



DATABASE SYSTEM CONCEPTS

Last updated: 4/7/2023

The information provided by Sam, Database System Concept ("we", "us", or "our") is for general information purpose only. All information in this template is provided in good faith, however we make no representation or warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of any information in the template. UNDER NO CIRCUMSTANCE SHALL WE HAVE ANY LIABILITY TO YOU FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS RESULT OF THE USE OF THE TEMPLATE OR RELIANCE ON ANY INFORMATION PROVIDED IN THE TEMPLATE. YOUR USE OF THE TEMPLATE AND YOUR RELIANCE ON THE DOCUMENT IS SOLELY AT YOUR OWN RISK.

This page intentionally left blank.

Contents

1	DATABASE SYSTEM CONCEPTS.....	4
2	RELATIONAL DATABASES	17
3	DATABASE DESIGN.....	127
4	DATA STORAGE AND QUERYING	200
5	TRANSACTION MANAGEMENT	288
6	SYSTEM ARCHITECTURE.....	352
7	DATA WAREHOUSING, DATA MINING, AND INFORMATION RETRIEVAL.....	401
8	SPECIALTY DATABASES.....	428
9	ADVANCED TOPICS.....	474
10	CASE STUDIES	509

1 DATABASE SYSTEM CONCEPTS

The landscape of computer science education has undergone a profound transformation in recent years, as database management has emerged as a critical pillar in modern computing environments. To equip students with the essential knowledge about database systems, we present a comprehensive exploration of the fundamental concepts of database management, ranging from database design to database languages to database-system implementation.

Intended for students at the junior or senior undergraduate or first-year graduate level, this text provides both basic materials for an introductory course as well as advanced material suitable for supplemental or advanced study. Assuming only a familiarity with basic data structures, computer organization, and high-level programming languages like *Java*, *C*, or *Pascal*, we present concepts as intuitive descriptions, grounded in the running example of a university.

While important theoretical results are covered, we eschew formal proofs in favor of figures and examples that vividly illustrate the truth of a given result. Formal descriptions and proofs can be found in advanced texts and research papers referenced in the bibliographical notes.

This repository is an indispensable resource for anyone seeking to acquire a deep understanding of database management, rooted in theoretical foundations but connected to practical applications.

Introduction

Database-System Applications

Database management systems (DBMS) is a system that stores and manages data, with the goal of providing a way to store and retrieve database information that is both convenient and efficient. Databases are used in many applications such as enterprise information, banking and finance, universities, airlines, and telecommunications. In addition to these applications, databases also play an important role in everyday life, such as when accessing an *online bookstore, bank website, or viewing advertisements online*. Database systems are designed to manage large bodies of information, ensuring the safety of information stored, despite system crashes or attempts at unauthorized access. The purpose of database systems is to provide a more efficient way to manage commercial data compared to earlier methods such as storing it in operating system files.

Purpose of Database Systems

The disadvantages of keeping organizational information in a file-processing system, highlighting the issues of **data redundancy**, difficulty in **accessing data**, **data isolation**, **integrity problems**, **atomicity problems**, and **concurrent-access anomalies**, these problems can lead to higher storage and access costs, data inconsistency, and data loss, and can make it difficult to maintain data consistency, supervise data access, and provide responsive data-retrieval systems. As a result, more responsive and efficient data-retrieval systems are required.

View of Data

The development of database systems to solve problems with file-processing systems is a collection of interrelated data and programs that allow users to access and modify the data. The system provides users with an abstract view of the data by using multiple levels of abstraction, including the physical level, logical level, and view level. The physical level describes how the data are stored, while the logical level describes what data are stored and what relationships exist among those data. The view level describes only part of the database to simplify users' interactions with the system. Many users of the database system do not need all the information stored in the database, and the view level of abstraction exists to simplify their interaction with the system.

