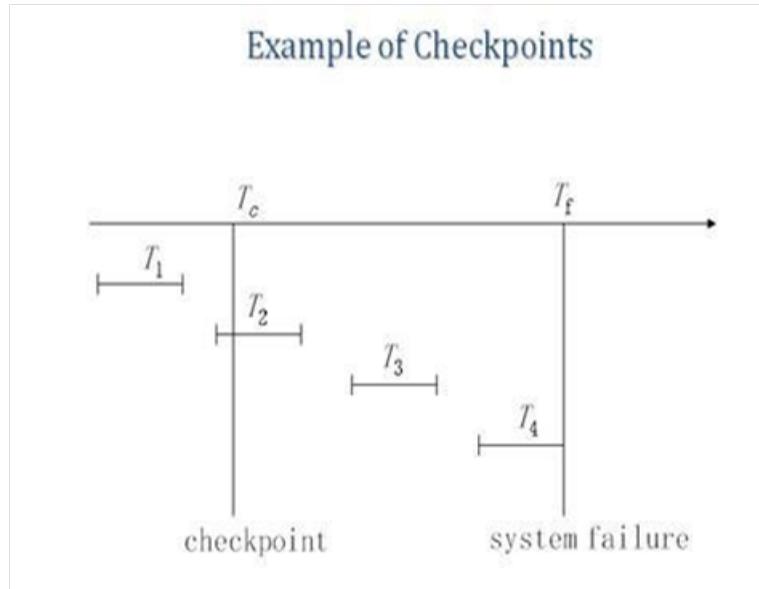


FIGURE 5.11: Shared and intention-exclusive

5.8 Recovery with concurrent transactions

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently. All transactions share a single disk buffer and a single logia buffer block can have data items updated by one or more transactions . We assume concurrency control using strict two-phase locking; i.e. the updates of uncommitted transactions should not be visible to other transactions, Otherwise how to perform undo if T_1 updates a, then T_2 updates A and commits, and finally T_1 has to abort? Logging is done as described earlier. Log records of different transactions may be interspersed in the log.

- The check pointing technique and actions taken on recovery have to be changed since several transactions may be active when a checkpoint is performed.

5.8.1 Log Record Buffering

Log record buffering: log records are buffered in main memory, instead 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.

5.8.2 Buffer Management

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. 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. Before a block is output to disk, the system acquires an exclusive latch on the block. Ensures no update can be in progress on the block. 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. 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

Failure With Loss of Nonvolatile Storage

Technique similar to check pointing 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 check pointing 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

5.9 Advanced Recovery Techniques

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 including redo of log records of incomplete transactions, followed by subsequent undo. Key benefits supports logical undo easier to understand/show correctness

Advanced Recovery: 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.

Advanced Recovery:Physical Redo Redo information is logged physically (that is, new value for each write) even for Operations with logical undo. Logical redo are 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.

Advanced Recovery: Operation Logging

Operation logging is done as follows: When operation starts, log $\langle Ti, on, \text{operation begin} \rangle$. Here on is a unique identifier of the operation instance. While operation is executing, normal log records with physical redo and physical undo information are logged. When operation completes, $\langle Ti, on, \text{operation-end}, U \rangle$ is logged, where U contains information needed to perform a logical undo information.

Advanced Recovery: Crash Recovery

The following actions are taken when recovering from system crash (Redo phase): Scan log forward from last $\langle \text{checkpoint } L \rangle$ record till end of log. Repeat history by physically redoing all updates of all transactions. Create an undo list during the scan as follows: undo list is set to L initially. Whenever $\langle Ti \text{ start} \rangle$ is found, Ti is added to undo-list. Whenever $\langle Ti \text{ commit} \rangle$ or $\langle Ti \text{ abort} \rangle$ is found, Ti 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. (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 re found.

Advanced Recovery: Check pointing Check pointing is done as follows:

- * Output all log records in memory to stable storage
- * Output to disk all modified buffer blocks
- * Output to log on stable storage at $\langle \text{checkpoint } L \rangle$ record.

Transactions are not allowed to perform any actions while check pointing is in progress.

Advanced Recovery: Fuzzy Check pointing Fuzzy check pointing is done as follows:

- * Temporarily stop all updates by transactions
- * Write a $\langle \text{checkpoint } L \rangle$ log record and force log to stable storage
- * Note list M of modified buffer blocks
- * Now permit transactions to proceed with their actions
- * 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 Store a pointer to the checkpoint record in a fixed position last_checkpoint on disk.

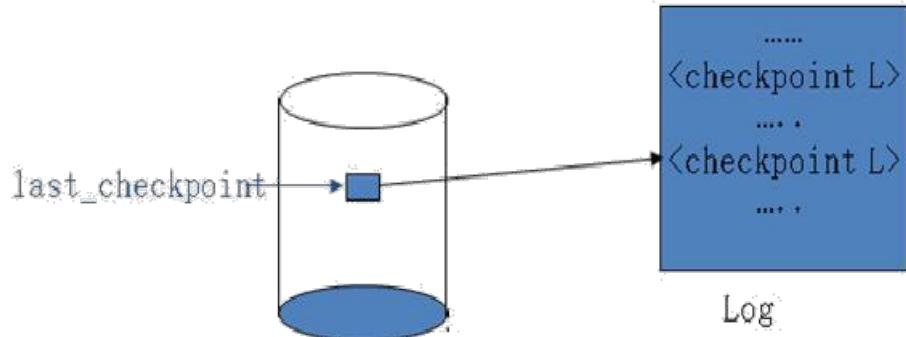


FIGURE 5.12: Fuzzy Check pointing

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. ARIES is a state of the art recovery method. Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery. The “advanced recovery algorithm” we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations. Unlike the advanced recovery algorithm, ARIES 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.

5.10 Remote Backup Systems

Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed. 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

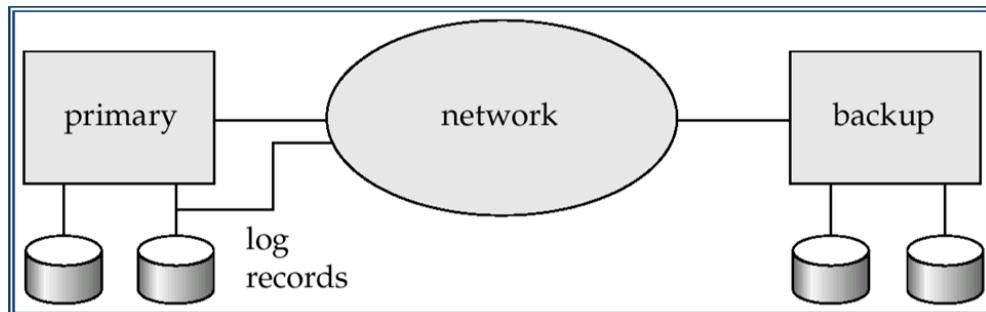


FIGURE 5.13: Remote Backup Systems

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.

Time to recover: To reduce delay in takeover, backup site periodically processes 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. 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.