PERIYAR INSTITUTE OF DISTANCE EDUCATION (PRIDE)

PERIYAR UNIVERSITY

SALEM - 636 011.

B.Sc. COMPUTER SCIENCE

THIRD YEAR

PAPER – V : DATA BASE MANAGEMENT SYSTEM

Prepared by :
S.RAJASEKARAN, M.Sc., M.Phil.,
Lecturer, Department of Computer Science,
Vivekanandha College of Arts
and Sciences for Women,
Elayampalayam,
Tiruchengode – 637 205.

B.Sc. COMPUTER SCIENCE

THIRD YEAR

PAPER – V : DATA BASE MANAGEMENT SYSTEM

CLIENT/SERVER TECHNOLOGY AND RDBMS

UNIT I

DBMS- INRODUCTION, ER MODEL,
RELATIONAL MODEL

UNIT II

SQL – STRUCTURE AND LANGUAGES

UNIT III

CONSTRAINTS AND NORMALIZATION
TECHNIQUES

UNIT IV

CLIENT/SERVER INTRODUCTION

UNIT V

NOS, DSS AND TP-MONITORS

INTRODUCTION

Dear Students

RDBMS is a tool which is used to create and access the information using database. It provides various techniques to create and manipulate data. This book can also specify different DBMS languages.

Totally this book covers five units. The first unit deals with the introduction about DBMS, Overall structure of DBMS and deals about database administrator and users. This unit also describes about the structure of E-R diagram and it benefits and also describes relational model and relational algebra.

The second unit deals with the structure of SQL. Various operations carried out in SQL. Different SQL Languages like DDL, DML, and TCL. The third unit deals with the concept of constraints, Assertions and Triggers. And also describes decomposition and different Normalization Techniques with examples.

The fourth unit deals with the concept of client/server technology and its advantages. This also describes different types of servers used in this client/server model. The fifth unit deals with introduction to Network Operating System, Remote Procedure Call, Decision support system, and Data warehouse. It also describes about TP Monitors.

All the above said five units of this book have been prepared by Mr.S.RAJASEKARAN, M.Sc.,M.Phil., to make your task much easier while going through it.

PRIDE would be happy in you could make use of this learning material to enrich your knowledge and skills to serve the society.

UNIT I

Overview of Database Management Systems: Introduction - File System versus a DBMS - Advantages of DBMS - Describing and Storing Data in a DBMS - Structure of a DBMS - Introduction to Database Design: Introduction to ER Model - Conceptual Design with ER Model - The Relational Model: Introduction to Relational Model - Integrity Constraints over Relational Model - Introduction to Views - Destroying / Altering Tables and Views.

UNIT II

Relational Algebra and Relational Calculus: Relational Algebra - Relational Calculus - SQL: Queries, Constraints, Triggers - The form of a Basic SQL Queries - UNION, INTERSECT, and EXCEPT - Nested Queries - Aggregate Operators - Null Values - Triggers and Active Databases.

UNIT III

Schema Refinement and Normal Forms, Security and Authorization: Introduction to Database Security - Access Control - Discretionary Access Control - Mandatory Access Control - Security for Interne Applications, Network Model, Hierarchical Model.

UNIT IV

Parallel and Distributed Databases: Introduction - Architecture of Parallel Databases - Parallel Query Evaluation -Parallelizing individual Operations - Parallel Query Optimization - Introduction to Distributed Database - Distributed DBMS Architecture - Storing Data in a Distributed Database - Distributed Catalog Management - Distributed Query Processing - Updating Distributed Query optimization - Distributed Transactions - Distributed Concurrency Control -Distributed Recovery, Object Database Systems: Motivating Examples - Structured Data types - Operations on Structured Data - Encapsulation and ADTs – Inheritance - Objects, OIDs, and Reference Types - Database Design for an ORDBMS - ORDBMS Implementation Challenges – OODBMS - Comparing RDBMS, OODBMS, and ORDBMS.

UNIT V

Data Warehousing and Decision Support: Introduction to Decision Support - OLAP: Multidimensional Data Model - Multidimensional Aggregation Queries - Window Queries in SQL:1999 - Finding Answers Quickly -Implementation Techniques for OLAP - Data Warehousing - Views and Decision Support - View Materialization -Maintaining Materialized Views, Data Mining: Introduction to Data Mining - Counting Co-occurrences - Mining for Rules - Tree Structured Rules – Clustering - Similarity Search over Sequences - Incremental Mining and Data Streams -Additional Data Mining Tasks.

TEXT BOOKS:

1. "Database Management Systems"
Ramakrishnan & Gehrke
MC Graw Hill international Edition
Third Eition
2. "Database System Concepts"
Abraham Silberschatz, Henry F.Korth & S.Sudarshan
MC Graw Hill Company
Third Edition
(Only last two topics in Unit III)

Contents

1.OVERVIEW OF DATABASE SYSTEM

1.1 Introduction

Database:

Database is a collection of data, typically describing the activities of one or more related organizations.

DBMS:

Database Management System is software designed to assist in maintaining and utilizing large collection of data. The alternative to using a DBMS is to store the data in files and write application-specific code to manage it.

1.2 File System Vs DBMS

File systems have many drawbacks over DBMS. The drawbacks are

- We probably do not have 500 GB of main memory to hold all the data. We must therefore store data in a storage device such a disk or tape and bring relevant parts into main memory for processing as needed.
- Even if we have 500 GB of main memory, on computer systems with 32-bit addressing, we cannot refer directly to more than about 4 GB of data. We have to program some method of identifying all data items.
- We have to write special programs to answer each question a user may want to ask about the data. These programs are likely to be complex because such of the large volume of data to be searched.
- We must protect the data from inconsistent changes made by different users accessing the data concurrently. If applications must address the details of such concurrent access, this adds greatly to their complexity.
- We must ensure that data is restored to a consistent state if the system crashes while changes are being made.
- Operating systems provide only a password mechanism for security. This is not sufficiently flexible to enforce security policies in which different users have to access different subsets of the data.

1.3 Advantages of DBMS

Using a DBMS to manage data has many advantages.

Data Independence: Application programs should not, ideally, be exposed to details of data representation and storage. The DBMS provides as abstract view of the data that hides such details.

Efficient Data Access: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data Integrity and Security: If data is always accessed through the DBMS, the DBMS can enforce integrity constraints. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, it can enforce access controls that govern what data is visible to different classes of users.

Data Administration: When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy send for fine-tuning the storage of the data to make retrieval efficient.

Concurrent Access and Crash Recovery: A DBMS schedules concurrent access to the data in such a manner that users can think of the data as being accessed by only user at a time. Further, the DBMS protects users from the effects of system failures.

Reduced Application Development Time: Clearly, the DBMS supports important functions that are common to many applications accessing data in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick application development. DBMS applications are also likely to be more robust that similar stand-alone applications because many important tasks are handled by the DBMS.

1.4 Describing and Storing Data in DBMS

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

A semantic data model is more abstract, high-level data model that makes it easier for a user to come up with a good initial description of the data in an enterprise. These models contain a variety of constructs that help describe a real application scenario. A widely used semantic model called the entity-relationship (ER) model allows us to pictorially denote entities and relations among them.

The Relational Model

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

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

Student(sid : string, name :string, login :string Age :integer, gpa :real)

The preceding schema says that each record in the student relation has five fields, with field names and types as indicated. An example instance of the students relation appears in fig 1.1.

Sid	Name	Login	Age	Gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

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

Levels of Abstraction

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

It must retrieve data efficiently. This concern led to the design of the complex data structure for the representation of data in the database. Through different levels of abstraction, to simplify the user interaction with system:

Physical Level: The lowest level of abstraction describes how the data are actually stored. At the Physical level, complex low level data structures are described in detail.

Logical Level: The next higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The entire database is thus described in terms of a small number of relatively simple structures. The logical level of abstraction is used by database administrators, who decide what information is to kept in the database.

View Level: The highest level of abstraction describes only part of the entire database. Despite the use of simple structures at the logical level, some complexity remains, because of the large size of the database. so that the interaction with the system is simplified, the view level of abstraction is defined. The system may provide many views for the same database.

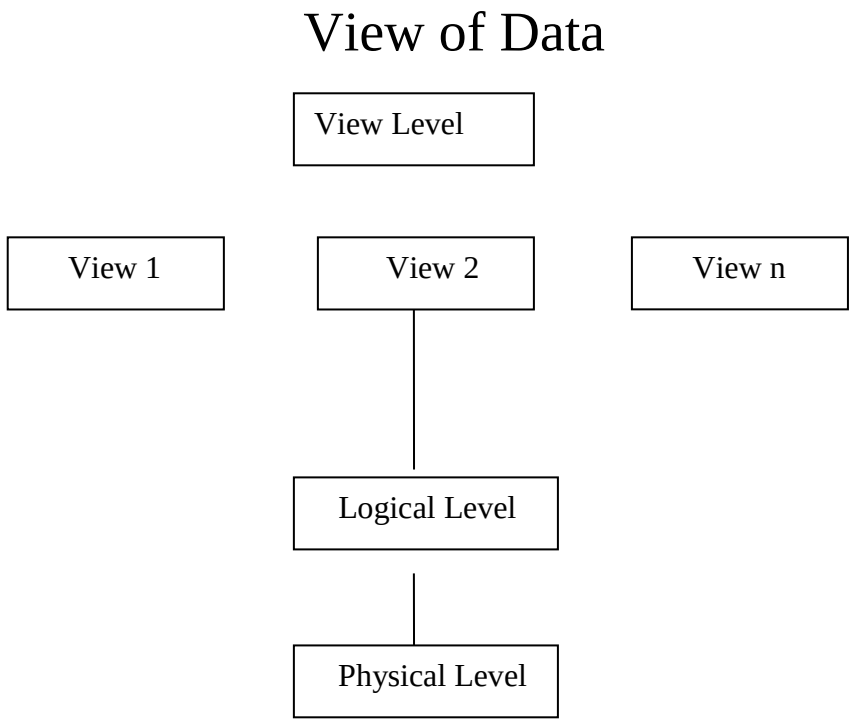


Fig. 1.1 The three levels of data abstraction

Most high level programming languages support the notion of a record type. Fro example, in a Pascal like language, we may declare a record as follows:

```
Type customer  = record
                   Customer-name  : string
                   Social-security : string
                   Customer-street : string
                   Customer-city : string
                 End;
```

This code defines a new record called customer with three fields. Each field has a name and a type associated with it.

A physical level, a customer, account, or employee record can be described as a book of consecutive storage locations.

At the logical level, each such a record is described by a type definition, as illustrated I the previous code segment, and the interrelationship among these recode type is defined.

At the view level, computer users see a set of application programs that hide details of the data types. Similarly, at the view level, several views of the database are defined, and database users see these views. The view can also provide a security mechanism to prevent users from accessing parts of the database.

Instance and Schemas

The collection of information stored in the database at a particular moment is called an instance of the database. The overall design of the database is called the database schema. Schemas are changed infrequently. At the lowest level is the physical schema; at the intermediate level is the logical schema; and at the highest level is a subschema. In general database systems support on e physical schema, one logical schema and several subschemas.

Data Independence

The ability to modify a schema definition in one level without affecting a schema definition in the next higher level is called data independence. There are two levels of data independence:

1. Physical data Independence is the ability to modify the physical definition without causing application program to be rewritten. Modifications at the physical level are occasionally necessary to improve performance.

2. Logical Data Independence is the ability to modify the logical schema without causing application programs to be rewritten. Modifications at the logical level are necessary whenever the logical structure of the database is altered.

The concept of data independence is similar in many respects to the concept of abstract data types in modern programming languages.

1.5 Structure of DBMS

The DBMS accepts SQL commands from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers.

Whenever a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a blueprint for evaluating query, usually represented as a tree of relational operators. Relational operators serve as the building blocks for evaluating queries posed against the data.

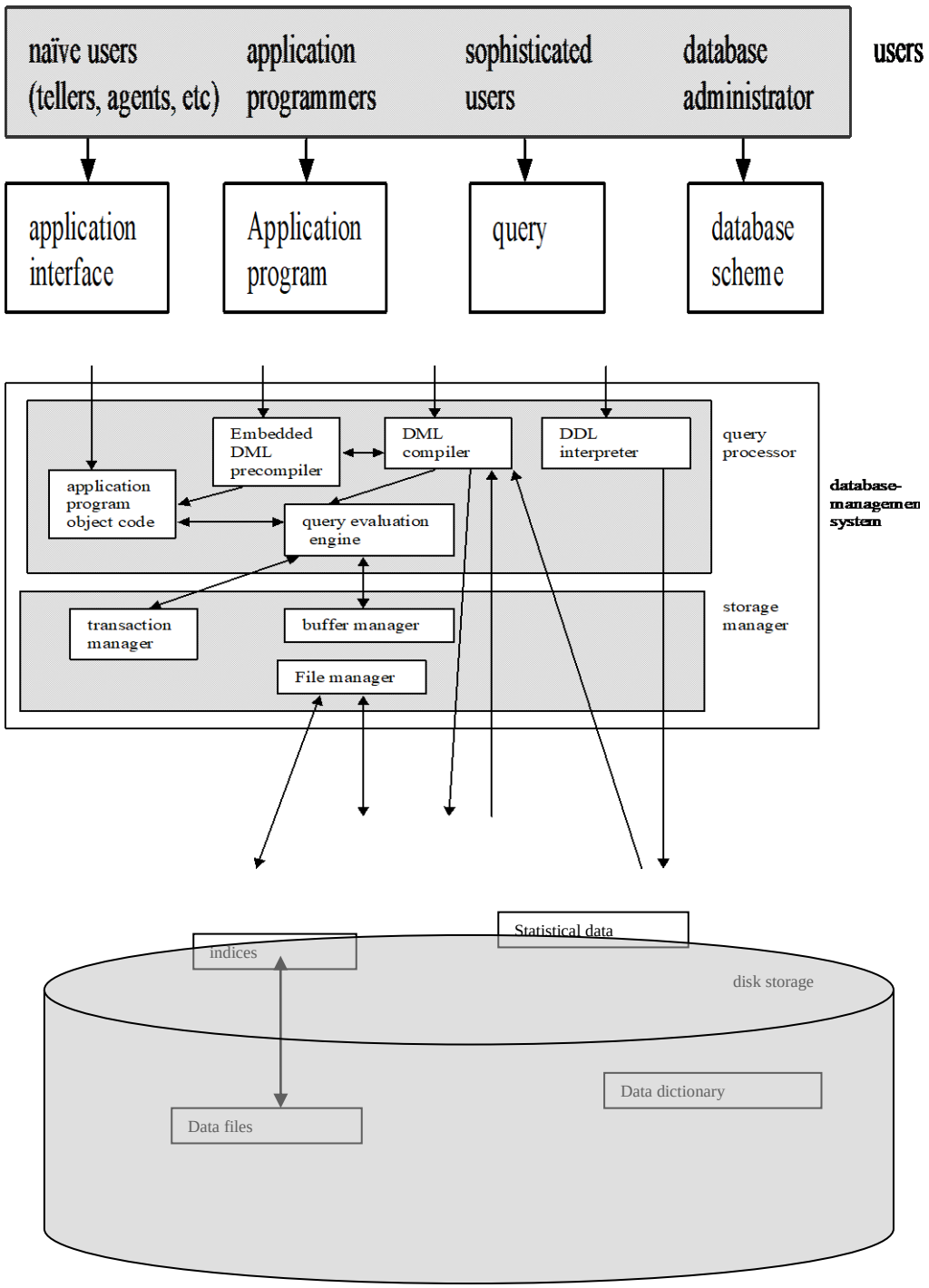
The code that implements relational operators sits on the top of the file and access methods layer. This layer supports the concept of a file, which in a DBMS, is a collection of pages or a collection of records. **Heap files**, or files of unordered pages, s well as indexes are supported. In addition to keeping track of the pages in a file, this layer organizes the information within a page.

The files and access methods layer sits on the top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests.

The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through this layer called **disk space manager**.

The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database.

DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on



database objects when they become available; and the **recovery manager**, which is responsible for maintaining a log and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components.

Self-Assesment Questions – I

1. Expansion of DBMS _____.
2. Database is a collection of _____.
3. Concurrent access is _____.
4. Transaction is an _____.
5. Expansionof DDL is
    a) Data Description Language
    b) Data Definition Language
    c) Data Derived Language
6. SQL stands for
    a) Structured Query Language
    b) Standard Query Language
    c) Standard Quality Language

Sample questions

1. Write notes about File systems Vs DBMS.
2. Write down the advantages of a DBMS.
3. Explain different levels of abstraction in DBMS?
4. Explain the Structure of DBMS?

Answers for Self-Assesment Questions – I

1. Database Management System
2. Data
3. Simultaneous Access
4. Execution of a user program
5. b – Data Definition Language
6. a – Structured Query Language

NOTES

2. 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. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed, precise description that can be implemented in a DBMS.

2.1 Introduction to ER Model

The database design process can be divided in to sis steps. The ER model is most relevant to the first three steps.

1. Requirement Analysis.

   The very first step in designing a database application is to understand shat data is to be stored in the database, ahat applications must be built on top of it, and what operations are most frequent and subject to performance requirements.

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 known to hold over this data. This step is often carried out using the ER model. The ER model is one of several high-level, or semantic, data models used in database design. The goal is to create a simple description of the data closely matches how users and developers think of the data and the people and processes to be represented in the data.

3. Logical 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 consider only relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

4. Schema Refinement

   The fourth step in database design is to analyze the collection of relations in our relational schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory.

5. Physical Database Design

   In this step, we consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema.

6. Application and Security Design

   Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. Design methodologies like UML try to address the complete software design and development cycle. Briefly, we must identify the entities and processes involved in the application. We must describe the role of each entity in every process that is reflected in some application task, as part of a complete work flow for this task.

   An entity is an object in the real world that is distinguishable from other objects; simply it is called as records. An entity is described using a set of attributes. All entities in a given entity set have the same attributes. A domain is a possible set of values of a particular attribute. A key is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key, if so, we designate one of them as the primary key. For now we assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set. That is, the set of attributes contains a key.

   A relationship is an association among two or more entities. Collection of set of similar relationships is known as relationship set.

2.2 Conceptual Design With the ER Model

Develop 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?

We now discuss the issues involved in making these choices.

1. Entity versus Attributes

   While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as entity set. Fro example, consider adding address information to the Employee entity set. Entity is nothing but a record and an attribute is a fields defined in the relation.

2. Entity versus Relationship

   The imprecise nature of ER modeling can thus make it difficult to recognize underlying entities, and we might associate attributes with relationships rather than the appropriate entities. In general, such mistakes lead to redundant storage of the same information and can cause many problems.

3. Binary versus Ternary Relationships

   There are situations, however, where a relationship inherently associates between two entities, is called as Binary relationship.

   There are situations, however, where a relationship inherently associates more than two entities, is called as Ternary relationship.

4. Aggregation versus Ternary Relationships

   Aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set. The choice may also be guided by certain integrity constraints that we want to express.

Self-Assesment Questions – II

1. Expansion of ER model _____.
2. Entity is an _____.
3. Relatinship is an _____.
4. Constraint is an _____.
5. Expansionof UML is
   a. Uniform Markup Language
   b. Unified Markup Language
   c. Unified Modeling Language

Sample questions

6. Write notes Entities, Attributes and Entity Sets.
7. Write down the conceptual ER model.
8. Explain Unified Modeling Language?
9. Explain Aggregation?

Answers for Self-Assesment Questions– II

1. Entity Relationship model
2. Object
3. Association among entities
4. Condition
5. c – Unified Modeling Language

NOTES

3. The Relational Model

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 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 other data models are its simple data representation and the ease with which even complex queries can be expressed.

3.1 Introduction to Relational Model

The main construct for representing data in the data model is a relation. A relation consists of 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, 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, it 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. 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 taught of as a table in which each tuple is a row, and all rows have the same number of fields.

As instance of the students relation appears in the following figure. The instance S1 contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. That is a requirement of the relational model – each relations defined to be a set of unique tuples or rows.

Field names FIELDS(ATTRIBUTES, COLUMNS)

Tuples (RECORDS, ROWS)

Sid	Name	Login	Age	Gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

fig 3.1 An instance S1 of the students relation

The following figure illustrates the same relation instance. If the fields are named, as in our schema definitions and figures depicting relation instances, the order of fields does not matte either. However, an alternative convention is to list of fields in a specific order and refer to a field by its position.

Sid	Name	Login	Age	Gpa
53832	Guldu	guldu@music	12	2.0
53831	Madayan	madayan@music	11	1.8
53688	Smith	smith@ee	18	3.2
53666	Jones	jones@cs	18	3.4
53650	Smith	smith@math	19	3.8
50000	Dave	dave@cs	19	3.3

Fig 3.2 An alternative representation of instance S1 of studetns

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

More formlly, let R(f_1:D1,….f_n:Dn) be a relation schema, and for each fi, $1 \leq i \leq n$, let Dom_i be the set of values associated with the domain named Di. An instance of R that satisfies the domain constraints in the schema is a set of tuples with n fields:

$$\{ <f_1 : d_1, \ldots, f_n : d_n> \mid d_1 \in Dom_1, \ldots, d_n \in Dom_n \}$$

The relational database is a collection of relations with distinct relation names. The relational database schema is the collection of schemas for the relations in the database.

Creating and Modifying Relations using SQL

The SQL language standard uses the word table to denote relation, and we often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the data definition language (DDL). Further, while there is a command that lets users define new domains, analogous to type definition commands in a programming language,

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

```
CREATE TABLE Students( Sid CHAR(20),
                       Name CHAR(30),
                       Login CHAR(20),
                       Age INTEGER,
                       Gpa REAL)
```

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

```
INSERT
INTO students (sid, name, login, age, gpa)
VALUES (53688, 'smith', 'smith@ee', 18, 3.2)
```

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

We can delete tuples using the DELETE command. We can delete all students tuples with name equal to smith using the command:

```
DELETE
FROM students S
WHERE S.name = 'smith'
```

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

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

The WHERE clause is applied first and determines which rows are to be modified. The SET clause then determines how these rows are to be modified. To illustrate these points further, consider the following variation of the previous query:

```
UPDATE students S
SET S.gpa = S.gpa = 0.1
WHER S.gpa >= 3.3
```

If this query is applied on the instance S! of students shown in the previous figure, we obtain the instance shown in the following figure.

Sid	Name	Login	Age	Gpa
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Fig 3.3 Students instance S1 after Update

3.2 Integrity Constraints over Relations

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

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. It is important to specify exactly when integrity constraints are checked relative to the statement that causes the change in the data and the transaction that it is part of. In this chapter different types of constraints.

Key Constraints

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

There are two parts of candidate key definition:

1. The distinct tuples in a legal instance cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

The first part of the definition means that, in any legal instance, the values in the key fields uniquely identify a tuple in the instance. When

specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a correct set of tuples.

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

Specifying Key Constraints in SQL

In SQL, we can declare that a subset of the columns of a table constitute a key by using the UNIQUE constraint. At most one of these candidate keys can be declared to be a primary key, using the PRIMARY KEY constraint.