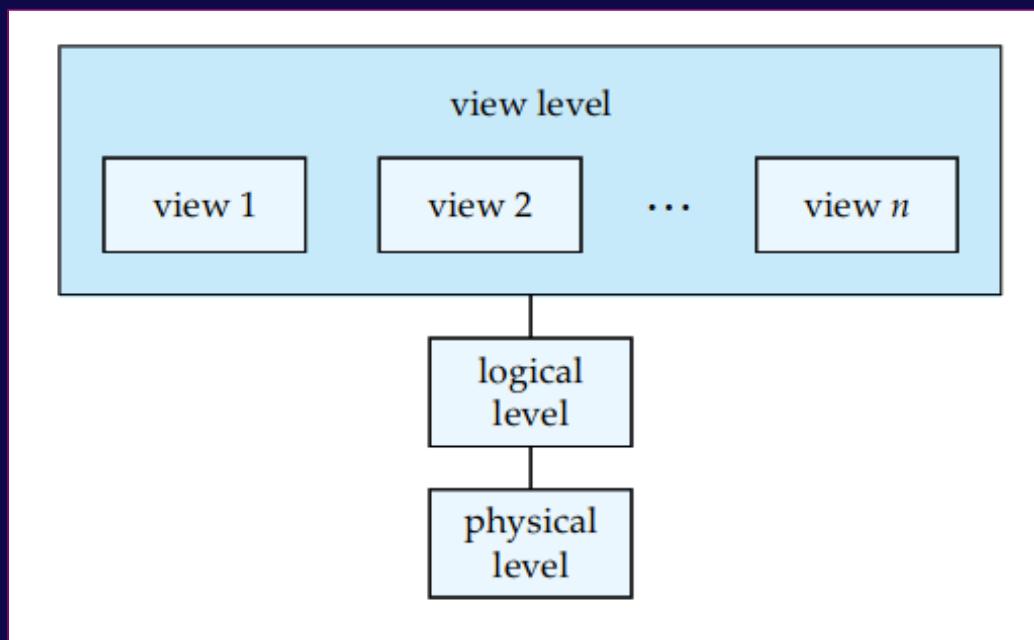


Figure 1 - The three levels of data abstraction

The code defines a new record type called "*instructor*" with four fields, each having a name and type. A university may have other record types, such as "*department*," "*course*," and "*student*," each with their own fields. At the physical level, each record is stored as a block of consecutive storage locations, but this detail is hidden from programmers and database users. **Database administrators** may be aware of these physical details.

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

end;
```

At the logical level, records are described by type definitions and their interrelationships are defined. **Programmers** and **database administrators work** at this level. At the view level, computer users see application programs that hide details of the data types. Views of the database are defined to provide a security mechanism to prevent users from accessing certain parts of the database.

The overall design of a database is called the database schema, while the information stored in the database at a particular moment is called an instance of the database. There are different types of schemas, such as the **physical schema**, **logical schema**, and **view level schema**. The logical schema is the most important as programmers use it to construct applications. The different data models used in databases are:

- Relational model
- Entity-relationship model
- Object-based data model
- Semistructured data model

The relational model is the most widely used. The network data model and the hierarchical data model are used little now and are outlined in appendices for interested readers.

Database Languages

The two main languages used in a database system are the **data-manipulation language (DML)** and the **data-definition language (DDL)**:

- The DML allows users to access, insert, delete and modify data stored in the database. It can be either procedural or declarative, with the latter being easier to use but requiring the system to figure out how to access data. A query language is a part of DML that involves information retrieval. SQL is the most widely used query language.
- The DDL specifies the database schema and additional properties of the data. It is also used to define the storage structure and access methods. The DDL includes constraints such as domain constraints, referential integrity, assertions, and authorization. These constraints ensure the consistency of data and restrict the type of access users have on various data values in the database.

Relational Databases

Relational databases are using tables to represent data and relationships among that data. Each table has multiple columns with unique names, and record-based models organize data in fixed-format records of several types. SQL is the most common language used in commercial relational database systems. The relational model can have schema design problems, such as unnecessarily duplicated information.

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

Figure 2 - The instructor table

The **Data-Manipulation Language (DML)** used in relational databases above, specifically the nonprocedural SQL query language. A query takes input from one or more tables and always returns a single table. An example SQL query is provided to illustrate how to retrieve data from a table.

```
select instructor.name  
      from instructor  
     where instructor.dept_name = 'History';
```

The expected result of running a specific SQL query on the tables in the Figure:

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

Figure 3 - The department table

The query is designed to find the names of all instructors in the History department. The system would search the tables and find that there are two departments with a budget greater than \$95,000: *Computer Science* and *Finance*. There are a total of five instructors in these departments, so the resulting table would have two columns (*ID*, *dept name*) and five rows listing the instructors in the two departments.

```
Create table department  
      (dept_name char (20), building char (15), budget numeric (12,2));
```

The **Data-Definition Language (DDL)** in SQL allows for the definition of tables and other database elements. The schema of a table is an example of metadata. SQL is not as powerful as a universal Turing machine and cannot support actions like input from users or output to displays, so application programs must be written in a host language like C, C++, or Java.

DML statements can be executed from the host language either by providing an application program interface or by embedding *DML calls* within the host language program using a preprocessor like the *DML precompiler*.

Database Design and Data Storage and Querying

Database design involves managing large amounts of information that are part of an enterprise's operation.

The initial step in designing a database is to characterize fully the data needs of prospective database users. Next, a data model is chosen and the requirements are translated into a conceptual schema.

The designer reviews the schema to confirm that all data requirements are satisfied and are not in conflict with one another. The final design phases involve mapping the conceptual schema onto the implementation data model of the database system that will be used and specifying the physical features of the database.

An example of how a database for a university could be designed. The entity-relationship data model is discussed as a way to represent entities and relationships in a database.

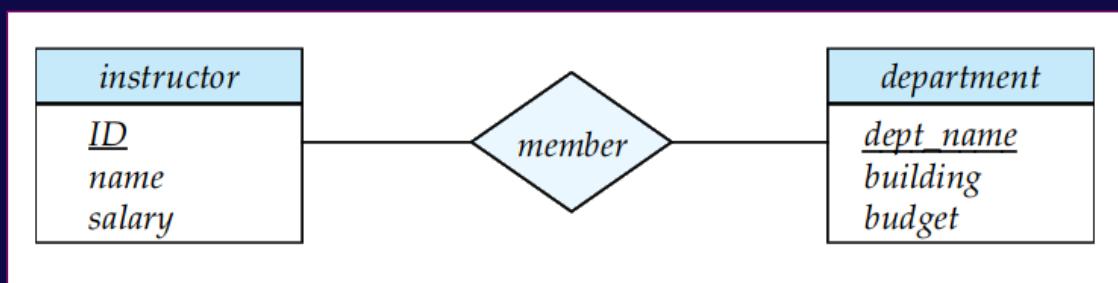


Figure 4 - A sample E-R diagram

Two methods for designing a relational database:

- Entity-relationship modeling
- Normalization

Entity-relationship modeling involves representing entity sets and relationship sets using UML, while normalization aims to generate relation schemas that avoid redundancy and allow easy information retrieval.

Also, how-to map correctly cardinalities in the E-R model and discusses the drawbacks of a poorly designed database, such as repetition of information and an inability to represent certain information.

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept.name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 5 - The faculty table

Transaction Management

Two problems arise when designing databases, namely, the **repetition of information** and the **inability to represent certain information**.

- These problems can be resolved through normalization, which is a formal method of designing databases.
- The storage manager is responsible for storing, retrieving, and updating data in the database, and its components include the authorization and integrity manager, transaction manager, file manager, and buffer manager.
- The query processor, on the other hand, simplifies access to data for database users.

Database Architecture

The three main components:

- Query Processor
- Transaction Management
- Database Architecture

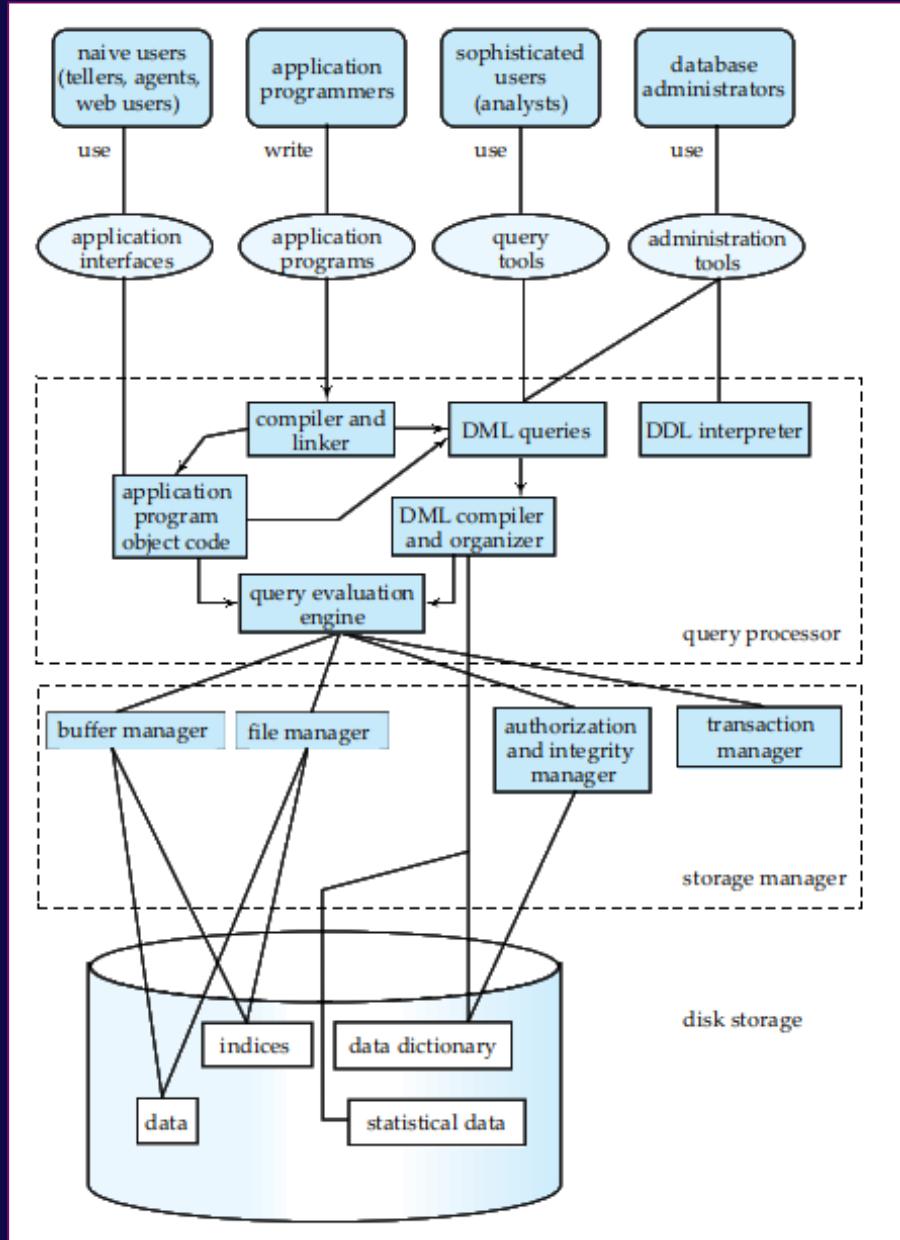


Figure 6 - System structure

The Query Processor consists of three parts:

- DDL Interpreter
- DML Compiler
- Query Evaluation Engine

The DDL Interpreter interprets DDL statements and records the definitions in the data dictionary. The **DML Compiler** translates **DML statements** in a query language into an evaluation plan consisting of low-level instructions.

The **Query Evaluation Engine** executes low-level instructions generated by the **DML Compiler**.

Transaction Management deals with the processing of a single logical function in a database application called a transaction, which must satisfy atomicity, consistency, and durability properties. The **Recovery Manager** ensures the atomicity and durability properties of a transaction, while the **Concurrency Control Manager** controls the interaction among concurrent transactions to ensure the consistency of the database.

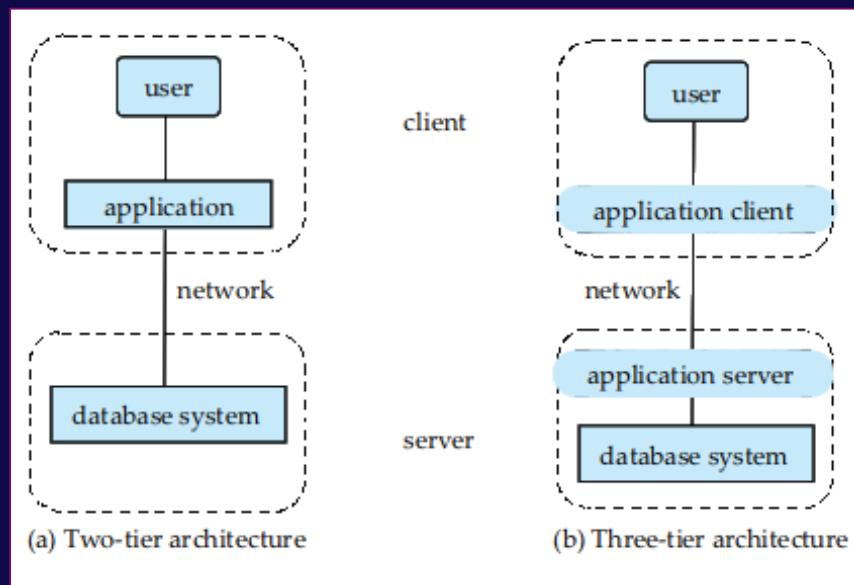


Figure 7 - Two-tier and three-tier architectures

The Database Architecture is greatly influenced by the underlying computer system on which the database system runs, and it includes **centralized**, **client-server**, **parallel**, and **distributed databases**.