Let us revisit our example table definition and specify key information

```
CREATE TABLE Students( Sid CHAR(20),
                       Name CHAR(30),
                       Login CHAR(20),
                       Age INTEGER,
                       Gpa REAL,
                       UNIQUE (name, age),
                       CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

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

Foreign Key Constraints

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

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

Enrolled(studid: string, cid: string, grade: string)

To ensure that only bonafide students can enroll in courses, any value that appears in the studid field of an instance of the enrolled relation should also appear in the sid field of some tuple in the students relation. The studid field of enrolled is called foreign key and refers to students. The foreign key in the referencing relation must match the primary key of the referenced relation; that is, it must have the same number of columns and compatible data types, although the column names can be different.

Foreign Key

cid	Grade	Studid
Carnatic101	C	53831
Reggae203	B	53832
Topology112	A	53650
History105	B	53666

Primary Key

Enrolled(Referencing relation)
Students(Referenced relation)

Sid	Name	Login	Age	Gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Fig 3.4 Referential integrity

This constraint is illustrated in the above figure. As the figure shows, there may well be some students tuples that are not referenced from enrolled. However, every studid value that appers in the instance of the enrolled table appears in the primary key column of a row in the students table.

If we try to insert the tuple (55555, Art104, A) into E1, the Ic is violated because there is no tuple in S1with sid 55555; the database system should reject such an insertion. Similarly, if we delete the tuple (53666, jones, jones@cs, 18, 3.4) from S1, we violate the foreign key constraint because the tuple (53666, History105, B) in E1 contains studid value 53666, the sid of the deleted students tuple. The DBMS should disallow the deletion or, perhaps also delete the enrolled tuple that refers to the deleted students tuple.

Specifying Foreign Key Constraints in SQL

Let us define enrolled(studid: string, cid: string, grade: string):

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

The foreign key constraint states that every studid value in enrolled must also appear in students, that is, studid in enrolled is a foreign key referencing students. Specifically, every studid value in ecrolled must appear as the value in the primary key field, sid of students.

General Constraints

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

For example, we may require that student ages be within a certain range of values; given such an IC specification, the DBMS rejects inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of students shown in the figure 3.1 is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in the following figure.

Sid	Name	Login	Age	Gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Fig 3.5 An instance S2 of the student relation

Current relational database systems support such general constraints in the form of table constraints and assertions. Table constraints are associated with a single table and checked whenever that table is modified. In contrast, assertions involve several tables and are checked whenever any of these tables is modified.

3.3 Introduction to Views

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

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

The view B-students has three fields called name, sid, and course with the same domains as the fields sname and sid in students and cid in enrolled.

name	sid	course
Jones	53666	History105
Guldu	53832	Reggae203

Fig 3.6 An instance of the B-students view

Views, Data Independence, Security

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

Views are also valuable in the context of security: we can define views that give a group of users access to just the information they are allowed to see.

Updates on Views

User can be allowed to updates the information in the views is as follows.

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

We can implement a command to modify the gpa of a good students row by modifying the corresponding row in students. We can delete a goodstudetns row by deleting the corresponding row from students. We can insert goodstudents row by inserting a row into students, using null values in columns of students that do not appear in goodstudents. An INSERT or DELETE may change the underlying base table so that the resulting row is not in the view.

Need to Restrict View Updates

While SQL rules on updatable views are more stringent than necessary, there are some fundamental problems with updates specified on views and good reason to limit the class of views that can be updated.consider the students relation and a new relation called clubs:

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

A tuple in clubs denotes that the student called mnme has been a member of the club cnamesince the date jyear. Suppose that we are often

interested in finding the names and logins of students with a gpa greater than 3 who belong to at least one club, along with the club name and the date they joined the club. We can define a view for this purpose:

CREATE VIEW activestudents (name, login, club, since)

AS SELECT S.sname, S.login, C.cname, C.jyear

FROM S.Sname = C.mname AND S.gpa > 3

Consider the instances of students and clubs shown in the following figures 3.7 and 3.8. when evaluated using the instances C and S3, activestudents contains the rows ahown in figure 3.9.

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

Fig 3.7 An instance C of Clubs

Sid	Name	Login	Age	Gpa
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7

Fig 3.8 An instance S3 of Students

Name	Login	club	since
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997
Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

Fig 3.9 Instance of Activestudents

3.4 Destroying / Altering Tables and Views

If we decide that we no longer need a base table and wan to destroy it, we can use the DROP TABLE command. For example, DROP TABLE students destroys the students table unless some view or integrity constraint refers to students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE, students is dropped and any referencing views or

integrity constraints are dropped as well; one of these two keywords must always be specified. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

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

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

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

Self-Assesment Questions – III

1. A relation consists of a _____and _____.
2. Primary Key is used to _____.
3. Unique key is used to check _____.
4. INSERT is used to _____.
5. Referential integrity can be implemented through
    a. Primary Key
    b. Unique
    c. Foreign Key
6. Table can be create by using the query
    a. Create table
    b. Alter table
    c. Drop table

Sample questions

7. Write notes about integrity constraints.
8. Write down the querying relational data.
9. Explain Foreign key constraints?
10. Explain the concept of view?

Answers for Self-Assesment Questions – III

1. Relation Schema, Relation instance
2. Identify the individual record
3. Both Primary Key and Not Null
4. Insert the tuples into the relation
5. c– Foreign Key
6. a – Create table

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

4. Relational Algebra and Calculus

Query languages are specialized languages for asking questions, or queries that involve the data in a database. The inputs and outputs of a query are relations. A query is evaluated using instances of each input relation and it produces an instance of the output relation.

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

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

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

Sailors (sid: integer, sname: string, rating: integer, age: real)

Boats (bid: integer, bname: string, color: string)

Reserves(sid: integer, bid: integer, day: date)

In several examples illustrating the relational algebra operators, we use the instances S1 and S2 (of sailors) and R1 (of Reserves) shown in the following figures 4.1, 4.2, and 4.3 respectively.

Sid	Sname	Rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Fig 4.1 Instance S1 of sailors

Sid	Sname	Rating	age
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	55.5
58	Rusty	10	35.0

Fig 4.2 Instance S2 of Sailors

Sid	Bid	day
22	101	10/10/96
58	103	11/12/96

Fig 4.3 instance of R1 of Reserves

a. 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 relation instances as arguments and returns a relation instance as the result. This property makes it easy to compose operators to form a complex query – a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators.

1. Selections and Projection

Relational Algebra includes operators to select rows from a relation (σ) and project columns (∏. These operations allow us to manipulate data in a single relation. Consider the instance of the sailors relation shown in the fog 4.2 denoted as S2. we can retrieve rows corresponding to expert sailors

by using the σ operator. The expression

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

evaluates to the relation shoen in the fig 4.4. the subscript rating>8 specifies the relation criterion to be applied while retrieving tuples.

Sid	Sname	Rating	age
28	Yuppy	9	35.0
58	Rusty	10	35.0

Sname	Rating
Yuppy	9
Lubber	8
guppy	5
Rusty	10

Fig. 4.4 $\sigma_{rating>8}(S2)$ Fig.4.5$\prod_{sname,rating}(S2)$

The selection operator σ specifies the tuples to retain through a selection condition. In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives ^ and v) of terms that have the form attribute op constant or attribute1 op attribute2, where op is one of the comparison operators <,<=,=,≠,>=, or >. The reference to an attribute can be by

position (of the form .i or i) or by name. the schema of the result of a selection is the schema of the input relation instance.

The projection operator ∏ allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using ∏. The expression

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

evaluates to the relation shown in fig 4.5. the subscript sname, rating specifies the fields to be retained; the other fields are projected out. The schema of the result of a projection is determined by the fields that are projected in the obvious way. Suppose that we wanted to find out only the ages of sailors. The expression

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

evaluates to the relation shown in figure 4.6. the important point to note is that, although three sailors are aged 35, a single tuple with age=35.0 appears in the result of the projection. This follows from the definition of a relation as a set of tuples. In practices, real systems often omit the expensive stop of eliminating duplicate tuples, leading to relations that are multisites.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. Fro example, we can compute the names and ratings of highly rated sailors by combaining two of the proceeding queries. The expression

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

produces the result shown in fig. 4.7. it is obtained by applying the selection to S2 and then applying the projection

Age
35.0
55.0

Fig 4.6. $\prod_{age}(S2)$

Sname	Rating
Yuppy	9
Rusty	10

Fig. 4.7$\prod_{sname,rating}(\sigma_{rating>8}(S2)$

Set Operations

The following standard operations on sets are also available in relational algebra: union (u), intersection (n), set difference (-), and cross-product (x).

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

Two relation instances are said to be union-compatible if the following conditions hold:

- they have the same number of the fields and
- corresponding fields, taken in order from left to right, have the same domains.

- **Intersection:** RnS returns a relation instance containing all tuples that occur in both relations R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.
- **Set-diffrence:** R-S returns a relation instance containing all tuples that occur in the relation R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.
- **Cross-product:** RxS returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). the result of RxS contains one tuple (r, s) (the concatenation of tuples in r and s) for each pair of tuples r r, s S. the cross-product in operation is sometimes called **Cartesian-product**.

We use the convention that the fields of RxS inherit names from the corresponding fields of R and S. it is possible for both R and S to contain one or more fields having the same name; this situation creates a naming conflict. The corresponding fields in RxS solely by position.

In the preceding definitions, note that each operator can be applied to relation instances that are computed using a relational algebra expression.

We now illustrate these definitions through several examples. The union of S1 and S2 is shown in fig. 4.8. fields are listed in order; field names are also inherited from S1. S2 has the same field names, of course, since it is also an instance of sailors. In general, fields of S2 may have different names; recall that we require only domains to match. Note that the result is a set of tuples. Tuples that appear in both S1 and S2 appear only once in S1uS2. also, s!uS2 is not a valid operation because the two relations are not union-compatible. The intersection of S1 and S2 is shown in Fig.4.9, and the set-difference S1-S2 is shown in Fig. 4.10.

Sid	Sname	Rating	age
22	Dustin	7	45.0
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	55.5
58	Rusty	10	35.0

Fig. 4.8 S1 U S2

Sid	Sname	Rating	age
31	Lubber	8	55.5
58	Rusty	10	35.0

Fig. 4.9 S1 n S2

Sid	Sname	Rating	age
22	Dustin	7	45.0

Fig. 4.10 S1 - S2

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

Fig. 4.11 S1 X S2

The result of the cross-product S1xR1 is shown in fig.4.11. because R1 and S1 both have a field named sid, by our convention on field names, the corresponding two fields in S1xR1 are unnamed, and referred to solely by the position in which they appear in Fig 4.11. The fields in S1xR1 have the same domains as the corresponding fields in R1 and S1. In figure 4.11, sid is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

Renaming

A renaming operator ρ for this purpose. The expression takes an arbitary relational algebra expression E and returns an instance of a relation called . R contains the same tuples as the result of E and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the renaming list F, which is a list of terms having the form old name -> new name or position -> new name. For ρ to be well defined, references to fields (in the form of old names or positions in the renaming list) may be unambiguous and no two fields in the result may have the same name. Sometimes we want to only rename fields in the result may have the relation; we therefore treat both R and F as optional in the use of ρ. (Of course, it meaningless to omit both.)

For example, the expression ρ(C(1-> sid1, 5->sid2), S1xR1) returns a relation that contains the tuples shown in fig. 4.11 and has the following schema: C(sid: integer, sname: string, rating: integer, age: real, sid2:integer, bid: integer, day:dates).

Joins

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

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:

$$R \; S = \sigma_c(RxS)$$

This is defined to be a cross-product followed by a selection. Note that the condition c can refer to attributes of both R and S. The reference to an attribute of a relation, say R, can be by position or by name.

An example, the result of S1 R1 is shown in fig.4.12. because sid appears in both S1 and R1. The corresponding fields in the result of the cross-product S1xR1 are unnamed. Domains are inherited from the corresponding fields of S1 and R1.

Equijoin

A common special case of the join operation R S is when the join condition consists solely of equalities 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 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 R.sid=S.sid R1 in the following fig. 4.13. Note that only one field called sid appears in the result.

Sid	Sname	Rating	age	Sid	Bid	Day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Fig. 4.12 S1 X $_{S1.sid<R1.sid}$ R1

Sid	sname	rating	Age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Fig. 4.13 S1 X $_{R.sid=S.sid}$ R1

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 actually a natural join and can simply be denoted as S! R1, since the only common field as sid. If the two relations have attributes in common, S! R1 is simply the cross-product.

Division

The division operator is useful for expressing certain kinds of example, "Find the names of sailors who have reserved all boats." Understanding how to use the basic operators of the algebra to define division is a useful exercise, however, the division operator does not have the same importance as the other operators-it is not needed as often, a database systems do not try to exploit the semantics of division by implementing it as a distinct operator.

Consider two relation instances A and B in which A has two fields x and y and B has just one field y, with the same domain as in A. we define the division operation A/B as the set of all x values such that for every y value in B, there is a tuple (x,y) in A.

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

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

A

Sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

A/B1

sno
S1
S2
S3
S4

B2

pno
p2
p4

A/B2

sno
S1
S4

B3

pno
p1
P2
p4

A/B3

sno
s1

Fig.4.14. Division operation Examples

To understand the division operation in full generality, we have to consuider the case when both x and y are replaced by a set of attributes. The generalization is straight forward and left as an exercise for the reader.

4.2 Relational Calculus

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

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

range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE.

Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relational 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 | p(T) }, where T is a tuple variable and p(T) denotes a fomula that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula p(T) evaluates to true with T=t. The language for writing for writing formulas p(T) is thus at the heart of TRC and essentially a simple subset of first-order logic. As a simple example, consider the following query.

Find all sailors with a rating above 7.

{ S | S Є sailors ^ S./rating > 7 }

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

Syntax of TRC Queries

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

- RЄ Rel
- R.a op S.b
- R.a op constant, or constant op R.a

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

- Any atomic formula
- P, p^q, pvq, or p➔q
- R(p(R)), where R is a tuple variable
- R(p(R)), where R is a tuple variable

In the last two clauses, the quantifiers and are said to bind variable R. A variable is said to be free in a formula or subformula if the formula does not contain an occurrence of a quantifier that binds it.

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

Semantics of Queries

TRC query {T|p(T)}, as denoted earlier, is the set of all tuples t for which the formula p(T) evaluates to true with variable T assigned the tuple

value t. to complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to free.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to true if one of the following jolds:

- F is an atomic formula R€ Rel, and R is assigned a tuple in the instance of relation Rel.
- F is a comparision R.a op S.b, R.a op constant, or constant op R.a, and the tuples assigned to R and S have field values R.a and S.b that make the comparison true.
- F is the form p and q is not true, or of the form p^q, and both p and q are true, or if the form pvq and one of them is true, or of the form p➔q and q is true whenever p is true.
- Fis the form R(p(R)), and there is some assignment of tuples to the free variables in p(R), including the variable R, that makes the formula p® true.
- F is the form R(p(R)), and there is some assignment of tuples to the free variables in p(R) that makes the formula p(R) true no mater what tuple is assigned to R.

Domain Relational Calculus

A domain variable is a variable that ranges over the values in the domain of some attribute. A DRC query has the form { <x1,x2,…xn> ¦ p(<x1,x2,….xn>) }, where each xi is either a domain variable or a constant and p(<x1,x2,….xn>) denotes a DRC formula whose only free variable are the variables among <x1,x2,….xn> for which the formula evaluates to true.

A DRC formula is defined in a manner 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 {<,>,=,≠,≤,≥} and let X and Y be domain variables. An atomic formula in DRC is one of the following:

- <x1,x2,….xn> € Rel, where Rel is a relation with n attributes; each xi 1<= I <= n is either a variable or a constant.
- X op Y
- X op constant or constant op X

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

- Any atomic formula
- ¬P, p n q, pVq, or p➔ q
- E X(P(X)), where X is a domain variable
- V X(P(X)), where X is a domain variable

The reader is invited to compare this definition with the definition of TRC formulas and see how closely these two definitions correspond. We will not define the semantics of DRC formulas formally.

Set-theoretic formulation

Basic notions in the relational model are *relation names* and *attribute names*. We will represent these as strings such as "Person" and "name" and we will usually use the variables $r, s, t, \ldots$ and *a,b,c* to range over them. Another basic notion is the set of *atomic values* that contains values such as numbers and strings.

Our first definition concerns the notion of *tuple*, which formalizes the notion of row or record in a table:

Tuple

A tuple is a partial function from attribute names to atomic values.

Header

A header is a finite set of attribute names.

Projection

The projection of a tuple *t* on a finite set of attributes *A* is $t[A] = \{(a, v) : (a, v) \in t, a \in A\}$.

The next definition defines *relation* which formalizes the contents of a table as it is defined in the relational model.

Relation

A relation is a tuple (*H*,*B*) with *H*, the header, and *B*, the body, a set of tuples that all have the domain *H*.

Such a relation closely corresponds to what is usually called the extension of a predicate in first-order logic except that here we identify the places in the predicate with attribute names. Usually in the relational model a database schema is said to consist of a set of relation names, the headers that are associated with these names and the constraints that should hold for every instance of the database schema.

Relation universe

A relation universe *U* over a header *H* is a non-empty set of relations with header *H*.

Relation schema

A relation schema (*H*,*C*) consists of a header *H* and a predicate *C*(*R*) that is defined for all relations *R* with header *H*. A relation satisfies a relation schema (*H*,*C*) if it has header *H* and satisfies *C*.

[edit] Key constraints and functional dependencies

One of the simplest and most important types of relation constraints is the *key constraint*. It tells us that in every instance of a certain relational schema the tuples can be identified by their values for certain attributes.

Superkey

A superkey is written as a finite set of attribute names.

A superkey *K* holds in a relation (*H*,*B*) if:

- $K \subseteq H$ and
- there exist no two distinct tuples $t_1, t_2 \in B$ such that $t_1[K] = t_2[K]$.

A superkey holds in a relation universe *U* if it holds in all relations in *U*.

Theorem: A superkey *K* holds in a relation universe *U* over *H* if and only if $K \subseteq H$ and $K \rightarrow H$ holds in *U*.

Candidate key

A superkey *K* holds as a candidate key for a relation universe *U* if it holds as a superkey for *U* and there is no proper subset of *K* that also holds as a superkey for *U*.

Functional dependency

A functional dependency (FD for short) is written as $X \rightarrow Y$ for *X*,*Y* finite sets of attribute names.

A functional dependency $X \rightarrow Y$ holds in a relation (*H*,*B*) if:

- $X, Y \subseteq H$ and
- $\forall$ tuples $t_1, t_2 \in B$, $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$

A functional dependency $X \rightarrow Y$ holds in a relation universe *U* if it holds in all relations in *U*.

Trivial functional dependency

A functional dependency is trivial under a header *H* if it holds in all relation universes over *H*.

Theorem: An FD $X \rightarrow Y$ is trivial under a header *H* if and only if $Y \subseteq X \subseteq H$.

Closure

Armstrong's axioms: The closure of a set of FDs *S* under a header *H*, written as S^+, is the smallest superset of *S* such that:

- $Y \subseteq X \subseteq H \Rightarrow X \rightarrow Y \in S^+$ (reflexivity)
- $X \rightarrow Y \in S^+ \wedge Y \rightarrow Z \in S^+ \Rightarrow X \rightarrow Z \in S^+$ (transitivity) and
- $X \rightarrow Y \in S^+ \wedge Z \subseteq H \Rightarrow (X \cup Z) \rightarrow (Y \cup Z) \in S^+$ (augmentation)

Theorem: Armstrong's axioms are sound and complete; given a header *H* and a set *S* of FDs that only contain subsets of *H*, $X \rightarrow Y \in S^+$ if and only if $X \rightarrow Y$ holds in all relation universes over *H* in which all FDs in *S* hold.

Completion

The completion of a finite set of attributes *X* under a finite set of FDs *S*, written as X^+, is the smallest superset of *X* such that:

- $Y \rightarrow Z \in S \wedge Y \subseteq X^+ \Rightarrow Z \subseteq X$

The completion of an attribute set can be used to compute if a certain dependency is in the closure of a set of FDs.

Theorem: Given a set *S* of FDs, $X \rightarrow Y \in S^+$ if and only if $Y \subseteq X^+$.

Irreducible cover

An irreducible cover of a set *S* of FDs is a set *T* of FDs such that:

- $S^+ = T^+$
- there exists no $U \subset T$ such that $S^+ = U^+$
- $X \rightarrow Y \in T \Rightarrow Y$ is a singleton set and
- $X \rightarrow Y \in T \wedge Z \subset X \Rightarrow Z \rightarrow Y \notin S^+$.

[edit] Algorithm to derive candidate keys from functional dependencies

INPUT: a set *S* of FDs that contain only subsets of a header *H*

OUTPUT: the set *C* of superkeys that hold as candidate keys in all relation universes over *H* in which all FDs in *S* hold