Data Mining and Information Retrieval and Specialty Databases

Data mining is the process of analyzing large databases to find useful patterns, and information retrieval, which is the querying of unstructured textual data. Specialty databases, such as **object-based data models** and **semistructured data models**. Object-oriented programming concepts, such as encapsulation and inheritance, have been applied to data modeling. The object-relational data model is a combination of object-oriented and relational models. Semistructured data models allow for data with varying sets of attributes and are used in XML language.

Database Users and Administrators

The different types of users who work with a database system, including users, application programmers, sophisticated users, and specialized users. Each type of user interacts with the system differently, using various user interfaces and tools. The role of a **database administrator (DBA)** is the central control over the system, including creating and modifying the database schema, granting authorization for data access, and performing routine maintenance tasks like backing up the database and monitoring performance.

History of Database Systems

The use of punched cards and mechanical systems were the precursors to automation of data processing tasks. In the late 1960s and 1970s, the use of hard disks led to the creation of network and hierarchical databases, and the introduction of the relational model by Codd. Although the relational model was not initially used due to perceived performance disadvantages, it became dominant in the 1980s. The 1990s saw the explosive growth of the World Wide Web, which necessitated database systems that supported high transaction-processing rates, reliability, and 24x7 availability. The 2000s saw the growth of XML and associated query language XQuery, as well as autonomic-computing/auto-admin techniques, and the use of open-source database systems like PostgreSQL and MySQL. The latter part of the decade saw the growth of specialized databases for data.

2 RELATIONAL DATABASES

The importance of data models, with a focus on the relational model. The relational model uses tables to represent data and relationships, and is widely adopted in database products. To make data available to users, query languages like SQL have been developed, and data integrity and protection are also important issues, including integrity constraints and authorization mechanisms. The chapter covers formal query languages based on mathematical logic, which form the basis for SQL and other user-friendly languages.

2.1 Introduction to the Relational Model and Structure of Relational Databases

The relational model is the primary data model for commercial data-processing applications because of its simplicity. It is made up of a collection of tables, where each table is assigned a unique name and consists of rows and columns.

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

Figure 8 - The instructor relation

A row in a table represents a relationship among a set of values, and each table is a collection of relationships.

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

Figure 9 - The course relation

In the relational model, a relation is used to refer to a table, while the term tuple is used to refer to a row, and the term attribute refers to a column of a table.

course.id	prereq.id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 10 - The prereq relation

Each attribute of a relation has a set of permitted values, called the domain of that attribute. The null value is a special value that signifies that the value is unknown.

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

Figure 11 - Unsorted display of the instructor relation

Database Schema

The concept of database schema refers to the logical design of a database, and the database instance, which is a snapshot of the data in the database at a given instant in time. A relation schema consists of a list of attributes and their corresponding domains, while a relation instance corresponds to the value of a variable.

The schema of a relation generally does not change, whereas the contents of a relation instance may change as the relation is updated.

```
department (dept name, building, budget)  
section (course id, sec id, semester, year, building, room number, time  
slot id)
```

<i>dept.name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 12 - The department relation

Examples of different relation schemas and instances in a university database.

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

course.id	sec.id	semester	year	building	room.number	time.slot.id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 13 - The section relation

Common attributes in relation schemas are used to relate tuples of distinct relations.

Keys

The concept of keys in database design is important for identifying tuples within a relation. A superkey is a set of one or more attributes that can uniquely identify a tuple in the relation, while a candidate key is a minimal superkey for which no proper subset is a superkey.

A **primary key** is a candidate key chosen by the database designer as the principal means of identifying tuples within a relation. Primary keys should be chosen carefully, as they represent a constraint in the real-world enterprise being modeled, and should be unlikely to change.

Foreign keys are attributes in a relation that reference the primary key of another relation. Referential integrity constraints require that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Schema Diagrams

Schema diagrams are used to depict the structure of a database schema, including primary key and foreign key dependencies. Each relation is represented as a box with the relation name and attributes listed inside, and foreign key dependencies are shown as arrows.

Other referential integrity constraints are not shown explicitly, but can be represented using **entity-relationship diagrams**. Many database systems provide graphical tools for creating schema diagrams.

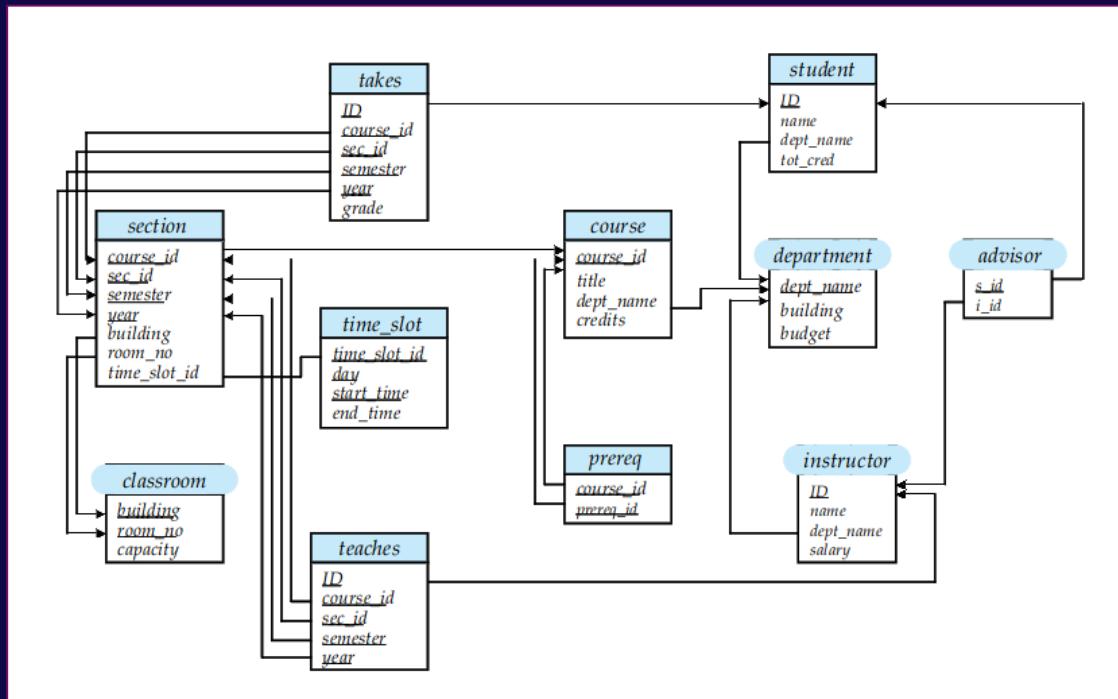


Figure 14 - Schema diagram for the university database

The example organization used in later chapters is a university, and its corresponding relational schema.

Relational Query Languages

Query languages used to retrieve information from a database can be procedural or nonprocedural. SQL is a widely used query language that incorporates elements of both approaches. Also, "pure" query languages are related to the relational algebra, tuple relational calculus, and domain relational calculus.

These languages are formal and lack the "syntactic sugar" of commercial languages but illustrate fundamental techniques for extracting data from a database. The relational algebra consists of a set of operations, while the relational calculus uses predicate logic to define desired results without giving specific procedures.