```
begin
  C := ∅;          // found candidate keys
  Q := { H };      // superkeys that contain candidate keys
  while Q <> ∅ do
    let K be some element from Q;
    Q := Q - { K };
    minimal := true;
    for each X->Y in S do
      K' := (K - Y) ∪ X;   // derive new superkey
      if K' ⊂ K then
        minimal := false;
        Q := Q ∪ { K' };
      end if
    end for
    if minimal and there is not a subset of K in C then
      remove all supersets of K from C;
      C := C ∪ { K };
    end if
  end while
end
```

Self-Assesment Questions – IV

1. Union operation will _____.
2. combine two or more elations is known as _____.
3. Expansion of TRC is _____.
4. Expansion of DRC is _____.
5. Relational Algebra is a
    a) Programming Language
    b) Communication Language
    c) Query Language
6. Symbol of cross product is
    a) U
    b) n
    c) x

Sample questions

7. Write notes about Relational Algebra.
8. Write down the Tuple Relational Calculus.
9. Explain different types of joins?
10. Explain Domain Relational Calculus?

Answers for Self-Assesment Questions – IV

1. combine two sets
2. Join
3. Tuple Relational Calculus
4. Domain Relational Calculus
5. c – query Language
6. c – x

NOTES

5. SQL: Queries, Constraints, Triggers

Structured Query Language (SQL) is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and system-R projects (1974-1977). Other vendors introduced DBMS products based on SQL, and it is a de facto standard. SQL continues to evolve in response to changing needs in the database area. The current ANSI / ISO standard for SQL id called SQL:1999.

SQL language has several aspects to it.

- The Data Definition Language (DDL) : This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. Integrity constraints can be defined on tables, either when the table is created or later.
- The Data Manipulation Language (DML) : This subset of SQL allows users to pose queries and to insert, delete, and modify rows.
- Triggers and Advanced Integrity Constraints: the new SQL:1999 standard includes support for triggers, which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.
- Embedded and Dynamic SQL : Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dyanmic SQL allow a query to be constructedat run-time.
- Client-Server Execution and Remote Database Access: These commands control how a client application can connect to an SQL database server, or access data from a database over a network.
- Transaction Control Language: Various commands allow a user to explicitly control aspects of how a transaction is to be executed.
- Security: SQL provides mechanisms to control users access to data objects such as tables and views.
- Advanced features : The SQL:1999 standard includes object-oriented features, recursive queries, decision support queries, and also addresses areas such as data mining, spatial data, and text and XML, data management.

Here we will present a number of sample queries using the following table definition.

Sailors(sid:integer, sname:string, rating:integer, age:real)

Boats(bid:integer, bname:string, color:string)

Reserves(sid:integer, bid:integer, day:date)

We illustrate queries using the instances S3 of sailors, R2 of Reserves, and B1 of Boats are reproduced in the following diagrams 5.1, 5.2 and 5.3 respectively.

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

Fig 5.1 Instance S3 of sailors

Sid	Bid	day
22	101	10/10/96
22	102	10/10/98
22	103	10/08/98
22	104	10/07/98
31	102	11/10/98
31	103	11/06/98
31	104	11/12/98
64	101	09/05/98
64	102	09/08/98
74	103	09/08/98

Fig 5.2 Instance of R12of Reserves

bid	name	Color
101	Interlake	Blue
102	Interlake	Red
103	Clipper	Green
104	Marine	Red

Fig 5.3 Instance of B1 of Boats

5.1 The form of a Basic SQL Query

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

The basic form of an SQL query is as follows:

SELECT [DISTINCT] select-list

FROM from-list

WHERE qualification

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. The close relationship between SQL and relational algebra is the basis for query optimization in a relational DBMS.

Let us consider simple examples.

SELECT DISTINCT S.sname, S.age

FROM Sailors S

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

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

Fig. 5.4 Answer to Q15

Sname	age

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

Fig. 5.5 Answer to Q15 without DISTINCT

Our next query is equivalent of the selection operator of relational algebra.

SELECT S.sid, S.sname, S.rating, S.age

FROM Sailors AS S

WHERE S.rating>7

This query uses the optional keyword AS to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides convenient shorthand. We can simply write SELECT *. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained because the schema of the result is not clear from the query itself.

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

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

The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears mare than once in the from-list.

The select-list is a list of column names of tables named in the from-list. Column names can be prefixed by a range variable.

The qualification in the WHERE clause is a Boolean combination (i.e., an expression using the logical connections AND, OR and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<,<=,+,>,>=,<>}. An expression is a column name, a constant, or an expression.

The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated.

Here we describe the meaning of the query.

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

This straight forward conceptual evaluation strategy makes the rows that must be present in the answer to the query. However, it is likely to be quite inefficient.

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

It can be expressed in SQL as follows.

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

Let us compute the answer to this query on the instances R3 of reserves and S4 of Sailors shown in the fig. 5.6 and 5.7, since the computation on our usual example instances would be unnecessarily tedious.

Sid	Bid	day
22	101	10/10/96
58	103	11/12/96

Fig. 5.6 Instance R3 of Reserves

Sid	Sname	Rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Fig. 5.7 Instance S4 of Sailors

The first step is to construct the cross-product S4 X R3, which is shown in the fig. 5.8.

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

Fig. 5.8 S4 X R3

The second step is to apply qualification S.sid = R.sid AND R.bid = 103. this step eliminates all but the last row from the instance shown in figure 5.8. the third step is to eliminate unwanted columns; only sname appears in the SELECT clause. The result shown in fig 5.9.

Sname
Rusty

Fig. 5.9 Answer to query Q1 on R3 and S4.

UNION, INTERSECT, AND EXCEPT

SQL provides three set- manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multi set of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT. SQL also provides other set operations: IN, op ANY, op ALL, op EXISTS.

Consider the following examples.

Q2 Find the names of sailors who have reserved a red or green boat.

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

This query is easily expressed suing the OR connective in the WHERE clause. However the following query, which is identical except for the use of AND rather than ‘OR’ in the English version, turns out to much more difficult:

Q3 Find the names of sailors who have reserved both a red and a green boat.

The query has been written as follows.

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

The previous query is difficult to understand. The same query can be rewritten by using UNION.

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

The following query can returns the sailors who can reserve both green and a red boat.

```
SELECT S.sname
```

FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

INTERSECT

SELECT S2.sname

FROM Sailors S2, Reserves R2, Boats B2

WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’

Q4 Find the sids of all sailors who have reserved red but not green boats.

SELECT S.sname

FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

EXCEPT

SELECT S2.sname

FROM Sailors S2, Reserves R2, Boats B2

WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’

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

Q5 Find all sids of sailors who have a ratting of 10 or reserved boat 104.

SELECT S.sid

FROM Sailors S

WHERE S.rating = 10

UNION

SELECT R.sid

FROM Reserves R

WHERE R.bid = 104

The first part of union returns the sids 58 and 71. the second part returns 22 and 31. the answer is , therefore, the set of sids 22, 31, 58, and 71.

Similarly, we can use UNION ALL, INTERSECT ALL and EXCEPT ALL.

NESTED QUERIES

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it. The embedded query is called a sub query. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. A sub query typically appears within the WHER clause of a query. Sub queries can sometimes appear in the FROM clause or the HAVING clause.

Introduction to Nested Queries

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

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

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

The nested subquery computes the set of sids for sailors who have reserved boat 103, and the top-level query retrieves the names of sailors whose sid is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. Note that it is easy to modify this query to find all sailors who have not reserved boat 103 – we can just replace IN by NOT IN.

An example of multiple nested query is as follows.

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

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

The innermost sub query finds the set of bids of red boats. The sub query one level above finds the set of sids of sailors who have reserved one of these boats. On instances B1, R1 and S3, this set of sids contains 22,31, and 64. the top-level query finds the name of sailors whose sid is in this set of sids; we get Dustin, Lubber and Horatio.

Q3 Find the names of sailors who have not reserved a red boat.

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

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

Correlated Nested Queries

In the nested queries seen thus far, the inner sub query has been completely independent of the outer query. In general, the inner subquery could depend on the row currently being examined in the outer query. Let us rewrite the following query once more.

Q1 Find tha names of sailors who have reserved boat number 103.

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

The EXIST operator is another set comparison operator, such as IN. it allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each sailor row S, we test whether the set of Reserves rows R such that R.bid = 103 AND S.sid = R.sid is nonempty. If so, sailor S has reserved boat 103, and we rewrite the name. The sub query clearly depends on the current row S and must be re-evaluated for each row in sailors. The occurrence of S in the sub query is called a Correlation, and such are called Correlated Queries.

As further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate.

5.5 Aggregated Operators

SQL provide five aggregated operations, which can be applied any column of a relation.

1. COUNT([DISTINCT] A) : The number of (unique) values in the relation.

2. SUM([DISTINCT] A) : The sum of all (unique) values in the A column.

3. AVG([DISTINCT] A) : The average of all (unique) values in the a column.

4. MAX(A) : The maximum value in the A column.

5. MIN(A) : The minimum value in the A column.

Note that it does not make sense to specify DISTINCT in conluction with MIN or MAX.

Find average of all sailors.

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

On instance S3, the average age is 37.4. Of course, the WHERE clause can be used to restrict the sailors considered in computing the average age. Find the average age of sailors with a rating of 10.

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

There are two sailors, and their average age is 25.5. MIN can be used instead of AVG in the above queries to find the age of the youngest. However, finding both the name and the age of the oldest sailor is more tricky, as the next query illustrates.

Find the name and age of the oldest sailor

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

We can be use nested query to compute the desired answer

```
SELECT S.sname, S.age
FROM Sailors S
WHERE  S.age=  (SELECT MAX (S2.age)
                FROM Sailors S1)
```

Observe that we have use the result of an aggregate operation in the sub query as an argument to a comparison operation. The following equivalent query for the above one is legal in the SQL standard but, unfortunately, is not supported in many systems.

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

We can count the number of sailors using the COUNT operation.

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

Count the number of different Sailors name by using DISTINCE

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

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

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

On instance S3, the oldest sailor with rating 10 is sailor 58, whose age is 35. The names of older sailors are both, Dustin, Horatio, and Lubber. Using ALL this query could alternativelt be written as follows:

```
SELECT S.sname
FROM Saiolrs S
WHERE S.age > ALL (SELECT MAX ( S2.age)
```

FROM Sailors S2
WHERE S2.rating=10)

GROUP BY and HAVING Clauses

We want to apply aggregate operations to each of a number for groups of rows in a relation, where the number of groups depends on the relation instance.

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

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

Where i=1,2,…..,10. writing 10 such queries is tedious. To write such a queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause. In fact, the extension also includes an optional HAVING clause that can be used to specify qualifications over groups.

The general form of an SQL query is

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

Set-Comparison Operators

We have already seen the set of set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<,<=,=,<>,>,>=}.

Q1 Find sailors whose rating is better than some sailor clled Horatio.

SELECT S.sid
FROM Sailor S
WHERE S.rating > ANY (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname = “Horation’)

If there are several sailors called Horatio, this finds all sailors whose rating is better than that of some sailor called Horatio. On instance S3, this computes the sids 31,32,58,71, and 74.

Q3 Find sailors whose rating is better than every sailor clled Horatio.

SELECT S.sid
FROM Sailor S
WHERE S.rating > ALL (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname = “Horation’)

This query can returns the result as the sids 58 and 71.

Another example by using ALL is as follows.

Q4 Find the sailors with the highest rating.

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

The sub query computes the se of all rating values in sailors. The outer WHER condition is satisfied only when S.rating is greater than or equal to each of these rating values, that is, when it is the largest rating value. In the instance S3, the condition is satisfied only for rating 10, and the answer includes the sides of sailors with this rating i.e., 58 and 71.

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

NULL Values

Thus far, we have assumed that column in a row are always known. In practice column values can be unknown. For example, when a sailor, says Dan, joins a yacht club, ha may not yet have a rating assigned. Since the definition for the sailors table has a rating column, what row should we insert for Dan?

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

Comparison Using Null Values

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

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

Logical Connections AND, OR, and NOT

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

But this example raises an important point – once we have null values, we must define the logical operators And, OR and NOT using a three-valued

login in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. AND of two argements evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown.

Impact on SQL Constructs

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

Another issue in the presence of null values is the definition of when two rows in a relation instance are regarded as duplicates. The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain null. Contrast this definition with the fact that if we compare two null values using =, the result is unknown in the context of duplicates, this comparison is implicitly related as true. As expected, the arithmetic operations +, -, * , and / all return null if one of their arguments is null. COUNT (*) handles null values just like other values; that is , they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX and DISTINCT) simply discard null values.

Outer joins

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

In fact, there are several variance of the outer join ideas. In a left outer join, sailors rows without a matching reserves row appear in the result, but not vice-versa. In aright outer join, reserves rows without a matching sailors row appear in the result, but not vice-versa. In a full outer join, both sailors and reserves rows without a match appear in the result.

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query list <sid, bid> pairs corresponding to sailors and boats. They have reserved:

SELECT S.sid, R.bid

FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R

The NATURAL keyword specifies that the join condition is equality on all common attributes, and the WHER clause is not required. On the instances of sailors and Reserves shown in the fig 5.6., this query computes the result shown in the fig. 5.10.

Sid	Bid
22	101
31	null
58	103

Fig. 5.10. Left Outer Join of Sailors1 and Reserves1.

Disallowing null values

We can disallow null values by specifying Not Null as part of the field definition; for example, Sname char(20) NOT NULL. In addition, the field in primary are not allowed to take on null values. Thus, there is an implicit not null constraint for every field listed in a PRIMARY KEY constraint. On coverage of null values is far from complete.

Triggers and Active Databases

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

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

A trigger can be thought of as a ‘daemon’ that monitors a database, and is executed when the database is modified in a way that matches the event specification. An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statements.

A condition in a trigger can be true/false statements or a query. A query is interpreted as true, if the answer set is nonempty and false, if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger action can examined the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database.

Example Triggers in SQL

The example shown in the following fig 5.11, written using Oracle server syntax for defining triggers, illustrate the basic concepts behind the triggers.

The trigger called init_count initializes a counter variable before every execution of an INSERT statement that adds tuples to the student relation. The trigger called incr_count increments the counter for each inserted tuple that satisfies the condition age < 15.

```
CREATE TRIGGER  init_count  BEFORE INSERT ON Students
                                                    /* Event */
    DECLARE
        Count INTEGER;
    BEGIN
        Count := 0;                                 /*Action*/