```
classroom(building, room.number, capacity)
department(dept.name, building, budget)
course(course.id, title, dept.name, credits)
instructor(ID, name, dept.name, salary)
section(course.id, sec.id, semester, year, building, room.number, time.slot.id)
teaches(ID, course.id, sec.id, semester, year)
student(ID, name, dept.name, tot.cred)
takes(ID, course.id, sec.id, semester, year, grade)
advisor(s.ID, i.ID)
time.slot(time.slot.id, day, start.time, end.time)
prereq(course.id, prereq.id)
```

Figure 15 - Schema of the university database

Relational Operations

Relational operations are a set of operations that can be applied to either a single relation or a pair of relations. The most frequent operation is selecting specific tuples from a single relation that satisfies a particular predicate.

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
83821	Brandt	Comp. Sci.	92000

Figure 16 - Result of query selecting instructor tuples with salary greater than \$85000

Another operation is selecting certain attributes from a relation. The join operation combines two relations by merging pairs of tuples, and the natural join operation matches tuples whose values are the same on all attribute names that are common to both relations.

<i>ID</i>	<i>salary</i>
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

Figure 17 - Result of query selecting attributes *ID* and *salary* from the *instructor* relation

The **Cartesian product** operation combines tuples from two relations, regardless of whether their attribute values match.

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept.name</i>	<i>building</i>	<i>budget</i>
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Figure 18 - Result of natural join of the instructor and department relations

Normal set operations like union, intersection, and set difference can be performed on relations. Operations can be performed on the results of queries.

<i>ID</i>	<i>salary</i>
12121	90000
22222	95000
33456	87000
83821	92000

Figure 19 - Result of selecting attributes *ID* and *salary* of instructors with salary greater than \$85,000

2.2 Introduction to SQL

The chapter covers SQL's fundamental concepts and constructs and acknowledges that individual SQL implementations may differ in details or support only a subset of the full language.

Overview of the SQL Query Language

The SQL language was originally developed by IBM in the *1970s* and has evolved into the standard relational database language. ANSI and ISO have published several versions of the SQL standard, including *SQL-86*, *SQL-89*, *SQL-92*, *SQL:1999*, *SQL:2003*, *SQL:2006*, and *SQL:2008*. The SQL language consists of various parts, including DDL, DML, integrity, view definition, transaction control, embedded SQL and dynamic SQL, and authorization.

Although most SQL implementations support standard features, differences between implementations exist, and users should consult their database system's user manuals to determine which features are supported.

SQL Data Definition

We define an SQL relation by using the create table command. The following command creates a relation department in the database.