    END

CREATE TRIGGER  incr_count  AFTER INSERT ON Students
                                                    /* Event */
    WHEN  (new.age <15)                         /* Condition */
    FOR EACH ROW
    BEGIN
        Count := count +1;                          /* Action */
    END
```

Fig. 5.11 Examples illustrating Triggers.

Self-Assesment Questions – V

1. From clause is used to _____.
2. Expansion of DML_____.
3. Nested query is_____.
4. SUM function is used to_____.
5. Null value is
    a) Empty
    b) 0
    c) 1
6. Distinct will
    a) Remove the duplicate values
    b) Consider the duplicate values
    c) Remove the different values

Sample questions

7. Write notes about SQL query.
8. Write down Union and intersection.
9. Explain different aggregate functions?
10. Explain Null values and nested queries?
11.

Answers for Self-Assesment Questions – V

1. specify table name
2. Data Manipulation Lnguage
3. Query within query
4. sum all values in a column
5. a – Empty
6. a – Remove the duplicate values

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

6.SCHEMA REFINEMENT AND NORMAL FORMS

6.1Introduction to Schema Refinement

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy.

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 certain information unless some other, unrelated, information is stored as well.
- Deletion Anomalies: It may not be possible to delete certain information without losing some other, unrelated, information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set.

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

It leads to possible redundancy in the relation Hourly_Emps, as illustrated in the following fig.

Ssn	Name	Lot	Rating	Hourly_ wages	Hours_ worked
123-22-3666	Attishoo	48	8	10	40
231-31-5368	smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Fig. 6.1 An instance of the Hourly_Emps Relation

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

- Redundant Storage: The rating value 8 corresponds to the hourly wage 10, and this association is repeated three times.
- Update Anomalies: the hourly-wages in the first tuple could be updated without making a similar in the second tuple.

- Insertion Anomalies: We cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value.
- Deletion Anomalies: If we delete all tuples with a given rating value, we lose the association between that rating and its hourly_wage value.

Decomposition

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

We can decompose Hourly_Emps into two relations:

Hourly_Emps2(snn, name, lot, rating, hours_worked)

Wages(rating, hourly_wages)

The instance of these relations corresponding to the instance of Hourly_Emps relation in the above fig. is shown in the following fig.

Ssn	Name	Lot	Rating	Hourly_ wages	Hours_ worked
123-22-3666	Attishoo	48	8	10	40
231-31-5368	smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Rating	Hourly_ wages
8	10
5	7

Fig.6.2. Instacne of Hourly_Emps2 and Wage

Note that we easily record the hourly wage for any rating simply by adding a tuple to wages, even if no employee with that rating appears in the current instance of Hourly_Emps.

Problems Related to Decomposition

Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problem does a given decomposition cause?

To help with the first question, several normal forms have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given

relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition.

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

6.2 Functional Dependencies

A functional dependency (FD) is a kind of IC that generalizes the concept of a key. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. we say that an instance R satisfies the FD X→ Y if the following holds for every pair of tuples t1 and t2 in r.

If t1.X = t2.X, then t1.Y = t2.Y

We use the notation t1.X to refer the projection of tuple t1, onto the attributes in X, in a natural extension of our TRC notation t.a for referring to attributes of tuple t. an FD X→ Y essentially says that if two tuples agree on the values in attributes X, they must also on the values in attributes Y.

The following fig. illustrates the meaning of the FD AB →C by showing instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint; although the FD is not violated, AB is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the A field or the B field, they can differ in the C field without violating the FD.

A	B	C	D
A1	B1	C1	D1
A1	B1	C1	D2
A1	B2	C2	D1
A2	B1	C3	D1

Fig. 6.3. An instance that satisfies AB → C

6.3 Reasoning about FDs

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

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

We know that ssn→ did holds, since ssn is the key, and FD did→ lot is given to hold. Therefore, in any legal instance of workers, if two tuples have the same ssn value, they must have the same did value, and because

they have the same did value, they must also have the same lot value. Therefore, the FD ssn→lot also holds on windows.

Closure of a set of FDs

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

- Reflection: If X Y, the X → Y.
- Augmentation: If X→Y, then XZ → YZ for any Z.
- Transitivity: If X→Y and Y→Z, then X→Z.

Theorem1

Armstrong's Axioms are sound, in that they generate only FDs in F^+ when applied to a set F of FDs. They are also complete, in that repeated applications of these rules will generate all FDs in the closure F^+.

It is convenient to use some additional rules while reasoning about F^+:

Union: If x→ Y and X→Z, then X→YZ.

Decomposition:If X→YZ, then X→Y and X→Z.

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

Attribute Closure

If we just want to check whether a given dependency, say, X→ Y, is in the closure of a set F of FDs, we can do so efficiently without computing F^+. we first compute the attribute closure X^+ with respect to F, which is the set of attributes a such that X→A can be inferred using the Amstrong Axioms. The algorithm for computing the attribute closure of a set X of attributes is shown in the following fig 6.4.

Closure = X

Repeat until there is no change: {

If there is an FD U→V in F such that U closure,

Then set closure = closure U V

}

Fig. 6.4. Computing the Attribute Closure if Attribute set X

6.4 Normal Forms

The Normal Forms based on FDs are First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form(3NF), Boyce-code 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 also 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 Boyce-code NF are important form a database design standpoint.

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

Boyce-Codd Normal Form

Let R ba a relaion schema, F be the set of FDs given to hold over R, X be a subset of the attributes of R, and A be an attribute of R. R is in Boyce-Codd Normal Form if, for every FD X→A in F, on of the following statements is true:

- A X ; that is, it is a trivial Fd, or
- X is a super key.

Intuitively, in a BCNF relation, the only nontrivial dependencies are those in which a key determines some attributes. Therefore, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Each attribute must describe the key, the whole key, and nothing but the key. If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in the following fig. 6.5.

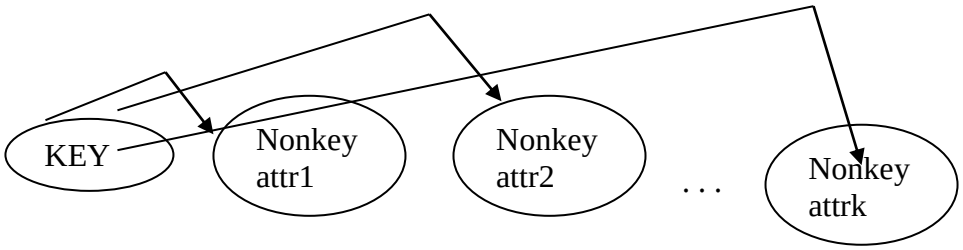


Fig. 6.5 FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form. If we take into account only FD information. This point is illustrated in fig. 6.6.

X	Y	A
x	Y1	A
X	Y2	?

Fig. 6.6. An instance illustrating BCNF

This figure shows an instance of a relation with three attributes X,Y, and A. there are two tuples with the same value in the X column. Now suppose that we know that this instance satisfies an FD X→ A. we can see that one of the tuples has the value a in the column. Therefore, if a elation is in BSNF, every field of every tuple records a piece of information that cannot be inferred from the values in all other fields in the relation instance.

Third Normal Form

Let R be a relation schema, F be the set of FDs given to hold over R, X be a subset of the attributes of r, and A be an attribute of R. R is in third normal form if, for every FD→A in F, one of the following statements is true:

- A X; that is, it is a trivial Fd, or
- X is a supre key, or
- A is part of some key for R.

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. to understand the third condition, recall that a key for a relation is a minimal set of attributes that uniquely determines all other attributes. A must be part of a key. It is not enough for A to be part of a super key, because the later condition is satisfied by every attribute1 finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

Suppose that a dependency X→ A causes a violation of 3NF. there are two case:

- X is a proper subset of some key K. Such a dependency is sometimes called a partial dependency. In this ceas, we store pairs redundancy.
- X is not a proper subset of any key. Such a dependency is sometimes called a transitive dependency, because it mans we have a chain of dependencies K→ X→ A. The problem is that we cannot associate an X value with a K value unless we also associate an A value with an X value.

6.5 Properties of Decompositions

Decomposition is a tool that allows us to eliminate redundancy. It is important to check that a decomposition does not introduce new problems. In particular, we should check whether a decomposition allows us to recover the original, and whether it allows us to check integrity constraints efficiently.

Lossless-Join Decomposition

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

This definition can easily be extended to cover a decomposition of R into more than two relations. In general, though, the other direction does not

hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation.

By replacing the instance r shown in fig. 6.7 with the instances ΠSP(r) and ΠPD(r), we lose some information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that relationships (s1, p1, d3) and (s3, p1, d1) do not hold. The decomposition of schema SPD into SP and PD is therefore lossy if the instance r shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled.

All decomposition used to eliminate redundancy must be loseless. The following simple test is very useful.

Theorem ! : Let r be a relation and F be a set of FDs that hold over R. The decomposition of R into relations with attribute sets R1 and R2 is lossless if and only if F^+ contains either the Fd R1nR2→R1 or the FD R1nR2→R2.

Dependency Preserving Decomposition

Consider the contents relation with attributes CSJDPQV. The given FDs are C→CSJDPQV, JP→C, and SD→P. because Sd is not a key the dependency SD→P causes a violation of BCNF.

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

A dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. To define dependency-preserving decompositions precisely, we have to introduce the concept of a projection of Fds.

Let R be a relation schema that is decomposed into two schemas with attribute sets X and Y, and let F be a set of FDs over R. the projection of F on X is the set of Fds in the closure F^+ that involve only attributes in X. we denote the projection of F on attributes X as Fx.

The decomposition of relation schema R with FDs F into schemas with attribute sets X and Y is dependency-preserving if (Fx U Fy)+= F^+. that is, if we take the dependencies in Fx and Fy and compute the closure of their union, we get back all dependencies in the closure of F.

6.6 Normalization

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

Decomposition into BCNF

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

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

R –A denotes the set of attributes other than A in R, and XA denotes the union of attributes in X and A. Since X→A violates BCNF, it is not a trivial dependency; further, a is a single attribute. Therefore, A is not in x; that is, X n A is empty. Therefore, each decomposition carried out in step1 is lossless-join.

BCNF and Dependency-preservation

Sometimes, there simply is no decomposition into BCNF that is dependency preserving. Consider the relation schema SBD, in which a tuple denotes that sailors S has reserved boat B on date D. If we have the FDs SB→D and D→B, SBD is not in BCNF because D is not a key. If we try to decompose it, however, we cannot preserve the dependency SB→D.

Decomposition into 3NF

Clearly, the approach we outlined lossless-join decomposition into BCNF also gives us a lossless-join decomposition into 3NF.

Minimal cover for a set of FDs

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

1. Every dependency in G is of the form X→a, whole A is a single attribute.
2. The closure F^+ is equal to the closure G^+
3. if we obtain a set H of dependencies from G by deleting one or more dependencies or by deleting attributes from a dependency in G, then $F^+ \neq H^+$.

A minimal cover fro a set F of FDs is an equivalent set of dependencies that is minimal in two respects:

1. Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute.
2. Every dependency in it is required for the closure to be equal to F^+.

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

A→B, ABCD→E, EF→G, EF→H, and ACDF→EG.

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

ACDF→E and ACDF→G.

Next consider ACDF→G. this dependency is implied by the following FDs:

A→B, ABCD→E, and EF→G.

Therefore, we can delete it. Similarly, we can delete ACDF→E.

A minimal cover for F is the set:

A→B, ACD→E, EF→G, and EF→H.

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

1. Put the FDs in a Standard Form: Obtain a collection G of equivalent FDs with a single attribute on the right side.
2. Minimize the left side of each FD: For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalent to F^+.
3. Delete Redundant FDs: Check each remaining FD in G to see if it can be deleted while preserving equivalence to F^+.

Dependency-Preserving Decomposition into 3NF

Dependency preserving decomposition into 3NF relations, let R be a relation with a set of FDs that is a minimal cover, and ler R1, R2, …, Rn be a lossless-join decomposition of R.

Self-Assesment Questions – VI

1. F Closure can be represented by_____.
2. Normal forms are used to _____.
3. Expansion of BCNF is _____.
4. Expansion of FD is _____.
5. Decomposition is used to
    a) Split the table
    b) Join the tables
    c) Combine the tables
6. Which is not a valid normal form
    a) 1 NF
    b) 8 NF
    c) 5 NF
7. **Sample questions**
8. Write notes about Functional dependency.
9. Write down the concept of Decomposition.
10. Explain different types Normal forms?
11. Explain Multivalued Dependencies?

Answers for Self-Assesment Questions – VI

1. F^+
2. Reduce redundant values
3. Boyce-Codd Normal Form
4. Functional Dependency
5. a – split the table
6. b – 8 NF

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

7. Security and Authorization

The data stored in a DBMS is often vital to the business interests of the organization and is regard as a corporate asset. In addition to protecting the intrinsic value of the data, corporations must not be revealed to certain groups of users for various reasons. In this chapter we discuss about various security and authorization mechanisms.

7.1 introduction to Database Security

There are three main objectives when designing a secure database application:

Security: Information should not be disclosed to unauthorized users. For example, a student should not be allowed to examine other student's grades.

Integrity: Only authorized users should be allowed to modify data. For example, students may be allowed to see their grades, yet not allowed to modify them.

Availability: Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.

To achieve these objectives, a clear and consistent security policy should be developed to describe what security measures must be enforced. In particular, we must determine what part of the data is to protected and which users get access to which portions of the ata. Next, security of the underlying DBMS an operating system, as well as external mechanism, such as securing access to buildings, must be utilized to enforce the policy, we emphasize that security measures must be taken at different levels.

We use the following schemas in our exapmpl.

Sailors(sid: integer, sname:string, rating: integer, age:real)

Boats(bid:integer, bname:string, color:string)

Reserves(sid:integer, bid:integer, day:date)

7.2 Access Control

A database for an enterprise contains a great deal of information and usually has several groups of users. Most users need to access only a small part of the database to carryout their tasks. Allowing users unrestricted access to all the data can be undesirable, and a DBMS should provide mechanisms to control access to data.

A DBMS offers two approaches to access control. Discretionary access control is based on the concept of access rights, or privileges, and mechanisms for giving users such privileges. A privilege allows a user to access some data object in a certain. A user who creates a database object such as a table or view automatically gets all applicable privileges on that object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and measures that at all times only users with the necessary privileges can access an object. SQL supports discretionary access control through the GRANT and REVOKE commands. The GRANT command gives privileges to the users, and the REVOKE command takes away privileges.

Mandatory access control is based on system wide policies that cannot be changed by individual users.

7.3 Discretionary Access Control

SQL supports discretionary access control through the GRANT and REVOKE commands. The GRANT command gives users privileges to base and views. The syntax of this command is as follows:

GRANT privileges ON users [WITH GRANT OPTION]

For our purposes object is either a base table or a view. SQL recognizes certain other kinds of objects, but we do not discuss them. Several privileges can be specified, including these:

SELECT: The right to accesss all columns of the table specified as the object, including columns added later through ALTER TABLE command.

INSERT: The right to insert rows with values in the named column of the table named as object. If this right is to granted with respect to all columns, including columns that might be added later, we can simply INSERT. The privilege UPDATE(column name) and UPDATE are similar.

DELETE: The right to delete rows from the table named as object.

REFERENCES: the right to define foreign keys that refer to the specified column of the table object. REFERENCES without a column name specified denotes this right with respect to all columns, including any that are added later.

Examples:

UPDATE sailors S

SET S.rating = 8

UPDATE sailors S

SET S.rating = S.rating-1

The first query update rating value for all the tuples in sailors as 8, and the second query will reduce the rating value by 1.

GRANT INSERT ON sailors TO Michel

This will grant the insert privilege to michel on the schema sailors.

General syntax:

REVOKE [GRANT OPTION FOR] privileges

ON object FROM users { RESTRICT | CASCADE }

7.4 Mandatory Access Control

To apply mandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of tables, rows, or even individual column values. Let us assume that each row is assigned a security class. This situation leads to the concept of a multilevel table.

Consider the instance of the Boats table in fig a. users with S and TS clearance get both rows in the answer when they ask to see all rows in Boats.

A user with C clearance gets only the second row, and user with U clearance gets no rows

Bid	bname	color	Security class
101	Salsa	Red	S
102	Pinto	Brown	C

Fig a. An instance B1 of Boats

The Boats is defined to have bid as the primary key. Suppose that a user with clearance C wishes to enter the row <101, Picante, Scarlet, C>. we have a dilemma:

- If the insertion is permitted, two distinct rows in the table have key 101.
- If the insertion is not permitted because the primary key constraint is violated, the user trying to insert the new row, who has clearance C, can infer that there is a boat with bid=101 whose security class is higher than C. this situation compromises the principle that users should not be able to infer any information about objects that have a higher security classification.

This delimma is resolved by effectively treating the security classification as part of the key. Thus, the insertion is allowed to continue, and the table instance is modified as shown in fig b.

Bid	bname	color	Security class
101	Salsa	Red	S
101	Picante	Scarlet	C
102	Pinto	Brown	C

Fig b. An instance B1 of Boats after insertion

7.5 Security for Internet Applications

When a DBMS is accessed from a secure location, we can rely upon a simple password mechanism for authenticating users. However, suppose our friends wants to place an order for a book over the internet. This presents some unique challenges: Encryption techniques provide the foundation for modern authentication.

Encryption

The basic idea behind encryption is to apply an encryption algorithm to the data, using a user-specified or DBA-specified encryption key. The output of the algorithm is the encrypted version of the data. There is also a decryption algorithm, which takes the encrypted data and decryption key as input and then returns the original data. Without the correct decryption key, the decryption algorithms produces gibberish. The encryption and decryption algorithms themselves are assumed to be publicly known, but one or both keys are secret.

In symmetric encryption, the encryption key is also used as the decryption key. The ANSI Data Encryption Standard (DES), which has been in use since 1977, is a well-known example of symmetric encryption.

Another approach to encryption, called public-key encryption, has become increasingly popular in recent days. Each authorized user has a public encryption key, known to every one, and a private decryption key, known only to him or her. Since the private decryption keys are known only to their owners, the weakness of DES is avoided.

The encryption function is S = I mod L.

The decryption function is I = S mod L.

Certifying Services : The SSL protocol

A number of companies serve as certification authorities, eg. Verisign. Amazon generates a public encryption key e and sends the public key to verisign. Verisign then issues a certificate to amazon that contain the following information:

< verisign, Amazon, http://www.amazon.com,e>

The certificate is encrypted using Verisign's own private key, which is known to Internet Explorer, Netscape Navigator and other browsers.

Digital Signatures

Public key cryptography can be used to create digital signatures for messages. That is, messages can be encoded in such a way that, if Elmer gets a message supposedly from Betsy, he can verify that it is from Betsy and further, prove that it is from Betsy, even if the message is sent form a hotmail account when Betsy is traveling.

A clever use of the encryption scheme, however, allows Elmer to verify whether the message was indeed sent by Betsy. Betsy encrypts the message using her private key and then encrypts the result using Elmer's public key.

If authenticating the sender is the objective and hiding the message is not important, we can reduce the cost of encryption by using message signature. A signature is obtained by applying a one-way function to the message and is considerably smaller. We encode the signature as in the basic digital signature approach, and send the encoded signature together with the full, un encoded message. The recipient can verify the sender of the signatures just described, and validate the message itself by applying the one-way function and comparing the result with the signature.

7.6. Network Model

A network database consists of a collection of records which are connected to one another through links. A record is in many respects similar to an array in the entity-relationship model. Each record is a collection of fields (attributes), each if which contains only one data value. A link is an association between precisely two records. Thus, a link can be viewed as a restricted form of relationship in the sense of the E-R model.

Data Structured Diagrams

A data structured diagram is a scheme representing the design of a network database. Such a diagram consists of two basic components namely boxes, which corresponds to record type and lines which corresponds to links.

A data structure diagram serves the same purpose as an entity relationship diagram, namely, it specifies the overall logical structure of the database.

Binary Relationship

Consider the entity relationship diagram

7.7 Hierarchical Model

A hierarchical database consists of a collection of records which are connected to one another through links. Each record is a collection of fields(attributes), each of which contains only one data value. A link is an association between precisely two records. Thus, a link is similar to the link concept in the network model.

Consider a database representing a customer account relationship in a banking system. There are two record types namely customer and account. The customer record consists of three fields called name, street, and city. Similarly the account record consists of two fields called number and balance.

a hierarchical database is a collection of such rooted trees, and hence forms a forest. We refer to each such rooted tree as a database tree.

Tree Structured Diagram

A tree structure diagram is the scheme for a hierarchical database. Such a diagram consists of two components, namely boxes and lines. Boxes, which correspond to record types and lines which corresponds to links. A tree structure diagram serves the same purpose as an entity relationship diagram; namely, it specifies the overall logical structure of the database.

A tree structure diagram is similar to a data structure diagram in the network model. The main difference is that, in the later, record types are organized in the form of an arbitrary graph, while in the former; record types are organized in the form of a rooted tree. The record tree is defined as there can be no cycles in the underlying graph. The relationships formed in the graph must be such that only one to one and one to many relationships exist between a parent and a child.

The database scheme is represented as a collection of tree structure diagrams. For each such diagram. There exists one single instance of a database tree. The root of this tree is a dummy node. The children of that node are instances of the appropriate record type. Each such child instance may, in turn, have several instance may, in turn have several instances, of various record type, as specified in the corresponding tree structure diagram.

Single Relationship

Between two entities, there is only one relationship is called as single relationship, which is also called as one-to-one relationship.

Several Relationships

Between two entities, there is more than one relationship is called as several relationship, which is also called as one-to-many and many-to-one relationship.

Self-Assesment Questions – VII

1. Encryption will _____.
2. Revoke will _____.
3. Decryption will _____.
4. Digital signature is _____.
5. Grant is used to
    a) Give permission to access the data
    b) Cancel permission to access the data
    c) Cancel permission to delete table
6. DES stands for
    a) Domain Encryption Standard
    b) Data Encryption Standard
    c) Data Enterprise Standard

Sample questions

7. Write notes about database security.
8. Write down the concept of encryption and decryption.
9. Explain different access control techniques?
10. Explain Digital signatures?
11. Explain network and Hierarchical Data models?

Answers for Self-Assesment Questions – VII

1. convert plain text into cipher text
2. withdraw the access permission
3. convert cipher text into plaintext
4. add the signature with the document
5. a – Give permission to access the data
6. b – Data Encryption Standard

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

8. Parallel and Distribution Database

8.1. Introduction

we have thus far considered centralized database management systems, in which all the data is maintained at a single side, and assumed that the processing of individual transactions are essentially sequential. One of the most important trends in database is the increased use of parallel evaluation techniques and data distribution. There are four distinct motivations.

Performance: Using several resources (Eg. CPUS and disks) in parallel can significantly improved performance.

Increased availability: If a site containing a relation goes down, the relation continues to be available it a copy is maintained at another site.

Distributed access to data: An organization may have branches in several cites. Although analysts may need to access data corresponding to different sites, we usually find locality in the access patterns, and this locality can be exploited by distribution the data accordingly.

Analysis of distributed data: Organizations increasingly want to examine all the data available to them, even when it is stored across multiple sites and on multiple databases systems. Support for such integrated access involves many issues; even enabling access to widely distributed data can be a challenge.

A parallel Database System is one that seeks to improve performance through parallel implementation of various operations such as loading data, building indexes, and evaluating queries. Although data may be stored in a distributed fashion in such a system, the distributed is governed solely by performance consideration.

In a Distributed Database System, data is physically stored across several sites, and each site is typically managed by a DBMS that is capable of running independently of the other sites. The location of data items and the degree of autonomy of individual sites have a significant impact on all aspects of the system, including query optimization and processing, concurrency control, and recovery. In contrast to parallel database the distributed of data is governed by factors such as local ownership and increased availability in addition to performance issues.

8.2 Architecture for Parallel Databases

The basic idea behind parallel database is to carry out evaluation steps in parallel whenever possible in order to improve performance. There are many opportunities for parallelism in a DBMS; database represent one of the most successful instance of parallel computing.

The shared- nothing architecture requires more extensive reorganization of the DBMS code, but it has been shown to provide linear speed-up, in that the time taken for operations decreases in proportion, to the increase in the number of CPUs and disks, and linear scale-up, in that performance is sustained if the CPUs and disks are increased in proportion to the amount of data consequently, even more powerful parallel database systems can be built by taking advantage

of rapidly powerful improving performance for singly CPU systems and connecting as many CPU as desired.

Speed-up and Scale-up is illustrated in fig. 8.1. the speed-up curves show how for a fixed database size, name transactions can be executed per second by adding CPUs. The scale-up curves show how adding more resources enables us to process large problems. The first Scale-up graph measures the number of transactions executed per second as the database. Size is increased and the number of CPUs is correspondingly increased. An alternative way to measure Scale-up is to consider the time taken per transaction as more CPUs are added to process an increasing number of transactions per second; the good here is to sustain the response time per transaction.

8.3 Parallel Query Evaluation

A relational query execution plan is a graph of relational algebra operations and the operators in a graph can be executed in parallel. Of an operator consumes the output of a second operator, we have pipelined parallelism (the output of the second operator is worked on by the first operator as soon as it is generated); if not, the two operators can proceed essentially independently. An operator is said to block if it produces no output until it has consumed all its inputs pipelined parallelism is limited by the presence of operators (E.g. Sorting or aggregation) that block.

In addition to evaluating different operators in parallel, we can evaluate each individual operator in a query plan in parallel fashion. The key to evaluating an operator in parallel is to partition the input data; we can then work on each partition is parallel and combain the results. This approach is called data-partitioned parallel evaluation. By expressing some care, existing code for sequentially evaluating relational operators can be ported easier for data-partitioned parallel evaluation.

An important observation which explains why shared-nothing parallel database systems have been very successful, is that database query evaluation is very amenable to data-partitioned parallel evaluation. The goal is to minimize data shipping by partitioning the data and by structuring the algorithms to do most of the processing at individual process.

Data Partitioning

Partitioning a large dataset horizontally across several disks enable s us to exploit the I/O bandwidth of the disks by reading and writing them in parallel. There are several ways to horizontally partition a relation. We can assign tuples to processors in a round-robin fashion, we can use hashing, or we can assign tuples to processors by ranges of field values. If there are n processors, the ith tuple is assigned to processor I mod n in round-robin partitioning. In hash partitioning, a hash function is applied to tuple to determine its processor. In range partitioning tuples are stored, and n ranges are chosen for the sort key values so that each range contains roughly the same number of tuples; tuples in range I are assigned to processor i.

Round-robin partitioning is suitable for efficiently evaluating queries that access the entire relation. If only a subset of the tuples (Eg. Those that satisfy the selection condition age = 20) is required, hash partitioning and range partitioning are better than round-robin partitioning because they enable us to access only those disks that contain matching tuples. If range selection such as 15<age<25 are specified, range partitioning is superior to hash partitioning because qualifying tuples are likely to be clustered together on a few processors. On the other hand, range partitioning can lead to data skew; that is, partitions with widely varying number of tuples across partitions or disks. Skew causes processors dealing with large partitions be to become performance bottlenecks. Hash partitioning has the additional virtue that it keeps data evenly distributed even if the data grows and shrinks over time.

To reduce skew in range partitioning, the main question is how to choose the range by which tuples are distributed. One effective approach is to take samples from each processor, collect and sort all samples, and divide the sorted set of samples into equally sized subsets. If tuples are to be partitioned on age, the age ranges of the sampled subsets of tuples can be used as the basis for redistributing the entire relation.

8.4 Parallelizing Individual Operations

1. **Bulk Loading and Scanning**

   We design with two simple operations; Scanning a relation and loading a relation. Pages can be read in parallel while scanning a relation, and the retrieved tuples can then be merged, if the relation is partitioned access several disks. More generally, the idea also applies when retrieving all tuples that meet a selection condition, if hashing or range partitioning is used, selection queries can be answered by going to just those processors that contain relevant tuples.

   A similar observation holds for bulk loading. Further, if a relation has associated indexes, any sorting of data entries required for building the indexes during bulk loading can also be done in parallel.

2. **Sorting**

   A simple idea is to let each CPU sort the part of the relation that is on its local disk and to then merge these sorted sets of tuples. The degree of parallelism is likely to be limited by the merging phase.

   A better idea is to first redistribute all tuples in the relation using range partitioning. For example if we want to sort a collection of employee tuples by salary, salary values range from 10 to 210 and we have 20 processors, we could send all tuples with salary values in the range 10 to 20 to the first processor, all in the range 21 to 30 to the second processor, and so on.

   Each processor then sorts the tuples assigned to it, using some sequential sorting algorithm. For example, a processor can collect tuples until its memory is full, then sort these tuples and write out a run, until all incoming have been written to such sorted runs on the local disk.

These runs can then be merged to create the sorted version of the set of tuples assigned to this processor. The entire sorted-relation can be retrieved by visiting the processors in an order corresponding to the range assigned to them and simple scanning the tuple.

The basic challenge in parallel sorting is to do the range partitioning so that each processor receives roughly the same number of tuples,; otherwise, a processot that receives a disproportionately large number of tuples to sort becomes a bottleneck and limits the scalability of the parallel sort. One good approach to range partitioning is to obtain a sample of the entire relation by taking samples at each processor that initially contains part of the relation. The sample is sorted and used to identify determined, we will not discuss how good subrange boundaries can be identified.

Having decided on a partitioning strategy, we can assign each partition to a processor and carry out a local join, using any join algorithm, we want , at each processor. In this case a number of partitions K is chosen to be equal to the number of processors n that are available for carrying out the join, and during partitioning, each processor sends tuples in the ith partition to processor i. after partitioning each processor joins the A and B tuples assigned to it. Each join process executes sequential join code, a merge operator merges all incoming A tuples, and another merge operator merges all incoming B tuples. Depending on how we want to distribute the result of the join of A and B, the output of the join process may be split into several data streams. The network of operators for parallel join is shown in fig. 8.2. to simplify the fig. we assume that the processors doing the join are distinct for the processors that initially contain tuples of A and B show only four processors.

If range partitioning is used, the algorithm outlined above leads to a parallel version of a sort merge join, with the advantage that the output is available in sorted order. If hash partitioning is used, we obtain a parallel version of a hash join.

8.5 Parallel Query Optimization

In addition to parallelizing individual operations, we can obviously execute different operations in a query in parallel and execute multiple queries in parallel. Optimizing a single query for parallel execution has received more attention; systems typically optimize queries without regard to other queries that might be executing at the same time.

Two kinds of inter operation parallelism can be exploited with in a query:

- The result of one operator can be pipelined into another. For example consider a left-deep plan in which all the joins use index nested loops. The result of the first joins is the other relation tuples for the next join node. As tuples are produced by the first join, they

can be used to prove the inner relation in the second join. The result of the second join can similarly be pipelined into the next join, and so on.

- Multiple independent operations can be executed concurrently. For example, consider a (non left –deep) plan in which relations A and D are joined, and the results of these two joins are finally joined. Clearly, the join of A and B can be executed concurrently with the join of C and D.

An optimizer that seeks to parallelize query evaluation has to consider several issues, and we will only outline the main points. The cost of executing individual operations in parallel (Eg. Parallel sorting) obviously differs from executing them sequentially, and the optimizer should estimate operation costs accordingly.

8.6 Introduction to Distributed Databases

The classical view of a distributed database system is that the system should make the impact of data distribution transparent. In particular the following properties considered desirable:

Distributed data independence: Users should be able to ask queries without specifying where the referenced relations or copies of the relations, are located. This principle is a natural extension of physical and logical data independence.

Distributed transaction atomicity: Users should be able to write transactions that access and update data at several sites just as they would write transactions over purely local data. In particular the effects of a transaction across sites should continue to be atomic; that is all changes persist if the transaction commits, and non persist if it aborts.

Types of Distributed Database

If data is distributed but all servers run the some DBMS software, we have a homogeneous distributed database system. If different sites run under the control of different DBMS, essentially autonomously, and are connected somehow to enable access to data from multiple sites, we have a heterogeneous distributed database system, also referred to as a multi database system.

The key to building heterogeneous systems is to have well-accepted standards for gateway protocols. A gateway protocol is an API that expose DBMS functionality to external applications. Examples include ODBC and JDBC. By accessing database servers through gateway protocols, their differences are masked, and the differences between the different servers in a distributed system are bridged to a large degree.

Gateways re not a panacea however, they add a layer of processing that can be expensive, and they do not completely mask the difference between servers. For example, a server may not be capable of providing the services required for distributed transaction management, and even if it is capable, standardizing gateway protocols at the way down to this level of interaction poses challenges that have not yet been resolved satisfactorily.

Distributed data management in the final analysis, comes at a significant cost in terms of performance, software complexity and administration difficulty. This observation is especially true of heterogeneous systems.

8.7 Distributed DBMS Architecture

There are three alternative approaches to separating functionality across different DBMS related processes; these alternative distributed DBMS architectures are called client-server, collaborating server and middleware.

1. **client-server system**

   A client-server system has on or more client processes and one or more several process. Clients are responsible for user-interface issues, and servers manage data and execute transactions. Thus a client process could run on a personal computer and send queries to a server running on a main frame.

   This architecture has become very popular far several reasons. First, it is relatively very simple to implement due to its clear separation of functionality and because the server is centralized. Second expensive server machines are not underutilized by dealing with mundane iser-interactions, which are now related to inexpensive client machines. Third, users can run a graphical user interface that they are familiar with rather than the user interface on the server.

   While writing client-server application, it is important to remember the boundary between the client and the server and to keep the communication between them as set-oriented as possible. In particular, opening a cursor and fetching tuples on at a time generates many messages and should be avoided. (Even if we fetch several tuples and cache them at the client, messages must be exchanged when the cursor is advanced to ensure that the current row is locked).

2. **Collaborating Server system**

   The client-server architecture does not allow a single query to span multiple servers because the client process would have to be capable of breaking such a query into appropriate sub queries to be executed at different sites and then piecing together the answers to the sub queries. The client process would thus be quite complex, and its capabilities would begin to overlap with the server; distinguishing between clients and servers becomes harder. Eliminating this distinction leads us to an alternative server system we can have a collection of database servers, each capable of running transaction against local data, which cooperatively execute transactions spanning multiple servers.

   When server receives a query that requires access to data at other servers, it generates appropriate sub queries to be executed by other servers and puts the results together to compute answers to the original query. Ideally, the decomposition of the query should be done

using cost-based optimization, taking into account the costs of network communication as well as local processing costs.

3. **Middleware Systems**

The middleware architecture is designed to allow a single query to span multiple servers, without requiring all database to be capable of managing such multi site execution strategies. It is especially alternative when trying to integrate several legacy systems, whose basic capabilities cannot be extended.

The idea is that we need just one database server that is capable of managing queries and transactions spanning multiple servers; the remaining servers only need to handle local queries and transactions. We can think of this special server as a layer of software that coordinates the execution of queries and transactions across one or more independent database servers; such software is often called middleware. The middleware layer is capable of executing joins are other relational operations on data obtained from the other servers, but typically, does not itself maintain any data.

8.8. Storing Data in a Distributed DBMS

In a distributed DBMS, relation are stores across several sites. Accessing a relational that is stored at a remote site incurs message passing costs, and to reduce this overhead, a single relation may be partitioned or fragmented across several sites, with fragments stored at the sites where they are most often accessed, or replicated at each site where the relation is in high demand.

1. **Fragmentation**

Fragmentation consists of breaking a relation into smaller relations or fragments, and storing the fragments, possibly at different sites. In horizontal fragmentation, each fragment consists of a subset of rows of the original relation. In vertical fragmentation, each fragment consists of s subset of columns of the original relation. Horizontal and vertical fragmentation are illustrated in fig. 8.4.

Tid	eid	name	city	age	Sal
T1	53666	Jomes	Madraas	12	35
T2	53688	Smith	Chicago	18	32
T3	53650	Smith	Chicago	19	48
T4	53831	Madayan	Mumbay	11	20
T5	53832	guldu	Mumbay	12	20

Fig. 8.4. Horizontal and Vertical Fragmentations

Typically the tuples that belong to a given horizontal fragment are identified by a selection query; for example, employee tuples might be organized into fragments by city, with all employees in a given city

assigned to the same fragments. The horizontal fragment show in fig. 8.4 corresponds to Chicago. By storing fragments in the database site at the corresponding city, we achieve locality of reference from Chicago, and storing this data on Chicago makes it local (and reduces communication costs) for most queries. Similarly, the tuples in a given vertical fragment are identified by a projection query. The vertical fragment in the figure results from projection on the first two columns of the employee relation.

When a relation is fragments, we must be able to recover the original relation from the fragments.

- **Horizontal fragmentation:** The union on the horizontal fragments must be equal to the original relation. Fragments are usually also required up to be disjoint.
- **Vertical fragmentation:** The collection of vertical fragments should be a lossless join decomposition.

2. Replication

Replication means that we store several copies of a relation or relation fragment. An entire relation can be replicated at one or more sites. Similarly, one or more fragments of a relation can be replicated at other sites.

Increased availability of Data: If a site that contains a replication goes down, we can find the same data at other sites. Similarly, if local copies of remote relations are available, we are less vulnerable to failure of communication links.

Faster query evaluation: Queries can execute faster by using a local copy of a relation instead of going to a remote site.

8.9 Distributed Catalog Management

Keeping track of data distributed across several sites can get accomplished. We must keep track of how relations are fragmented and replicated- that is how relation fragments are distributed across several sites and where copies of fragments are stored.

Naming Objects

If a relation is fragmented and replicated, we must be able to uniquely identify each replica of each fragment. Generating such unique names requires some autonomy is compromised.

The usual solution to the naming problem is to use names consisting of several fields. For example, we would have:

A local name field, which is the name assigned locally at the site where the relation is created. Two objects at different sites could have the same local name, but two objects at a given site cannot have the same local name.

A birth site field, which identifies the site where the relation was created, and where information is maintained about all fragments and replicas of the relation .

Those two fields identify a relation uniquely; we call the combination a global relation name.

Catalog structure

A centralized system catalog can be used but is vulnerable to failure of the site containing the catalog. An alternative is to maintain a copy of a global system catalog, which describes all the data at every sites.

A better approach, which preserves local autonomy and is not vulnerable to a single site failure, was developed in the R* distributed database project, which was a successor to the system R project at IBM site maintains a local catalog that describes all copies of data stored at that site.

To locate relation, the catalog at its birth site must be looked up. This catalog information can be cached at other sites for quicker access, but the cached information may become out of date if, for example, a fragment is moved. we would discover that the locally cached information is out of date when we use it to access the relation, and at that point , we must update the cache by looking at the birth site of a relation.

Distributed Data Independence

Distributed data independence means that users should be able to write queries without regard to how a relation is fragmented or replicated; it is the responsibility of the DBMS to compute the relation as needed.

In particular, this property implies that users should not have to specify the full name for the data objects accessed while evaluating a query. The local name of a relation in the system catalog is really a combination of a user name and a user-defined relation name. users can give whatever name they wish to give their relations, without regard to the relations created by other users.

A user may want to create objects at several sites or to relations created by other users. To do this, a user can create a synonym for a global relation name, using an SQL-style command and subsequently refer to the relation using synonym.

8.10 Distributed Query Processing

1. Non Joins queries in a Distributed DBMS

Even simple operations such as scanning a relation, selection and projection are affected by fragmentation and replication consider the following query:

```
SELECT S.age
FROM Sailors S
WHERE S.rating > 3 AND S.rating<7
```

Suppose that the sailors relation horizontally fragmented, with all tuples having a rating less than 5 at Shanghai and all tuples having a rating greater than 5 at Tokyo.

The DBMS must answer this query by evaluating it at both sites and taking the union of the answers. If the SELECET class contained

AVG(S.age). combining the answers cannot be done by simple taking the union. The DBMS must compute the sum and count of age values at the two sites and use this information to compute the average of all sailors.

If the WHERE clause contained just the condition S.rating>6, on the other hand, the DBMS should recognize that this query can be answered by just executing it and Tokyo.

2. **Joins in a Distributed DBMS**

Joins are relations at different sites can be very expensive, and we now consider the evaluation options that must be considered in a distributed environment. Suppose that the sailors relation is stored at London, and that the reserves relation is stored at paris. We will consider the cost of various strategies for computing sailors reserves.

Fetches as needed

We would be a page oriented nested loops join in Landon with Sailors as the outer, and the each Sailors page, fetch all Reserves pages from Paris. If we cache the fetched reserves pages in London until the join is complete, pages are fetched only once but lets assume that reserves pages are not cached, just to see how bad things can get. (this situation can get much worse it we use a tuple-oriented nested loops join).

The cost is 500 t_d to scan sailors plus, for each sailors page, the cost of scanning and shipping all of reserves, which is 1,000 (t_d +t_s).The total cost is therefore 500 t_d + 500000(t_d +t_s).

Semi joins and bloom joins

Two techniques semi join and bloom join have been proposed reducing the number of reserves tuples to be shipped. The first technique is called semijoin, the idea is to proceed in three steps:

1. At London, compute the projection of sailors on to the join columns (in this case just the sid field), and ship this projection to Paris.

2. At Paris, compute the natural join of the projection received from the first site with the reserves relation. The result of this join is called the reduction of reserves with respect to sailors. Clearly only those reserves tuples in the reduction will join with tuples in the sailors relation. Therefore, ship the reduction of reserves to London, rather than the entire reserves relation.

3. At London, compute the joi of the relation of reserves with sailors.

Let us compute the cost using this example technique for our join query. If we assume that the size of the sid field is 10 bytes, the cost of projection is 500 for scanning sailors, plus 100 for creating the temporary, plus 400 for sorting it (in two passes). Plus 100 for the final scan, plus 100 for writing the result of 12000 . (Because sid is a key, there are no duplications to be eliminated, if optimizer is good enough to recognize this, the cost of projection is just (500+100) .

The second technique, called Bloom join, is quite similar/ the main diference is that a bit-vector is shipped in the final stop, instead of the

projection of sailors, A bit-vector of (some chosen) size K is computed by hashing each tuple of sailors into the range 0 to K-1 and setting bit I to 1, if some tuple hashes to I, and 0 otherwise. In the second step, the reduction of reserves is computed by hashing each tuple of reserves (using the sid field) into the range 0 to K-1, using the same hash function used to construct the bit-vector and discarding tuples whose hash value I corresponds to a 0 bit. Because no Sailors tuples hash to such an I \, no Sailors tuple can join with any reserves tuple that is not in the relation.

Cost-based Query Optimization

We have seen how data distribution can affect the implementation of individual operations such as selection, projection, aggregation and join. In general , ofcourse a query involves several operations, and optimizing queries in a distributed database poses the following additional challenges.

- Communication costs must be considered. If we have several copies of a relation, we must also decide which copy to use.
- If individual sites are run under the control of different DBMS, the autonomy of each site must be represented while doing global query planning.

8.11 Updating Distributed Data

1. Synchronous Replication

There are two basic techniques for ensuring that transactions see the same value regardless of which copy of an object they access.

In the first technique called Voting, a transaction must write a majority of copies in order to modify an object and read at least enough copies to make sure that one of the copies is current. For example, if there are 10 copies and 7 copies are written by update transactions, then at least 4 copies must be read. Each copy has version number and the copy with the highest version number is current. This technique is not attractive in most situations because reading an object requires reading multiple copies in most applications, objects are read must more frequently that they are updated and efficient performance on reads is very important.

In the second technique, called read-any write-all, to read an object, a transaction can read any one copy, but to write an object, it must write all copies. Reads are fast, especially if we a local copy, but writes are slower, relative to the first technique. This technique is attractive when reads are must more frequent than writes, and it usually adopted for implementing synchronous replications.

2. Asynchronous Replication

Synchronous replication comes at a significant cost. Before an update transaction can commit, it must obtain exclusive locks on all copies assuming that the read-any write –all technique is used of modified data. The transaction may have to send lock requests to remote sites, and wait for the locks to be granted, and during this potentially long period, it continues to hold all its other

locks. If sites or communication links fail, the transaction cannot commit until all sites at which it has modified data recover and are reachable. Finally, even if licks are transaction requires several additional to be sent as part of a commit protocol.

For these reasons, synchronous replication is undesirable or even unachievable in many situations. Asynchronous replication is gaining in popularity, even through it allows different copies of the same object to have different values for short periods of time. This independence, users must be aware of which copy they are accessing, recognize that copies are brought up-to-date only periodically, and live with this reduced level of data consistency. Nonetheless, this seems to be a practical compromise that is acceptable in many situations.

Primary site Versus Peer-to-Peer Replication

Asynchronous replication comes in two flavors. In primary site asynchronous replication, one copy of a relation is designated as the primary or master copy. Replicates of the entire relation or of fragments of the replication can be created at other sites; these are secondary copies, and unlike the primary copy, they cannot be updates. A common mechanism for setting up primary and secondary copies is that users first register or publish the relation at the primary site and subsequently subscribe to a fragment of a registers relation from another site.

In Peer-to-Peer asynchronous replication, more than one copy (although perhaps not all) can be designated as being updatable, that is a master copy. In addition to propagating changes, a conflict resolution strategy must be used to deal with conflicting changes made at different sites.

The main issue in implementing primary site replication is determining how changes to the primary copy are propagated to the secondary copies. Changes are usually propagated in two steps called Capture and Apply. Changes made up committed transactions to the primary copy are some how identified during the capture step and subsequently propagated to secondary copies during the apply step.

8.12 Distributed Transactions

In a distributed DBMS, a given transaction is submitted at some one site, but it can access data at other sites as well. Here we discuss about sub transaction. When a transaction is submitted at some site, the transaction manager at that site breaks it up into a collection of one or more sub transactions that execute at different sites, submits them to transaction managers at the other sites.

8.13 Distributed concurrency control

Lock management can be distributed across sites in many ways:

Centralized: A single site is in-charge of handling lock and unlock requests for all objects.

Primary Copy: One copy of each object is designed the primary copy. All requests to lock or unlock a copy of this object are handled by the lock manager at the site where the primary copy is stored, regardless of where the copy itself is stored.

Fully Distributed: Requests to lock or unlock a copy of an object stored at a site are handled by the lock manager at the site where the copy is stored.

8.14 Distributed Recovery

Recovery in a distributed DBMS is more complicated that in a centralized DBMS for the following reasons:

- New kinds of failure can arise, namely, failure of communication links and failure of a remote site at which a sub transaction is executing.
- Either all sub transactions of a given transaction must commit or none must commit, and this property must be guaranteed despite any combination of site and link failures. This guarantee is achieved using a commit protocol.

As in the centralized DBMS, certain actions are carried out as part of normal execution in order to provide the necessary information to recover from failures. A log is maintained at each site, and in addition to the kinds of information maintained in a centralized DBMS, actions taken as part of the commit protocol are also logged. The most widely used commit protocol is called Two-phase commit (2PC). A variant called 2PC with Presumed Abort, which we discuss below, has been adopted as an industry standard.

(1) Normal execution

Each site maintains a log, and the actions of a sub transaction are logged at the site where it executes. The transaction manager at the site where the transaction originated is called coordinator for the transaction; transaction managers at sites where its sub transaction executes are called subordinates (with respect to the coordination of this transaction).

We now describe the two-phase commit (2PC) protocol, interms of the message exchanged and the log records written. When the user decides to commit a transaction, the commit command is sent to the coordinator for the transaction. This initiates the 2PC protocol.

1. The coordinator sends a prepare message to each subordinate.
2. When a subordinate receives a prepare message, it decides whether to abort or commit its sub transaction. It force-writes an abort a prepare log record, and then sends a no or yes message to the coordinator. Notice that a prepare log record is not used in a centralized DBMS; it is unique to the distributed commit protocol.
3. If the coordinator receives yes message from all subordinate, it force-writes a commit log record and then sends a commit message to all subordinates. If it receives even one no message, or does not receive any response from some subordinate for a specified time-out interval, if

force-writes an abort log record, and then sends an abort message to all subordinates.

4. When a subordinate receives an abort message, it force-writes an abort log record, sends an ack message to the coordinator, and aborts the sub transaction. When a subordinate receives a commit message, if force-writes a commit log record sends an ack message to the coordinator, and commits the sub transaction.
5. After the coordinator has received ack messages from all subordinates, it writes an end log record for the transaction.

The names two-phase commit reflects the fact that two rounds of messages are exchanged; first a voting phase, then a terminating phase, both initiated by the coordinator. The basic principle is that any of the transaction, where as there must be unanimity to commit a transaction. When a message is sent in 2PC, it signals a decision by the sender. In order to ensure that this decision survives a crash at the sender's site, the log record describing the decision is always forced to stable storage before the message is sent.

A transaction is officially committed at the time the coordinators commit log record reaches stable storage. Subsequently failures cannot affect the failures of the outcome; it is irrevocably committed. Log records written to record the commit protocol actions contain the type of record, the transaction id, an the identity of the coordinator. A coordinators commit or log record, also contains the identifies of the subordinates.

(2) Restart after a failure

When a site comes back up after a crash, we invoke a recovery process that reads the log and processes all the transactions that were executing the commit protocol at the time of the crash. The transaction manager at this site could have been the coordinator for some of these transactions and a subordinate for others. We do the following in the recovery process.

- If we have a commit or abort log record for transaction T, its status is clear; we redo or undo T respectively. If this site is the coordinator, which can be determined from the commit or abort log record, we must periodically resend, because there may be other link or site failures in the system. A commit or abort message to each subroutine until we receive an ack. After we have received acks from all subordinates, we write an end log record for T.
- If we have prepare log record for T but no commit or abort log record, this site is a subordinate and the coordinator can be determined from the prepare record. We must repeatedly contact the coordinator site to determine the status of T. once the coordinator responds with either commit or abort, we write a corresponding log record, redo or undo the transactions; and then write an end log record for T.
- If we have no prepare, commit or abort log record for transaction T, T certainly could not have voted to commit be before the crash; so we can unilaterally abort undo undo T and write an end log record. In this case

we have no way to determine whether the current site is the coordinator or a subordinate for T. however, if this site is the coordinator it mite have sent a prepare message prior to the crash, and if so, other sites may have votes yes. If such a subordinate site contacts the recovery process at the current site, we now know that the current site is the coordinator for T, and given that there is no commit or abort log record, the response to the subordinate should be to abort T.

Observe that if the coordinator site for a transaction T fails, subordinates who have voted yes cannot decide whether to commit or abort T until the coordinator site recovers, we say that T is blocked. In principle, the active subroutine sites could communicate among themselves, and if at least one of them contains abort or commit log record for T, its status becomes globally known. In order to communicate among themselves, all subordinates must be told the identify of the other subordinates at the time they are sent the prepare message. However 2PC is still vulnerable to coordinator failure during recovery because even if all subordinates at the time they are sent the prepare message. However, 2PC is still vulnerable to coordinator failure during recovery because even if all subordinates have voted yes, the coordinator may have decided to abort T, and this decision cannot be determined until the coordinator site recovers.

(3) Two-phase Commit Revisited

Now that we have examined how a site recovers from a failure, and seen the inter action between the 2PC protocol, it is instructive to consider how 2PC can be refined further. In doing so, we will arrive at a more efficient version of 2PC, but equally important perhaps, we will understood the role of the various steps of 2PC more clearly. There are three basic observations.

1. The ack messages in2PC are used to determine when a coordinator can forget about a transaction T. until the coordinator knows that ll subordinates are aware of the commit/abort decision for T, it must keep information about T in the transaction table.
2. If the coordinator site fails after sending out prepare messages but before writing a commit or abort log record, when it comes backup it has no information about the transaction's commit status prior to the crash. However, it is still free to abort the transaction unilaterally (because written a commit record, it can still cast a no vote itself). If another site inquires about the status of the transaction, the recovery process as we have seen, responds with an abort message. Ths, in the absence of information, a transaction is presumed to have aborted.
3. If a subroutine does no updates, it has no changes to either redo or undo; in other words its commit or abort status is irrelevant.
    - When a coordinator aborts a transaction T, it can undo T and remove it from the transaction table immediately. After all, removing T from the table results in a 'no information' state

with respect to T, and the default response in this state which is abort, is the correct response for an aborted transaction.

- By the same token, if a subordinate receives an abort message, it need not waiting to hear from subordinates after sending an abort message does not receive an abort or commit message for a specified time-out interval, it will contact the coordinator again. If the coordinator decided to abort, there may no longer be an entry in the transaction table for this transaction, but it will receive the default abort message, which is the correct response.
- Because the coordinator is not waiting to hear from subordinates after deciding to abort a transaction, the names of subordinates need not be recorded in the abort log record for the coordinator.
- All abort log records can simply be appended to the log tail, instead of doing a force-write. After all, if they are not written to stable storage before a crash the default decision is to abort the transaction.

The third basic observation suggests some additional refinements:

- If a sub transaction does not updates the subroutine can respond to a prepare message from the coordinator with a reader message, instead of yes or no. the subordinate writes no log records in this case.
- When a coordinator receives a reader message, it treats the message as yes vote, but with the optimization that it does not send any more messages to the subordinate, because the subordinates commit or abort status is irrelevant.
- If all the subtransactions, including the sub transaction at the coordinator site, send a reaer message, we don't need the second phase of the commit protocol. Indeed, we can simply remove the transaction from the transaction table, without writing any log records at any site for this transaction.

The two phase commit protocol with the refinement discussed in this section is called two phase commit protocol.

(4) Three phase Commit

A commit protocol called three-phase commit (3PC) can avoid blocking even if the coordinator site fails during recovery. The basic idea is that when the coordinator sends out prepare messages and receives yes votes from all subordinates, it sends all sites a precommit message, rather than a commit message, when a sufficient number more than the maximum number of failures that must be handled-of acks have been received, coordinator force-write a commit lo record and sends a commit message to all subordinates. In 3PC the coordinator effectively postpones the decision to commit until it is sure that enough sites know about the decision to commit; if the coordinator subsequently fails, these sites can communicate with each other and detect that

the transaction must be committed message – without waiting for the coordinator to recover.

The 3PC protocol imposes a significant additional cost during normal execution and requires that communication link failures do not lead to a network partition in order to ensure freedom from blocking. For this reasons, it is not used in practice.

Self-Assesment Questions – VIII

1. Fragmentation _____.
2. Server will _____.
3. Data can be transferred across various servers is _____.
4. Expansion of ODBC is _____.
5. Capture will
    a) remove the changes
    b) accept the changes
    c) modify the changes
6. Replication means
    a) store several copies of a relation
    b) store one copies of a relation
    c) delete several copies of a relation

Sample questions

7. Write notes about parallel query evolution.
8. Write down the parallel query optimization.
9. Explain distributed architecture?
10. Explain how to update distributed data?
11. Write note about distributed concurrency control.

Answers for Self-Assesment Questions – VIII

1. Breaking a relation into smaller relation
2. Response the client's request
3. Middleware
4. Open Database Connectivity
5. b – accept the changes
6. a – store several copies of a relation

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

9.Object Database Systems

Object Database systems have developed along two distinct paths.

Object Oriented Database System: Object Oriented Database Systems are proposed as an alternative to relational systems and are aimed at application domains where complex objects play a central role. The approach is heavily influenced by object oriented programming languages and can be understood as an attempt to add DBMS functionality to a programming language environment.

Object Relational Database: Object Relational Database system can be thought to as an attempt to extent relational database system with the functionality necessary to support a broader class of application and in many ways, provide bridge between the relational and object-oriented paradigms.

9.1 Motivating Examples

New Data types

User-defined Data types: Dinky's assets include Herbert's image, voice, and video footage, and these must be stored in the database. To handle these new types, we need to be able to represent richer structure. Further, we need special functions to manipulate these objects. For example, we may want to write functions that produce a compressed version of an image or a lower resolution image.

Inheritance: As the number of data types grows, it is important to take advantage of the commonality between different types. For example, both compressed images and lower-resolution images are, at some level, just images. It is therefore desirable to inherit some features of image objects wile defining compressed image objects and lower-resolution image objects.

Object Identity: Given that some of the new data types contain very large instances, it is important not to store copies of objects; instead, we must store references, or pointers, to such objects.

Here we discuss some basic examples used in DBMS:

1. CREATE TABLE Frames
   (frameno integer, Image jpeg-image, catalog integer);
2. CREATE TABLE Categories
   (cid integer, name text, lease_price float, comments text);
3. CREATE TYPE theater_t AS
   ROW(ino integer, name text, address text, phone text)
   REF IS SYSTEM GENERATED;
4. CREATE TABLE Theaters OF theater_t REF is tid SYSTEM
   GENERATED;
5. CREATE TABLE Nowshowing
   (film integer, theater REF(theater_t) SCOPE Theaters, start
   date, end date);

6. CREATE TABLE Films

(filmno integer, title text, stars VARCHAR(25), ARRAY[10],

director text, budget float);

7. CREATE TABLE Countries

(name text, boundary polygon, population integer, language

text);

users to define arbitrary new data types is a key feature of ORDBMS. The DBMS allows users to store and retrieve objects of type jpeg-image, just like an object of any other type, such as integer. New atomic data types usually need to have type-specific operations defined by the user who creates them. For example, one might crop. The combination of an atomic data type and its associated methods is called an abstract data type, or ADT. Traditional SQL comes with built in ADTs, such as integers or strings. Object relational systems include these ADTs and also allow users to define their own ADTs.

The label 'abstract' is applied to these datatypes because the database system does not need to know how an ADT's data is stored or how the ADTs methods work. It merely needs to know that methods are available and the input and output types for the methods. Hiding of ADT internals is called encapsulation. Note that even in a relational system, atomic types such as integers have associated methods that are encapsulated into ADTs. In the case of integers the standard methods for the ADT are the usual arithmetic operators and comparators. To evaluate the additional operator on integers, the database system need to know how to invoke the laws of addition – it merely needs to know how to invoke the additional operators code and what type of data to expect in return.

In an object-oriented system, the simplification due to encapsulation is critical because it hides any substantive distinctions between data types and allows an ORDBMS to be implemented without anticipating the types and methods that users might want to add.

9.2 Structured Data types

Atomic types and user-defined types can be combined to describe more complex structures using type constructors. The set of syntax is an example of constructor. Other common type constructors include:

Row(n1t1,…..nntn): A type representing a row, or a\tuple of n fields n1….n1 of types t1,…..tn respectively.

Listof(base): A type representing a sequence of base-type items.

ARRAY(base): A type representing an array of base-type items.

Setof(base): A type representing a set of base-type items. Sets cannot contain duplicate elements.

Bagof(base): A type representing a bag or multiset of base-type items.

To fully appreciate the power of type constructors, observe that they be composed; for example, ARRAY(Row(age:integer, sal:integer)). Types using

listof, ARRAY, bagof or setoff as the outermost type construction are sometimes referred to as collection types, or bulk data types.

9.3 Operations on Structured Data

Manipulating Data Structured Types

The DBMS provides built-in methods for types supported through type constructors. These methods are analogous to built-in operations such as additional and multiplication for atomic typres such as integer.

Built-in operators for structured types

Rows: given an item I whose type is row(n_1t_1,….n_nt_n), the field extraction method allows us to access an individual field n_k, using the traditional dot notation i.n_k. if now constructors are nested in a type definition, dots may be nested to access the fields of the nested row; for example, i.n_k.m_1. if we have a collection of rows, the dot notation gives us a collection as a result. For example, if it is a list of rows, t_n ; if I is a set of rows, i.n_k gives is a set of items of type t_n.

This nested-dot notation is often called a path expression because it describes a path through the nested structure.

Sets and Multisets: Set objects can be compared using the traditional set methods <,<=,=,>,>=. Two set objects can be combined to form a new object using the U, n and – operators.

Each of the methods for sets can be defined for multisets, taking the number of copies of elements into accounts. The U operation is simply adds up the number of copies an element, the n operation counts the lesser number of items a given element appears in the two input multi sets, and - subtracts the number of times a given element appears in the second multiset from the number of times it appears in the first multi set. For example, using multi set semantics U({1,2,2,2},{2,2,3}) = {1,2,2,2,2,2,3}; n(1,2,2,2},{2,2,3})= {2,2}; and –({1,2,2,2} ,{1,2})={1,2}.

Lists: Traditional list operations include head, which returns the first elements, tail, which returns the list obtained by removing the first element; prepend, which takes an element and inserts it as the first element in a list; and append, which appends one list to another.

Arrays: Array types support an array index method to allow users to access array items at a particular offset. A postfix ‘square bracket’ syntax is usually used, for example, foo-array[5].

Others: the operators listed above are just a sample. We also have the aggregate operators count, sum, avg, max and min, which can be applied to any object of a collection type. Operation for type conversions are also common. For example, we can provide operators to converts a multicast objects to a set object by eliminating duplicate.

9.4 Encapsulation and ADTs

The combination of atomic data type and its associated methods is called an abstract data type(ADT). Traditional SQL comes with built-in ADTs, such as integer or strings.

The label abstract is applied to these datatypes because the database system does not need to know how an ADTs data is stored nor how the ADTs methods work. It merely need to know what methods are available and input and output types for the methods. Hiding ADT internals is called encapsulation. Note that even a relational system, atomic types such as integers have associated methods that encapsulate them. In the case of integers, the standard methods for the ADT are the usual arithmetic operators and comparators. To evaluate the addition operator on integers, the database system need not understand the laws of addition – it merely needs to know how to invoke the addition operator's code and what type of data to expect in return.

9.5 Inheritance

In object-database systems, unlike relational systems, inheritance is supported directly and allows type definitions to be reused and refined very easily. It can be very helpful when modeling similar but slightly different classes of objects. In object database systems, inheritance can be used in two ways for reusing and refining types, and for creating hierarchies of collections of similar but not identical objects.

Defining types with inheritance

In the Dinky database we model movie theaters with the type theater_t. Dinky also wants their database to represent a new marketing technique in the theater business, the theater_cafe, which serves pizza and other meals while screening movies. Theater_cafes require additional information to be represented in the database. In particular, a theater_cafe is just like a but an additional attribute representing the theater's menu. Inheritance allows us to capture this specialization explicitely in the database design with the following DDL statement:

```
CREATE TYPE theatercafe_t UNDER theater_t(menu text);
```

This statement creates a new type, theatercafe_t, which has the same attributes and methods are theater_t, along with one additional menu of type text. Methods defined on theater_t apply to objects of type theatercafe_t, but not vice versa. We say that theatercafe_t inherits the attributes and methods of theater_t.

Note that the inheritance is not merely a macro to shorten CREATE statements. It creates an explicit relationship in the database between the subtype (theatercafe_t) and the super type (theater_t). an object of the subtype is also considered to be an object of the supertype. This treatment means that any operations that apply to the supertype also apply to the sub type. This is generally expressed in the following principle:

The Substitution Principle: Given a super type A and subtype B, it is always possible to substitute an object of type B into a legal expression written for object of type A, without producing type errors.

This principle enables easy code reuse because queries and methods written for the supertype can be applied to the subtype without modification.

Note that inheritance can also be used for atomic types, in addition to two types. Given a super type image_t with methods title(), number-of-colors(), and display(), we can define sup type thumbnail-image_t for small images that it inherits the methods fro image_t.

Building of Methods

In defining a sub type, it is sometimes useful to replace a method for the super type with the new version that operates differently on the sub type. Consider the image_t type, and the subtype jpeg_image_t from the Dinky database. Unfortunately, the display() method for standard images does not work for JPEG images, which are specifically compressed. Thus, in creating type jpeg_image_t, we write a special display() method for JPEG image and register it with the database system using the CREATE FUNCTION command:

CREATE FUNCTION display(jpeg_image) RETURNS jpeg_image

AS EXTERNAL NAME '/a/b/c/jpeg.class' LAN-GUAGE ' java';

Registering a new method with the same name as an old method is called Overloading the method name.

Because of overloading, the system must understand which method is intended in a particular expression. For example, when the system needs to invoke the display() method on an object of type jpeg_image_t, it uses specialized display method. When it needs to display on an object of type image_t that is not otherwise sub typed, it invokes the standard display method. The process of deciding which method to invoke is called binding the method to the object. In certain situations this binding can be done when an expression is parsed, but in other case \s the most specific type of an object cannot be known until runtime, so the method cannot be bound until then. Late binding facilities add flexibility, but can make it harder for the user to reason about the methods that get invokes for a given query expression.

Collection Hierarchies, Type Extents and Queries

Type inheritance was invented for object-oriented programming language and our discussion of inheritance up to this point differs little form the discussion one might find in a book on an object-oriented language such as C++ or Java.

However, because database systems provide query languages over tabular datasets, the mechanisms from programming languages are enhanced in object database to deal with tables and queries as well. In particular, in object-relational systems we can define a table containing object of particular type, such as the theaters table in the Dinky schema. Given a new sub type such as

theater_cafe, we would like to create another table theater_cafe_t to store the information about theater_cafes. But when writing a query over the theaters table, it is sometimes desirable to ask the same query over the theater_cafes table, after all , it use project out the additional columns, an instance of the theater_cafes table can be regarded as an instance of the theater table.

Rather than requiring the user to specify query for each such table, we can inform the system that a new table of the such type is to be treated as part of a table of the super type, with respective queries over the later table. In our example, we can say:

CREATE TABLE theater_cafes OF TYPE theater_cafe_t UNDER theaters

This statement tells the system that queries over the theaters table should actually be run over all types in oth the theaters and theater_cafes tables. In such cases, if the sub type definition involves method overloading, late binding is used to ensure that the appropriate methods are called for each tuple.

In general, the UNDER clause can used to generate an arbitrary tree of tables, called a collectionhierarchy. Queries over a particular tree T in the hierarchy are run over all tuples in T and its descendentss. Sometimes, a user may want the query to run only on T and not on the descendents; additional syntax, for example the keyword ONLY, can be used in the queries FROM clause to achieve this effect.

Some systems automatically create special tables for each type, which contain references to every instances of the type that exists in the database. These tables are called type extents and allow queries over all objects of a given type, regardless of where the objects actually reside in the database. They extents naturally from a collection hierarchy that parallels the type hierarchy.

9.6 Objects, Object identity and Reference types

In object, database systems data objects can be given an object identifier, which is some value that is unique in the database across time. The DBMS is responsible for generating oids and ensuring than an oid identifies an object uniquely over its entire lifetime. In some systems, all tuples stored in any table are objects and are automatically assigned unique oids; in other system, a user can specify the table for which the tuples are to be assigned oids. Often, there are also facilities for generating oids for larger structures as well as smaller structures.

An objects oid can be used to refer to it from elsewhere in the data. Such a reference has a type (similar to the type of a pointer in a programming language), with a corresponding type constructor.

Ref(base): a type representing a reference to an object of type base.

The ref type constructor can be interleaved with the type constructors for structured types, for example, Row(ref(ARRAY(integer))).

Notations of equality

The distinction between reference types and reference-tree structured types raises another issue: the definition of equality. Two objects having the same type are defined to be deep equal if and only if.

The objects are of atomic type and have same value, or

The objects are of reference type, and the deep equals operator is true for the two referenced objects, or

The objects are of structured type, and the deep equals operator is true for all the corresponding subparts of the two objects.

Two objects that have the same reference type are defined to be shallow equal if they both refer to the same object (ie., both references use the same oid). The definition of shallow equality can be extended to objects of arbitrary type by taking the definition of deep equality and replacing deep equals by shallow equals in parts 2 and 3.

9.7 Database Design for an ORDBMS

An ORDBMS supports a much better solution. First we can store the video as an ADT object and write methods that capture any special manipulation that we wish to perform. Second, because we are allowed to store structured types. Such as lists, we can store the sequence for a probe in a single tuple, along with the video information1 this layout eliminates the need for joins in queries that invoke both the sequence and video information. An ORDBMS design for our example consists of a single relation called probes_AllInfo:

Probes_AllInfo (pid:integer, ocseq:location-seq, camera:string, video: mpeg tream)

This definition involves two new types, location-seq and mpeg_stream. The mpeg_stream type is defined as an ADT with a method display() that takes a start time and an end time and displays the portion of the video recorded during that interval. This method can be implemented efficiently by looking at the total recording duration and the total length of the video and interpolating to extract the segment recorded during the interval specified in the query.

Our first query is shown below in extended SQL syntax; suing this display method. We now retrieve only the required segment of the video, rather than the entire video.

SELECT display(P.video,1:10p.m May 10 1996, 1:15p.m May 10 1996)

FROM Probes_AllInfo P

WHERE P.pid=0

Now consider the location_seq type. We would define it as a list type containing a list of row type objects:

CREATE TYPE location_seq listof

(row(time:timestamp, lat:real, long:real))

Consider the locseq field in a row for a given probe. This field contains a list of rows, each of which has three fields. If the ORDBMS implements collection types in their fullgenerality, we should be able to extract the time column from this list to obtain a list of timestamp values, and to apply the MIN aggregate operator to this list to find the earliest time at which the given probe ecorded. Such support for collection types would enable us to express out second query as shown below:

SELECT P.pid, MIN(P.locseq.time)

FROM Probes_AllInfo P

Current ORDBMS are not as general and clean as this example query suggests. For instance, the system may not recognize that projecting the time column from a list of rows gives us a list of timestamp values; or the system may allow us to apply an aggregate operator only to a table and not nested list value.

Object Identity

We now discuss some of the consequences of using refernce type if ids. The use of oids is especially significant because it is a structured data type or because it is a big object such as an image. Although reference types and structured types seen similar, they are actually quite different.

- **Deletion:** Objects with references can be affected by the deletion of objects that they reference; while reference-free structured objects are not affected by deletion of other objects. For examples, the theaters table were dropped from the database an object of type theater might change value to null, because the theater_t object that it refers to has been deleted, while a similar object of type my_theater would not change value.
- **Update:** Objects of reference types will change value if the referenced object is updated. Objects of reference-free structured types change value only if updated directly.
- **Sharing versus Copying:** An identified object can be referenced by multiple reference- type items, so that each update to the object is reflected in many places. To set a similar affect in reference-free types requires updating all copies of an object.

There are also important storage distinctions between reference types and non reference types, which might affect performance:

- **Storage Overhead:** Storing copies of a large value in multiple structured type objects may use more space than storing the value once and referring to it elsewhere through reference type objects. This additional storage requirement can affect both disk usage and buffer management (if many copies are accessed at once).
- **Clustering:** the subparts of a structured object are typically stored together on disk. Objects that are far away on the disk, and the disk arm may require significant movement to assemble the object and it

references together. Structured objects can thus be more efficient than reference types if they are typically accessed is their entirely.

Many of these issues also arise in traditional programming languages such as C or Pascal, which distinguish between the notions of referring to objects by value and by reference. In database design the choice between using a structured type or a reference type will typically include consideration of the storage costs, clustering issues and the effect of updates.

9.8 ORDBMS Implementation Challenges

Storage and Access Methods

Since object-oriented database store new types of data, ORDBMS implementors need to revisit some of the storage and indexing issues.

Storing Large ADT and Structured Type Objects

Large ADT objects and structured objects complicate the layout of data on disk. User defined ADTs can be quite large. In particular, they can bigger than a single page. Large ADts like BLOBs require special storage, typically in adifferent location on disk from tuples that contain them.

Structured object can also be large, but unlike ADT objects, they often vary in size during the lifetime of a database. For example, consider the star attribute of the film table. As the year pass, some of the bit actors in an old movie may become famous. When a bit actor becomes famous, Dinky might want to advertise his or her presence in the earlier flimes.

This includes an insertion into the stars attribute of an individual tuple in flims. Because these bulk attributes can grow arbitrary, flexible disk layout mechanisms are required.

Query Processing

ADTs and structured types call for new functionality in processing queries in ORDBMS. They also change a number of assumptions that affect the efficiency of queries. So all the queries can be processed in an effective manner by using the following methods.

- User-defined Aggregation function
- Method Security
- Method Caching
- Pointer Swizzling

Query Optimization

To handle the new query processing functionality, an optimizer must know about the new functionality and use it appropriately. Here two issues in exposing information to the optimizer and an issue in the query planning that was ignored in the relational system.

Different optimizing methods

- Registering Indexes with the optimizer
- Reduction factor and Cost estimation for ADT methods
- Expensive Selection Optimization

9.9 OODBMS

We define OODBMS as a programming language with support for persistent objects. While this definition reflect the origins of OODBMSs accurately, and to a certain extent the implementation focus of OODBMS support collection types makes it possible to provide a query language over collections. Indeed, a standard has been developed by the Object Database Management Group (ODMG) and is called Object Query Language, or OQL.

The ODMG data model is the basis for an OODBMS, just like the relational data model is the basis for an RDBMS. A database contains a collection of objects, which are similar to entities in the Er model. Every object has a unique oid, and a database contains collections of objects with similar properties; such a collection is called a class.

The properties of a class are specified using ODL and are of three kinds. Attributes, relationship and methods. Attributes have an atomic type or a structured type. OLD supports the set, bag, list, array and struct type constructors.

Relationships have a type that is either reference to an object or a collection of such references. A relationship captures how an object is related to one or more objects of the same class or of a different class. A relationship in the ODMG model is really just a binary relationship in the sense of the ER model. A relationship has a corresponding inverse relationship; intuitively, it is the relationship 'in the other direction'.

Methods are functions that can be applied to objects of the class. There is no analog to methods in the ER or relationship models.

The keyword interface is used to define a class. For each interface, we can declare an extent, which is the name for the current set of objects of that class. The extent is analogous to the instance of a relation, and the interface is analogous to the schema. If the user does not anticipate the need to work with the set of objects of a given class - it is sufficient to manipulate individual objects the extent declaration can be omitted.

The following ODL definitions of the movie and theater classes illustrate the above.

Interface Movie

(extent Movies key moviename)

{attribute data start;

Attribute data end;

Attribute string moviename;

Relationship set <theater> shown at inverse theater: now showing:}

The collection of database objects whose class is movie is called Movies. No two objects in Movies have the same moviename value, as the key declaration indicates,. Each movie is shown at a set of theaters and is shown during the specified period. A theater is an object of class theater, which is defined below:

```
Interface Theater
(extent Theaters key theatername)
{attribute string theatername;
Attribute string address;
Attribute integer ticketprice;
Relationship set <Movie> now showing inverse
Movie :: ShownAt;
Float numshowing() raise(error counting Movies);
}
```

Each theater shown several movies and charges the same ticket price for every movie. Observe that the shownAt relationship of Movie and the nowshowing relationship of theater are declared to be inverse of each other. Theater also has a method numshowing() that can be applied to a theater object to find the number of movies being shown at that theater.

OQL

The ODMG query language OQL was deliberately designed to have syntax similar to SQL, in order to make it easy for users familiar with SQL to learn OQL. Let us begin with a query that fields pairs of movies and theaters such that the movie is shown at the theater and the theater is showing than one movie.

```
SELECT mname:M.moviename, tname:
T.theaterName
NAME Movie M, M.shownAt T
WHERE T.numshowing() > 1
```

The SELECT clause indicates how we can given names to fields in the result; the two result fields are called mname and tname. The part of this query that differs from SQL is the FORM clause. The variable M is bound in turn to each movie in the extent movies. For a given movie M, we bind the variable T in turn to each theater in the collection M.ShownAt. thus, the use of the path expression M.ShownAt allows us to easily express a nested query. The following query illustrates the grouping construct OQL.

```
SELECT T.ticketprice,
AvgNum:AVG(SELECT P.T.numShowing()
FROM partition P)
FROM Theaters  T
GROUP BY T.ticketprice
```

For each ticketprice, we create a group of theaters with the ticketprice. This group of theater is the partition for that ticket price and is referred to using OQL keyword partition. In the SELECT clause, for each ticketprice, we compute the average number of movies shown at theaters in the partition for that ticketprice.

9.10 Comparing RDBMS with OODBMS and ORDBMS

RDBMS Versus ORDBMS

Comparing and RDBMS with an ORDBMS is straightforward. The resulting simplicity of the data model makes it easier to optimize queries for efficient execution, for example, a relation system is also easier to use because there are fever features to master. On the other hand, it is less versatile than an ORDBMS.

OODBMS Versus ORDBMS: Similarities

OODBMSs and ORDBMSs both support user-define ADTs, a structured types, object identity and reference types, and inheritance. Both support a query language for manipulating collection types. ORDBMSs support ODL/OQL. The similarities are by no means accidental: OODBMSs in turn have developed query languages based on relational query languages. Both OODBMSs and ORDBMSs provide DBMS functionality such as concurrency control and recovery.

OODBMS Versus ORDBMS: Differences

The fundamental difference is really a philosophy that is carried all the way through: OODBMS try to add DBMS functionality to a programming language, whereas ORDBMS try to add richer data types to a relational DBMS.

The query facilities of OQL are not supported efficiently in most OODBMSs, whereas the query facilities are the center pieced of an ORDBMS. To some extent, this situation is the result of different extent, it is also a consequence of the systems being optimized for very different kinds of applications.

OODBMSs aim to achieve seamless integration with a programming language such as C++, Java or Smalltalk. Such integration is not an important goal for an ORDBMS.

Self-Assesment Questions – IX

1. Inheritance will _____.
2. combining data and functions into a single unit is_____.
3. Expansion of ORDBMS is _____.
4. Expansion of OODBMS is _____.
5. Expansion of ODM is
    a) Operator Data Model
    b) Object Domain Model
    c) Object Data Model

6. Expansion of OQL is
    a) Object queue Language
    b) Object Query Language
    c) Operator Query Language

Sample questions

7. Write notes about structured data types.
8. Write down the concept of inheritance.
9. Explain ORDBMS?
10. Differentiate RDBMS,OODBMS and ORDBMS?
11. Explain OQL?
12. Explain different operations in structured datatypes?

Answers for Self-Assesment Questions – IX

1. derive new class from the existing class
2. class
3. Object Relational Database Management System
4. Object Oriented Database Management System
5. c – Object Data Model
6. b – Object Query Language

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

10. Data Warehousing and Decision Support

Database management systems are widely used by organizations for maintaining data that documents their everybody operations. In applications that update such operational data, transactions typically make small changes and a large number of transactions must be reliably and efficiently processed. Such online transaction processing (OLTP) applications have driven the growth of the DBMS industry in the past three decades and will continue to be important. carrying the motivation for pre computed views one step further, organizations can consolidate information from several databases into a data warehouse by copying tables from many sources into one location or materializing a view defined over many specialized products are now available to create and manage warehouse of data from multiple databases.

10.1 Introduction to Decision Support

Organizational decision making requires a comprehensive view of all aspects of enterprise, so many organizations created consolidated data warehouses that contain data drawn from several database maintained by different business units together with historical and summary information.

Three broad classes of analysis tools are available.

- Some systems support a class of stylized. First, some systems support a class of stylized queries that typically involve group-by and aggregation operators and provide excellent support for complex Boolean conditions, statistical functions, and features for time-seies analysis. Applications dominated by such queries are called Online Analytic Processing (OLAP).
- Second, some DBMS support traditional SQL-style queries but are designed to also support OLAP queries efficiently. Such systems can be regarded as relational DBMS optimized for decision support applications.
- The third class of analysis tools is motivated by the desire to find interesting or unexpected trends and patterns in large data sets rather than the complex query characteristics just listed. In exploratory data analysis, although an analyst can recognize an interesting pattern when shown with the pattern, it is very difficult to formulate a query that capture the essence of an interesting pattern.

The amount of data in many applications is too large to permit manual analysis or even traditional statistical analysis, and the goal of data mining is to support exploratory analysis over very large data sets.

OLAP : Multidimensional Data Model

In multi dimensional data model, the focus is on a collection of numeric measures. Each measure depends on a set of dimensions. We use a running example based on sales data. The measure attribute in our example is sales. The dimensions are product, location, and time.

This view of data as a multidimensional array is readily generalized to more than three dimensions. In OLAP applications, the bulk of data can be represented in such a multidimensional array. Indeed some OLAP systems actually store data in a multidimensional array. OLAP systems that use arrays to store Multidimensional OLAP (MOLAP) systems.

The data in a multidimensional array can also be represented as a relation as illustrated in the following figure.

Locid	City	State	Country
1	Madison	WI	USA
2	Fresno	CA	USA
5	chenni	TN	India

Location

pid	pname	Category	Price
11	Lee james	Apparel	25
12	Zord	Toys	18
13	Biro pen	stationary	2

Products

pid	timeid	locid	Sales
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Salse

Fig. 10.1 Locations, Products and Sales Represented as Relations

Information about dimensions can also be represented as a collection of relations:

Locations(locid: integer, city: string \, state: string, country: string)

Products(pid: integer, pname: string, category: string, price: real)

Times(timeid: integer, date: string, week: integer, month: integer,

Quarter: inteer, year: integer, holiday_flag: Boolean)

These relations are much smaller than the fact table in a typical OLAP applications; they are called the dimension tables. OLAP systems that store all information, including fact tables, as relations are called relational OLAP (ROLAP) systems.

10.3 Multidimensional Aggregation Queries

The operations supported by this model are strongly influenced by end user tools such as spreadsheets. A very common operation is aggregating a measure over one or more dimensions. The following queries are typical:

- Find the total sales.
- Find total sales for each city.
- Find total sales for each state.

These queries can be expressed as SQL queries over the fact and dimensions tables. When we aggregate a measure on one or more dimensions, the aggregated measure depends on fewer dimensions than the original measure.

Another use of aggregation is to summarize at different levels of a dimension hierarchy. If we a re given total sales per city, we can aggregate on the location dimension to obtain sales per state. This operation is called roll-up in the OLAP literature. The inverse of roll-up is drill-down: given total sales by state, we can ask for a more detailed presentation by drilling down on location.

Another common operation is pivoting. Consider a tabular presentation of the sales table. If we pivot it on the location and time dimensions, we obtain a table of total sales for each location for each time value. This information can be presented as two-dimensional chart in which the axes are labeled sales for that location and time. Therefore, values that appear in columns of the original presentation become labels of exes in the result presentation. The result of pivoting, called a cross-tabulation.

	WI	CA	Total
1995	**63**	**81**	144
1996	**38**	**107**	145
1997	**75**	**35**	110
Total	176	223	399

Fig: Cross-tabulation of sales by year and state

The OLAP framework makes it convenient to pose a broad class of queries. It also gives catchy names to some familiar operations: Slicing a dataset amounts to an equality selection on one or more dimensions, possibly also with some dimensions projected out. Dicing a dataset amounts to a range a cube or cross-tabulated representation of the data.

A note on Statistical Database

Many OLAP concepts are present in earlier work on statistical databases (SDBs), which are database systems designed to support statistical applications, although this connection has not been sufficiently recognized because a differences in application domains and terminology. The multi dimensional data model, with the notions of a measure associated with dimensions and classification hierarchies for dimension values, is also used in SDBs. OLAP operations such as roll-up and drill-down have counter parts in SDBs.

ROLLUP and CUBE in SQL:1999

A single OLAP operation leads to several closely related SQL queries with aggregation and grouping. For example, from the cross-tabulation table , we obtain by pivoting the sales table. To obtain same information, we would issue the following query:

```
SELECT T.year, L.state, SUM(S.sales)
FROM   Sales S, Times T, Location L
WHERE  S.timeid=T.timeid AND S.locid=L.locid
GROUP BY T.year, L.state
```

This query generates the entries in the body of the chart. The summary column on the right is generated by the query:

```
SELECT T.year, L.state, SUM(S.sales)
FROM   Sales S, Times T,
WHERE  S.timeid=T.timeid
GROUP BY T.year
```

The summary row at the bottom is generated by the query:

```
SELECT  L.state, SUM(S.sales)
FROM   Sales S,  Location L
WHERE  S.locid=L.locid
GROUP BY  L.state
```

The cumulative sum in the bottom-right corner of the chart is produced by the query:

```
SELECT SUM(S.sales)
FROM   Sales S, Location L
WHERE  S.locid=L.locid
```

10.4 WINDOW Queries in SQL:1999

The time dimension is very important in decision support and queries involving trend analysis have traditionally been difficult to express in SQL. To address this, SQL:1999 introduced a fundamental extension called query window. Examples of queries that can be written using this extension, but are either difficult or impossible to write in SLQ without it, include,

1. Find total sales by month.
2. Find total sales by month for each city.
3. Find the percentage change in the total monthly sales for each product.
4. Find the top five products ranked by total sales.
5. Find the trailing n day moving average of sales.
6. Find the top five products ranked by cumulative sales, for every monthover the past year.
7. Rank all products by total sales over the past year, and for each product, print the difference in total sales relative to the product ranked behind it.

There is some similarities from GROUP BY and CUBE clauses, there are important differences as well. For example, like the window operator, GROUP BY allows us to create partitions of rows and apply aggregate functions such as SUM to the rows in the partitions. However unlike windows there is a single output row for partition rather than one output row for each row and each partition is a unordered collection of rows.

We now illustrate the window concept through an example:

```
SELECT L.state, T.month, AVG(S.sales) OVER AS movavg
FROM     Sales S, Times T, Location L
WHERE  S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state ORDER BY T.month
RANGE BETWEEN INTERVAL ‘1’  MONTH PRECEDING
AND INTERVAL ‘1’ MONTH FOLLOWING)
```

The FROM and WHERE clauses are processed as usual to generate an intermediate table, which we refer o as temp. windows are created over the temp relation.

There are three steps in defining window, first we define partitions of the table, using the PARTITION BY clause, in the example, partitions are based on the L.state column. Partitions are similar to groups created with GROUP BY, but there is a very important difference in how they are processed. To understand the difference observe that the SELECT clause contains a column, T.month , which is not ued to define the partition, different rows in a given partition could have different values in this column. Such a column cannot appear in a SELECT clause in conjunction with grouping, but it is allowed for partitions.

The second step in defining a window is to specify the ordering of rows within a partition. We do this using the ORDERED BY clause, in the example, the rows within each partition are ORDERED BY T.month.

The third step in window definition is to frame windows , that is to establish the boundaries of window associated with each row in terms of the ordering of rows within partition.

Framing a window

There are two distinct ways to framing a window in SQL:1999. the example query illustrated the RANGE construct, which defines a window based on the values in some column. The ordering column has to be a numeric type, a date time type, or an interval type since these are the only types for which addition and subtraction are defined.

The second approach is based on using the ordering directly and specifying how many rows before and after the given row are in its window. Thus we could say

```
SELECT L.state, T.month, AVG(S.sales) OVER AS movavg
FROM   Sales S, Times T, Location L
WHERE  S.timeid=T.timeid AND S.locid=L.locid
WINDOW W AS (PARTITION BY L.state ORDER BY T.month
ROWS BETWEEN  1  PRECEDING AND 1 FOLLOWING)
```

If there is exactly one row in temp for each months, it is equivalent to the previous query. However, if a given month has no rows or multiple rows, the two queries produced different results. In this case, the result of the second query is hard to understand because the windows for different rows do not align in a natural way.

New Aggregate functions

While the standard aggregate functions that apply to multi sets of value can be used in conjunction with windowing, there is a need for a new class of function that operate on a list of values.

The RANK function returns the position of a row within its partitions. If a partition has 15 rows, the first row has RANK1 and the last row has rank15. The rank of intermediate rows depends on whether there are multiple rows for a given value of the ordering column.

10.5 Finding Answers Quickly

A recent trend, fueled in part by a popularity of the internet, is an emphasis on queries for which a user wants only the first few, or the best few, answers quickly. When users pose queries to a search engine such as Alta-Vista, they rarely look beyond the first or second page of results. If they do not find what they are looking for, they refined their query and resubmit it. The same phenomenon occurs in decisions support application and some DBMS products already support extended SQL construct to specify such queries. A related trend is that, for complex queries users would like to see an approximate answers quickly and they have it be continually refined, rather than wait until the exact answer is available. We now discuss these two trends briefly.

Top N Queries

An analyst often wants to identify top selling handful of products, for example. We can sort by sales for each product and return answers in this order. If we have a million product and analysts is interested only in top 10. this straight forward evaluation strategy is clearly wasteful. It is desirable for users to be able to explicitly indicate how many answers they want, making it possible for the DBMS to optimize execution. The following example query asks for the top 10 products ordered by sales in the given location and time.

```
SELECT P.pid, P.pname, S.sales
FROM   Sales S, Products P
WHERE  S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY S.sales DESC
OPTIMIZE FOR 10 ROWS
```

The OPTIMIZE FOR N ROWS construct is not in SQL92, but is supported in IBM's DB2 products, and other products have similar products have similar construct. In the absence of a cue such as OPTIMIZE FOR 10 ROWS, the DBMS computes sales for all products and returns them in descending order by the sales. The application can close the result curser after consuming 10 rows, but considerable effort has already been expended in computing sales for all products and sorting them.

Online Aggregation

Consider the following query, which asks for the average sales amount by state.

```
SELECT  L.state, AVGS.sales)
FROM    Sales S,  Location L
WHERE  S.locid=L.locid
GROUP BY  L.state
```

This can be an expensive query, if sales and locations are large relations. W cannot achieve fast response times with traditional approach of computing the answer in its entirety when the query is presented. One alternative, as we have seen, is to use pre computation. Another alternative is to compute the answer to the query when the query is presented but return an approximate answer to the user as soon as possible. As the computation progresses the answer quality is continually refined. This approach is called Online Aggregation. It is very attractive for queries involving aggregation, because efficient technique for computing and refining approximate answers are available.

To implement online aggregation a DBMS must incorporate statistical technique to provide confidence intervals for approximate answers and use non blocking algorithms for the relational operators. An algorithm is said to block if it does not produce output tuples until it has consumed all its input tuples. For example, the sort-merge join algorithm blocks because sorting requires all input tuples before determining the first output tuple. Nested loops join and hash join

are therefore preferable to sort-merge join for online aggregation. Similarly hash based application is better than sort based aggregation.

10.6 Implementation Techniques for OLAP

In this section we survey some implementation techniques motivated by the OLAP environment. The goal is to provide a feel for how OLAP systems differ from more traditional SQL system, our discussion is far from comprehensive.

The mostly-read environment of OLAP systems makes the CPU overhead of maintaining indexes negligible and the requirement of interactive response times for queries over very large data sets makes the availability of suitable indexes very important. This combination of factors has let to the development of new indexing techniques. We discuss several of these techniques. We then consider file organizations and other OLAP implementations issues briefly.

Bitmap Indexes

Consider a table that describes customer:

Customers(custid: integer, name:string, gender:Boolean, rating:integer)

The rating value is an integer in the range 1 to 5 , and only two values are recorded for gender. Columns with few possible values are called Sparse. We can exploit sparsity to construct a new kind of index that greatly speeds up queries on these columns.

The idea is to record value for sparse column as a sequence of bit , one for each possible value. Fro example, a gender value is either 10 or 01, a 1 in the first position denotes male and 1 in the second position denotes female. Similarly 10,000 denotes the rating value 1 , and 00001 denotes the rating value 5.

If we considered the gender value for all rows in the customers table, we can treat this as a collection of two bit vector, one of which as the associated value M and other the associated vale F. each bit vector has one bit per row in the customers table, indicating whether the value in that row is the value associated with the bit vector. The collection of bit vectors for a column is called bitmap index for that column.

An example instance of the customers table, together with the bitmap indexes for gender and rating is shown in the figure.

M	F
1	0
1	0
0	1
1	0

custid	name	gender	Rating
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
112	Woo	M	4

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Fig. Bitmap indexes on the Customers Relation

Bitmap indexes offer two important advantages over conventional hash and tree indexes. First , they allow the use of efficient bit operation to answer queries. For example, considered the query, "how many male customers have the rating of 5?" we can take the first bit vector for gender and do a bitwise AND with the 5^{th} bit vector for rating to obtain a bit vector that has one for every male customer with rating 5. we can then count the number of ones in this bit vector to answer the query. Second, bitmap indexes can be much more compact than a traditional B+ tree index and are very amenable to the use of compression techniques.

This hybrid approach, which can easily be adopted to work with hash indexes as well as B+ tree indexes, as both advantages and disadvantages relative to a standard list of rids approach:

1. It can be applied even to columns that are not sparse: that is , in which a many possible values can appear. Index levels allows us to quickly find the list of rids in a standard list or bit vector representation, for a given key value.
2. Overall, the index is more compact because we can use a bit vector representation for long rid lists. We can have the benefits of fast bit vector processing.
3. On the other hand, bit vector representation of an rid list relies on a mapping from a position in the vector to an rid. If the set of rows is static, and we do not worry about inserts and deletes of rows, it is straightforward to ensure this by assigning contiguous rids for rows in a table. If inserts and deletes must be supported, additional steps are required. For example, we can continue to assign right contiguously on a per-table basis and simply keep track of which rid corresponds to deleted rows. Bit vector can now be longer than the current number

rows, and periodic organizations is required to compact the 'holes' in the assignment of rids.

Join Indexes

Computing joins with small response times is extremely hard for very large relation. One approach to this problem is to create an index designed to speedup specific join queries. Suppose that the customers table is to be joined with the table called purchase on the custid field. We can create a collection of <c,p> pairs , where p is the rid of purchases record that joins with a customers record with custid c.

File organizations

Since many OLAP query involves such a few columns of a large relation, vertical partitioning becomes attractive. However, storing a relation column-wise can degrade performance for queries that involves several columns. An alternative in a mostly read environment is to store the relation row-wise, but also store each column separately.

10.7 Data Warehousing

Data warehouses contain consolidated data from many sources, augmented with summary information and recovering a long time period. Warehouses are much larger than other kind of databases; sizes ranging from several giga bytes to tera bytes are common. Typical workloads involves ad hoc, fairly complex queries and fast response are important. These characteristics differentiate warehouse application from OLTP application, and different DBMS design and implementation techniques must be used to achieve satisfactory results. A distributed DBMs with good scalability and high availability is required for very large warehouses.

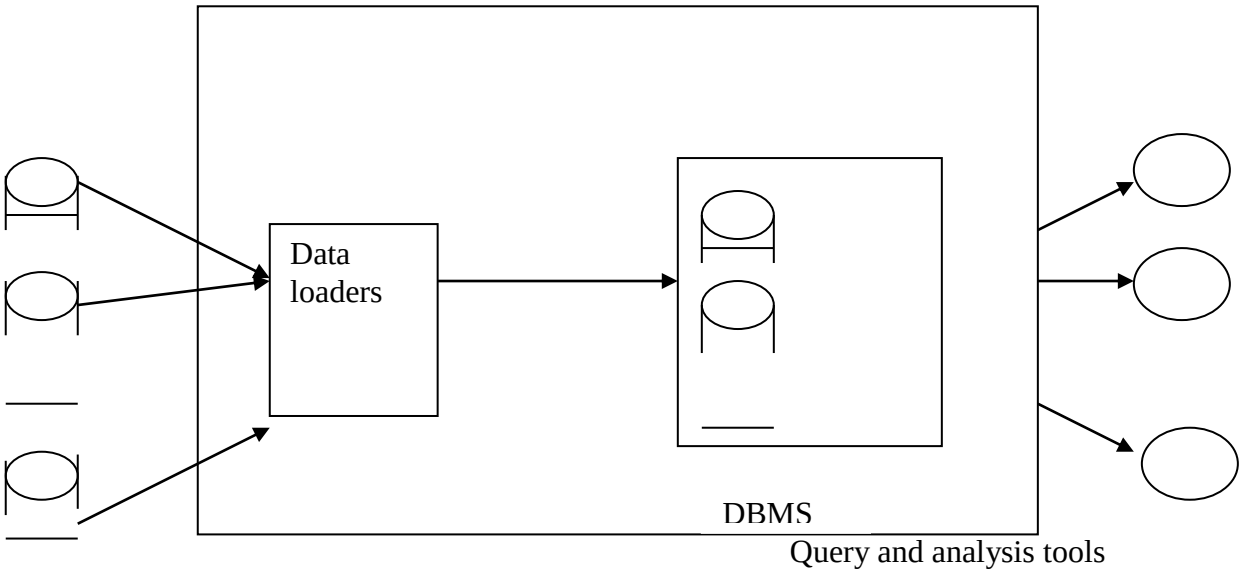


Figure 5.4.a Data Warehouse architecture

A typical data warehousing architecture is illustrated in the above figure. An organizations daily operations access and modify operational database. Data from these operational databases and other external sources are extracted by using interface such as JDBC.

Creating and Maintaining A Warehouse

Many challenges must be met in creating and maintaining a large data warehouse. A good database scheme must be designed to hold and integrated collection of data copied form diverse sources.

Data is extracted from operational database and external sources, cleaned to minimize errors and fill in missing information when possible, and transformed to reconcile semantic mismatches. Transforming data is typically accomplished by defining a relational views over the table in the data sources. Loading data consist of materializing such views and storing them in the warehouse. Unlike a standard view in a relational DBMS therefore, the view is stored on a database that is different from the database containing the table it is defined over.

The cleaned and transformed data is finally loaded into the warehouse. Additional preprocessing such as sorting and generation if summary information is carried out at this stage. Data is partitioned and indexes are built for efficiency. Due to the large volume of data loading is a slow processor. Loading a terabyte of data sequentially can take weeks, and loading even a gigabytes can take house. Parallelism is therefore important for loading warehousing.

After data is loaded into a warehouse additional measures must be taken to ensure that the data in the warehouse is periodically refreshed to reflect updates to the data sources and periodically purge old data. Observed the connection between the problem of refreshing warehousing table and asynchronously maintaining replicas of tables in a distributed DBMS.

The system catalogs associated with the warehouse are very large and often stored and managed in a separate databases called meta data repository. The size and complexity of the catalog is in part due to the size and complexity of warehouse itself and in part because a lot of administrative information must be maintained.

10.8 Views and Decision Support

Views are widely used in decision support application. Different groups of analysts with in an organization are typically concerned with different aspects of the business and it is convenient to define views that give each group insight into the business detail that concern it.

Views, OLAP, and Warehousing

Views are closely related to OLAP and data warehousing. Analysts wants fast answers to these queries over a very large data sets, and it is natural to consider pre computing views.

Queries over views

Consider the following views, regional sales , which computes sales of products by category and sales.

```
CREATE VIEW Regionalsales ( category, sales, state)
As SELECT P.category, L.state, S.sales
FROM     Sales S, Products P, Location L
WHERE  P.pid=S.pid AND S.locid=L.locid
```

The following query computes the total sales for each category by state.

```
SELECT R.category, R.state, SUM(R.sales)
FROM     Regionalsales R
GROUP BY R.category, R.state
```

While the sQL standard does not specify how to evaluate queries on views, it is useful to think interms of a process called query modification. The idea is to replace the occurrence of regional sales in the query by the view definition. The result on the query is:

```
SELECT R.category, R.state, SUM(R.sales)
FROM  (SELECT P.category, L.state, S.sales
         FROM     Sales S, Products P, Location L
        WHERE  P.pid=S.pid AND S.locid=L.locid) AS R
GROUP BY R.category, R.state
```

10.9 View materialization

We can answer a query on a view by using the query modification technique just describe. Often however queries against complex view definitions must be answered very fast because users encaged in decision support activities required interactive response time. Even with sophisticated optimization and evaluation techniques, there is a limit to how fast we can answer such queries also if the underlining table or in a remote database, the query modification approach may not even be feasible because of issues like connectivity and availability.

An alternative to query modification is to pre compute the view definition and store the result. When a query is posed on the view, the query is executed directly on the pre computed result. This approach called view materialization, is likely to be much faster than the query modification approach because the complex view need not be evaluated when the query is computed. Materialized views can be used during query processing in the same way as regular relation; for example, we can create indexes on materialized views to further speed up query processing. The drawback, of course, is that we must maintain the consistency of the pre computed view whenever the underlying tables are updated.

10.10 Maintaining Materialized Views

A materialized view is said to be refreshed when we make it consistent with changes to its underlying table. The process of refreshing a view to keep it

consistent with changes to the underlying table is often referred to as view maintenance.

Incremental View Maintenance

A straightforward approach to refreshing a view is to simplify re compute the view when an underlying table is modified. This may, in fact, be a reasonable strategy in some cases. Fro example, in the underlying table are in a remote database, the view can be periodically recomputed and send to data warehouse where the view is materialized. This has the advantage that the underlying table need not be replicated at the warehouse.

Whenever possible, however, algorithms for refreshing a view should be incremental, in that the cost is proportional to the extend of the change rather than the cost of re computing the view from scratch.

Join Views

Consider a view V defined as a join of two tables R S. suppose we modify R by inserting a collection of Rows Ri and deleting the collection of rows Rd. we compute Ri S and add the result to V. we also compute Rd S and subtract the result form V. observe that if r appears in Rd S with count c, it must also appear in V with the higher count.

Maintaining Warehouse Views

The views materialized in a data warehouse can be based on source table in remote databases. The synchronous replication techniques were discussed previously and it allows us to communicate changes at the source to the warehouse, but refreshing views incrementally in a distributed setting presents some unique challenges. To illustrate this, we consider a simple view that identifies supplier of Toys:

CREATE VIEW ToySupplier(sid)

AS SELECT S.sid

FROM Suppliers S, Products P

WHERE S.pid=P.pid AND P.category='Toys'

Suppliers is new table introduced for this example; Let us assume that it has just two fields, sid and pid, indicating that supplier sid, supplies part pid. The location of the tables products and suppliers and the view Toysuppliers influences how we maintain the view.

We could try to maintain the view incrementally as follows:

1. The warehouse site sends this updates to the source site.
2. To refresh the view, we need to check the suppliers table to find suppliers of the item, and so the warehouse site asks the source site for this information.
3. The source site returns the set of suppliers for the sold item, and warehouse site incrementally refreshes the view.

Suppose that products is empty and supplies contains just the row <s1,5> initially an consider the following sequences of events:

1. Product pid=5 is inserted with category='toys'; source notifies warehouse.
2. Warehouse asks source for suppliers of product pid=5.
3. The row <s2,5> is inserted into suppliers, source notifies warehouse.
4. To decide whether s2 should be added to the view, we need to know the category of the product pid=5, and warehouse asks source.
5. Source now processes the first query from warehouse, finds two suppliers for part 5, and returns this information to the warehouse.
6. Warehouse gets the answer to its first question: suppliers S1 and S2 , and add these to the view each with count 1.
7. Source processes the second query from warehouse and response with the information that part 5 is a toy.
8. Warehouse gets the answer to its second question an accordingly increments the count for suppliers S2 in view.
9. Product pid=5 is now deleted; source notifies warehouse.
10. Since the deleted part is a toy, ware house increments the counts of matching view tuples; S1 has count 0 and is removed but S2 has count 1 and is retained.

When Should be Synchronous Views?

A view maintenance policy is a decision about when a view is refreshed, independent of whether the refresh is incremental or not a view can be refreshed with in the same transaction that updates underlying table. This is called immediate view maintenance. The update transaction is slowed by the refresh step, and impact of refresh increases with the number of materialized view that depends on the update table.

Alternatively we can differ refreshing the view. Updates and captured in a log and applied subsequently to the materialized view. There are several deferred view maintenance policies:

Lazy: the materialized View V is refreshed at the tie of query is evaluated using V, if V is not already consistent with its underlying base table. This approach slows down queries rather than updates, in construct to immediate view maintenance.

Periodic: The materialized view is refreshed periodically, say once a day. The discussion of the capture and applied steps in asynchronous replication should be reviewed at this point. Since it is very relevant periodic view maintenance. In fact many vendors and extending their asynchronous replication features to support materialized views. Materialized views that are refreshed periodically are also called snapshots.

Forced: The materialized view is refreshed after a certain number of changes have been made to the underlying table.

Self-Assesment Questions – X

1. Data warehouse is used to _____.
2. Decision support system will _____.
3. Expansion of OLAP is _____.
4. Meta data is _____.

Sample questions

5. Write notes about Decision support system.
6. Write down the concept of finding answers quickly.
7. Explain multidimensional aggregation queries?
8. Explain Data warehousing with block diagram?
9. Explain view materialization?
10. How to maintain materialized views?
11. Explain views and decision support?

Answers for Self-Assesment Questions – X

1. Maintain organization's information
2. Make decisions automatically
3. Online Analytical Processing
4. Data about data

NOTES

……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..
……..

11. Data Mining

Data mining consists of finding interesting trends or patterns in large data sets to guide decisions about future activities.

Introduction to Data Mining

Data mining is related to the sub area of statistics called exploratory data analysis, which has similar goals and relies on statistical measures. It is also closely related to the sub area of artificial intelligence called knowledge discovery and machine learning. The important distinguishing characteristic of data mining is that the volume of data is very large; although ideas from these related areas of study are applicable to data mining problems, scalability with respect to data size is an important new criterion.

Finding useful trends in datasets is a rather loose definition of data mining: in a certain sense, all database queries can be thought of as doing just this. Indeed, we have a continuum of analysis and exploration tools with SQL queries at one end, OLAP queries in the middle, and data mining techniques at the other end.

The Knowledge Discovery Process

The Knowledge discovery and data mining (KDD) process can roughly be separated into four steps.

Data Selection: The target subset of data and the attributes of interest are identified by examining the entire raw dataset.

Data Cleaning: Noise and outliers are removed, field values are transformed to common units and some new fields are created by combining existing fields to facilitate analysis. The data is typically put into a relational format, and several tables might be combined in a de normalization step.

Data Mining: We apply data mining algorithms to extract interesting patterns.

Evaluation: The patterns are presented to end –users in an understandable form, for example, through visualization.

The result of any step in the KDD process might lead us back to an earlier step to read the process with the new knowledge gained.

11.2 Counting Co-occurrences

We begin by considering the problem of counting co-occurring items, which is motivated by problems such as market basket analysis. A market basket is a collection of items purchased by a customer in a single customer transaction. A customer transaction consists of a single visit to store, a single order through a mail order catalog, or an order at a store on the web. A common goal for retailers id to identify items that are purchased together. This information can be used to improve the layout of goods in a store or the layout of catalog pages.

Transid	Custid	Date	Item	Qty
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	5/10/99	Pen	1
113	106	5/10/99	Milk	1
114	201	6/1/99	Pen	2
114	201	6/1/99	Ink	2
114	201	6/1/99	Juice	4
114	201	6/1/99	water	1

Fig. the purchase relation

Frequent Item sets

We use purchases relation shown in the above figure to illustrate frequent item sets. The records are shown sorted into groups by transaction. All tuples in a group have the same transid, and together they describe a customer transaction, which involves purchases of one or more items. A transaction occurs on a given data, and the name of each purchased item is recorded, along with the purchased quantity. Observe that there is redundancy in purchase: it can be decomposed by storing transid-custid-date triples in a separate table and dropping custid and date from purchase: this may be how the data is actually stored. However, it is convenient to consider the purchase relation, as shown in the above figure, to compute frequent itemsets. Creating such a denormalized tables for ease of data mining is commonly done in the data cleaning step of the KDD process.

Extrapolation to future transactions should be done with caution, as discussed in the previous section. Let us begin by introducing the terminology of market basket analysis. An itemset is a set of items. The support of an itemset is the fraction of transactions in the database that contain all the items in the item set. We are interested in all itemsets whose support is higher than a user-specified minimum support called minsup; we call such itemsets frequent itemsets.

We show an algorithm for identifying frequent itemsets in the below figure. This algorithm relies on a simple yet property of frequent itemsets.

Foreach item,

Check if it is frequent itemset //appears in > minsup transaction

K=1

Repeat // iterative, level-wise identification of frequent itemsets

Foreach new frequent itemset Ik with k items, $I_k < I_{k+1}$

Scan all transactions once and check if the generated

K+1 itemsets are frequent

K=k+1

Until no new frequent itemsets are identified

Fig. an algorihm itemsets are identified

The a priori property: Every subset of a frequent itemset is also a frequent itemset.

The algorithm proceeds iteratively, first identifying frequent itemsets with just one time. In each subsequent iteration, frequent itemsets identified in the previous iteration are extended with another item to generate larger candidate itemsets. By considering only itemsets obtained by enlarging frequent itemsets, we greatly reduce the number of candidate frequent itemsets; this optimization is crucial for efficient execution. The a priori property guarantees that this optimization is correct, that is, we do not miss any frequent itemsets. A single scan of all transactions suffices to determine which candidate itemsets generated in an iteration are frequent. The algorithm terminates when no new frequent itemsets are identified in an iteration.

Iceberg Queries

We introduce iceberg queries through an example. Consider again the purchases relation shown in previous figure. Assume that we want to find pairs of customers and items such that the customer has purchased the item more than five times. We can express this query in SQL as follows:

SELECT P.custid, P.item, SUM(P.qty)

FROM Purchase P

GROUP BY P.custid, P.item

HAVING SUM(P.qty) > 5

Think about how this query would be evaluated by a relational DBMS. Conceptually, for each (custid, item) pair, we need to check whether the sum of the qty field is greater than 5. one approach is to make a scan over the purchase relation and maintain running sums for each (custid, item) pair. This is a feasible execution strategy as long as the number of pairs is larger than main memory, more expensive query evaluation plans, which involve either sorting or hashing, have to be used.

Therefore, we call such a query an iceberg query. In general, given a relational schema R with attributes A1,A2,….Ak, and B and anaggregation function aggr, an iceberg query has the following structure:

SELECT R.A1, R.A2,….R.Ak, aggr(R.B)

FROM Relation R

GROUP BY R.A1, …R.Ak

HAVING aggr(R.B) >= constant

Traditional query plans for this query that use sorting or hashing first compute the value of the aggregation function for all groups and then eliminate groups that do not satisfy the condition in the HAVING clause.

11.3 Mining for Rules

Many algorithms have been proposed for discovering various forms of rules that succinctly describe the data. We now look at some widely discussed forms of rules and algorithms for discovering them.

Association Rules

We use the purchase relation shown in the above figure to illustrate association rules. By examining the set of transactions in purchases, we can identify rules of the form:

{ pen} ➔ {ink}

There are two important measures for an association rule:

Support: The support for a set of items is the percentage of transactions that contain all these items. The support for a rule LHS➔RHS is the support for the set of items LHS u RHS. For example, consider the rule {pen} ➔{ink}. The support of this rule is the support of the item set {pen, ink}, which is 75%.

Confidence: Consider transaction s that contains all items in LHS. The confidence for a rule LHS➔RHS is the percentage of such transactions that also contain all items in RHS. More precisely, let sup(LHS) be the percentage of transactions that contain LHS and let sup(LSH U RHS) be the percentage of transactions that contain both LHS and RHS. Then the confidence of the rule LHS ➔RHS is sup(LHS U RHS) / sup(LHS).

Association rules and ISA Hierarchies

In many cases, an ISA hierarchy or category hierarchy is imposed on the set of items. In the presence of a hierarchy, a transaction contains, for each if its items, implicitly all the items ancestors in the hierarchy, the purchase relation is conceptually enlarged be the eight records shown in the following figure.

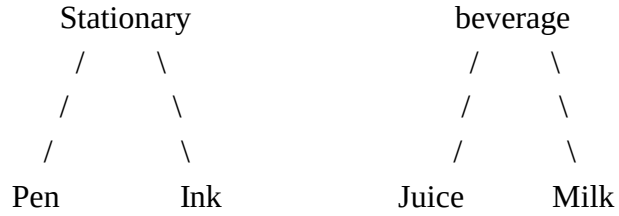


Fig. An ISA Category Taxonomy

Transid	Custid	Date	Item	Qty
111	201	5/1/99	Stationary	3
111	201	5/1/99	Beverage	9
112	105	6/3/99	Stationary	2
112	105	6/3/99	Beverage	1
113	106	5/10/99	Stationary	1
113	106	5/10/99	Beverage	1
114	201	6/1/99	Stationary	4
114	201	6/1/99	Beverage	5

Fig. Conceptual Additions to the Purchases Relation with ISA Hierarchy

Generalized Association Rule

Although association rules have been most widely studied in the context of market basket analysis, or analysis of customer transactions, the concept is more general. Consider the purchase relation as shown in the following figure, grouped by custid. By examining the set of customer groups, we can identify association rule such as {pen} ➔ {milk}. This rule should now be read as follows: "if a pen is purchased by a customer, it is likely that milk is also be purchased by that customer". In the purchases relation shown in the following figure, this rule has both support and confidence of 100%.

Transid	Custid	Date	Item	Qty
112	105	6/3/99	Pen	1
112	105	6/3/99	Ink	1
112	105	6/3/99	Milk	1
113	106	5/10/99	Pen	1
113	106	5/10/99	Milk	1
114	201	6/1/99	Pen	2
114	201	6/1/99	Ink	2
114	201	6/1/99	Juice	4
114	201	6/1/99	water	1
111	201	5/1/99	Pen	2
111	201	5/1/99	Ink	1
111	201	5/1/99	Milk	3
111	201	5/1/99	Juice	6

Fig. The Purchases Relation Sorted on Customer ID

Similarly, we can group tuples bt date and identify association rules that describe behavior on the same day. As an example consider again the purchasees relation. In this case, the rule {pen} ➔ {milk} is now interpreted as follows: "on a day when a pen is purchased, it is likely that milk is also be purchased;.

If we use the date field as grouping attribute, we can consider a more general problem called calendric market basket analysis. In calendric market basket analysis, the user specifies a collection of calendars. A calendar is any group of dates, such as every Sunday in the year 1999, or every first of the month. A rule holds if it hold on every day in the calendar. Given a calendar, we can compute association rules over the set of tuples whose date field falls within the calendar.

Sequential Patterns

Consider the purchase relation shown in the above figure. Each group of tuples having the same custid value, can be thought of as a sequence of transactions ordered by date. This allows us to identify frequently arising buying patterns over time.

We begin by introducing the concept of a sequence of itemsets. Each transaction is represented by a set of tuples, and by looking at the values in the item column, we get a set of item purchased in that transaction. Therefore, the sequence of transactions associated with a customer corresponds naturally to a sequence if itemsets purchased by the customer.

A subsequence of a sequence of itemsets is obtained by deleting one or more itemsets, and is also a sequence of itemsets. The support for a sequence S of itemsets is the percentage of customer sequences of which S is a subsequence. The problem of identifying sequential patterns is to find all sequences that have a user specified minimum support. A sequence <a1,a2,…am> with minimum support tells us that customers often purchase the items in set a1 in a transaction, then in some sub sequent transaction buy the items in set a2, then the items in set a3 in the later transaction, and so on.

Bayesian Networks

Finding causal relationship is a challenging task, as we saw in the previous section. In general, if certain events are highly correlated, there are many possible explanations. Fro example, suppose that pens, pencils, and ink are purchased together frequently. It might be that the purchase of one of these items depends casually on the purchase of another item. Or it might be that the purchase of one of these items is strongly correlated with the purchase of another because of some underlying phenomenon.

One approach is to consider each possible combination of causal relationship among the variables or events of interest to us and evaluate the likelihood of each combination on the basis of the data available to us. If we think of each combination of casual relationships as a model of the real world underlying the collected data, we can assign score to each model by considering how consistent it is with the observed data. Bayesian network is a

graphs that can be used to describe a class of such models, with one node per variable or event, and arcs between nodes to indicate causality.

Classification and Regression Rules

Consider the following view that contains information from a mailing compaign performed by an insurance company:

insuranceInfo(age: integer, cartype: string, highrisk: Boolean)

the insuranceInfo view has information about current customers. Each record contains a customer's age and type of car as well as flag indicating whether the person is considered a high-risk customer. If the flag is true, the customer is considered high-risk. We would like to use this information to identify rules that predict the insurance risk of new insurance applicants whose age an car type are known.

- If the dependent attribute is categorical, we call such rules classification rules.
- If the dependent attribute is numerical, we call such rules regression rules.

We can define support and confidence for classification and regression rules, as for association rules:

- **Support:** The support for a condition C is the percentage of tuples that satisfy C. the support for a rule C1➔C2 is the support for the condition C1 n C2.
- **Condition:** Consider those tuples that satisfy condition C1. the confidence for a rule C1➔C2 is the percentage of such tuples that also satisfy the condition C2.

11.4 Tree structured Rules

In this section, we discuss the problem of discovering classification and regression rules from a relation, but we consider only rules that have a very special structure. The type of rules we discuss can be represented by a tree, and typically the tree itself is the output of the data mining activity. Trees that represent classification rules are called classification trees or decision trees and trees that represent regression rules are called regression trees.

An example, consider the decision tree shown in the above figure. Each path from the root node to a leaf node represents one classification rule. For example, the path from the root to the leftmost leaf node represents the classification rule: "if a person is 25 years or younger and drives a sedan, then he or she is likely to have a low insurance risk". The path from the root to the right most leaf node represents the classification rule: "if a person is older than 25 years, then he or she is likely to have a low insurance risk".

Tree-structured rules are very popular since they are easy to interpret. Ease of understanding is very important because the result of any data mining activity need s to be comprehensible by non specialists. There exists efficient algorithms to construct tree-structured rules from large databases.

Decision Trees

A decision tree is a graphical representation of a collection of classification rules. Given a data record, the tree directs the record from the root to a leaf. Each internal node of the tree is labeled with the predictor attribute. This attribute is often called a splitting attribute, because the data is 'split' based on conditions over this attribute. The outgoing edges of an internal node are labeled with predicts that involve the splitting attribute of the node; every data record entering the node must satisfy the predicate labeling exactly one outgoing edge. The

Combined information about the splitting attribute and the predicates on the outgoing edges is called the splitting criterion of the node. A node with no outgoing edges is called a leaf node. Each leaf node of the tree is labeled with the value of the dependent attribute. W consider only binary trees where internal nodes have two outgoing edges, although trees of higher degree are possible.

A decision tree is usually constructed in two phases. In phase one, the growth phase, an overlay large tree is constructed. This tree represents the records in the input database very accurately; for example, the tree might contain leaf nodes for individual records from the input database. In phase two, the purning phase, the final size of the tree is determined. The rules represented by the tree constructed in phase one are usually overspecialized. By reducing the size of the tree, we generate a smaller number of more general rules that are better than a very large number of very specialized rules. Algorithms for tree pruning are beyond our scope discussion here.

Classification tree algorithms build the tree greedily top-down in the following way. At the root node, the database is examined and the locally best splitting criterion is computed. The database is then partitioned, according to the root nodes splitting criterion, into two parts, one partition for the left child and one partition for the right child. The algorithm then recourses on each child. This is shown in the following figure.

Input: node n, Partition D, split selection method S

Output: decision tree for D rooted at nod n

Top-Down Decision Tree Induction Schema:

BuildTree(Noden, data partition D, split selection method S)

Apply S to D to find the splitting criterion

If (a good splitting criterion is found)

Create two children nodes n1 and n2 of n

Partition D into D1 and D2

BuidTree(n1, D1,S)

BuildTree(n2, D2, S)

Endif

Fig. Decision Tree Induction Schema

11.5 Clustering

In this section we discuss the clustering algorithm. The goal is to partition a set of records into groups such that records within a group are similar to each other and records that belong to two different groups are dissimilar. Each such group is called a cluster and each record belongs to exactly one cluster. Similarly between records is measured computationally by a distance function. A distance function takes two input records and returns a vale that is a measure if their similarity. Different application have different notions of similarly, and no one measure works for all domains.

As an example, consider the schema of the customerInfo view:

CustomerInfo(age: int, salary: real)

There are two types of clustering algorithms. A partitional clustering algorithm partitions the data into k groups such that some criterion that evaluates the clustering quality is optimized. The number of clusters k is a parameter whose value is specified by the user. A hierarchical clustering algorithm generates a sequence of partitions of the records. Starting with a partition in which each cluster consists of one single record, the algorithm merges two partitions in each step only one single partition remains in the end.

A Clustering Algorithm

Clustering is a very old problem, and numerous algorithms have been developed to cluster a collection of records. Traditionally, the number of records in the input database was assumed to be relatively small and the complete database was assumed to fit into main memory. In this section, we describe a clustering algorithm called BIRCH that handles very large databases. The design of BIRCH reflects the following two assumptions:

- The number of records is potentially very large, and therefore we want to make only one scan over the database.
- Only a limited amount of main memory is available.

A user can set two parameters to control the BIRCH algorithm. The first is a threshold on the amount of main memory available. This main memory threshold translates into a maximum number of aluster summaries k that can be maintained in memory. The second parameter e is an initial threshold for the radius of any cluster. The value of e is an upper bound on the radius of any cluster and controls the number of clusters of clusters that the algorithm discovers.

The algorithm reads records from the database sequentially and processes them as follows:

Compute the distance between record r and each of the existing cluster centers. Let I be the cluster index such that the distance between r and Ci is the smallest.

Compute the value of the new radius Ti of the ith cluster under the assumption that r is inserted into it. If Ri < e , the the ith cluster remains compact, and we assign r to the ith cluster by updating its center and setting its

radius to Ri. If Ri > e then the ith cluster would no longer be compact if we insert r into it. Therefore we start a new cluster containing only the record r.

11.6 Similarity Search Over Sequences

A lot of information stored in databases consists of sequences. In this section, we introduce the problem of similarity search over a collection of sequences. Our query model is very simple: we assume that the user specifies a query sequence and wants to retrieve all data sequences that are similar to the query sequence. similarly search is different from normal queries in that we are interested not only in sequences that match the query sequence exactly but also those that differ only slightly from the query sequence.

We begin by describing sequences and similarity between sequences. A data sequence X is a series of numbers X=(x1,x2,…..xk). some times X is also called time series. We call k the length of the sequence. A sub sequence z=(z1,z2,…. ,zj) is obtained from another sequence X=(x1,x2,…..xk) by deleting numbers from the front and back of the sequences.

Similarity queries over sequences can be classified into two types.

- **Complete sequence Matching:** The query sequence and the sequence s in the database have the same length. Given a user specified threshold parameter e, our goal is to retrieve all sequences in the database that are within e-distance to the query sequence.
- **Subsequence Matching:** The query sequence is shorter than the sequences in the database. In this case, we want to find all sub sequences of sequences in the database such that the sub sequence is within distance e of the query sequence. we do not discuss sub sequence matching.

11.7 Incremental Mining and Data streams

Real life data is not static, but is constantly evolving through addition or deletions of records. In some applications, such as network monitoring, data arrives in such high speed streams that it is infeasible to store the data for offline analysis. We describe both evolving and streaming data in terms of a framework called block evolution. In block evolution, the input dataset to the data mining process is not static but periodically updated with the new block of tuples.

The goal of change detection is to quantify the difference, in terms of their data characteristics, between two sets of data and determine whether the change is meaningful. In particular, we must quantify the difference between the models of the data as it existed at some timet1 and the evolved version at a subsequent time t2; that is we must quantify the difference between M(D[1,t1]) and M(D[1,t2]).

Incremental model maintenance has received much attention. Since the quality of the data mining model is of utmost importance, incremental model maintenance algorithms have concentrated on computing exactly the same model as computed by running the basic model construction algorithm on the

union of old and new data. One widely used scalability technique is localization n of changes due to new blocks.

When working with high-speed data streams, algorithms must be designed to construct data mining models while looking at the relevant data items only once and in a fixed order, with a limited amount of main memory. Data stream computation has given rise to several recent studies of online or one pass algorithms with bounded memory. Algorithms have been developed for one-pass computation of qualities and order statistics, estimation of frequency moments and join sizes, clustering and decision tree construction, estimating correlated aggregates and computing one-dimensional histograms and haar wavelet decompositions.

11.8 Additional Data Mining Tasks

We focused on the problem of discovering patterns from a database, but there are several other equally important data mining tasks. We now discuss some of these briefly.

Dataset and Feature Selection: It is often important to select the right dataset to mine. Dataset selection is the process of finding which data sets to mine. Feature selection is the process of deciding which attributes to include in the mining process.

Sampling : One way to explore a large dataset is to obtain one or more samples and analyze them. The advantage of sampling is that we can carry out detailed analysis on a sample that would be infeasible on the entire dataset, for very large datasets. The disadvantage of sampling is that obtaining a representative sample for a given task is difficult; we might miss important tends or patterns because they are not reflected in the sample.

Visualization: Visualization techniques can significantly assist in understanding complex datasets and detecting interesting patterns, and the importance of visualization in data mining is widely recognized.

Self-Assesment Questions – XI

1. Data mining will ----------.
2. Data Cleaning----------.
3. Expansion of KDD is----------.
4. Clustering is----------.
5. Data mining will maintain a large database
    a) True
    b) False
    b) None
6. Expansion of PMML
    a) Product Model Markup Language
    b) Process Model Markup Language
    c) Predictive Model Markup Language

Sample questions

7. Write notes about Data mining.
8. Write down the association rules.
9. Explain different mining rules?
10. Explain Tree structured rules?
11. Explain classification regression rule?
12. Explain Clustering?
13. write down data mining additional tasks.

Answers for Self-Assesment Questions – XI

1. Extracting hidden knowledge from large dataset
2. Will enhance the data
3. Knowledge Discovery and Data mining
4. to grouping records
5. b – False
6. c – Predictive Model Markup Language

NOTES