```
create table department  
  (dept name varchar (20), building varchar (15), budget numeric  
  (12,2), primary key (dept name));
```

An example of creating a relation in SQL with three attributes - dept name, building, and budget.

```
create table r  
  (A1 D1, A2 D2, ..., An Dn,  
  (integrity-constraint1), ..., (integrity-constraintk));
```

The data types for these attributes are specified as character string of maximum length 20, character string of maximum length 15, and number with 12 digits in total, 2 of which are after the decimal point, respectively. The primary key for this relation is set to be the dept name attribute.

- SQL commands and constraints, the syntax of creating tables and defines various constraints such as primary key, foreign key, and not null.
- SQL prevents any updates to the database that violate an integrity constraint.
- For exampleL insert command used to load data into a relation.
- Also, refers to a partial SQL DDL definition of the university database.

We can use the delete command to delete tuples from a relation:

```
delete from student;

create table department
(dept.name      varchar (20),
 building       varchar (15),
 budget         numeric (12,2),
 primary key (dept.name));

create table course
(course.id      varchar (7),
 title          varchar (50),
 dept.name     varchar (20),
 credits        numeric (2,0),
 primary key (course.id),
 foreign key (dept.name) references department);

create table instructor
(ID             varchar (5),
 name           varchar (20) not null,
 dept.name     varchar (20),
 salary         numeric (8,2),
 primary key (ID),
 foreign key (dept.name) references department);

create table section
(course.id      varchar (8),
 sec.id         varchar (8),
 semester       varchar (6),
 year           numeric (4,0),
 building       varchar (15),
 room.number   varchar (7),
 time.slot.id  varchar (4),
 primary key (course.id, sec.id, semester, year),
 foreign key (course.id) references course);

create table teaches
(ID             varchar (5),
 course.id     varchar (8),
 sec.id         varchar (8),
 semester       varchar (6),
 year           numeric (4,0),
 primary key (ID, course.id, sec.id, semester, year),
 foreign key (course.id, sec.id, semester, year) references section,
 foreign key (ID) references instructor);
```

Figure 20 - SQL data definition for part of the university database

The delete command in SQL can be used to remove specific tuples from a relation or all tuples from the relation. The drop table command is used to remove a relation from the SQL database, deleting all information about it:

```
drop table r;  
delete from r;
```

- SQL commands for modifying the structure of a relation, including deleting tuples or dropping a relation entirely using the DELETE and DROP TABLE commands, respectively.
- The ALTER TABLE command for adding attributes to an existing relation, with all tuples assigned null as the value for the new attribute.

```
alter table r add AD;
```

The alter table command allows us to add a new attribute to an existing relation named **r**, where **A** is the name of the attribute to be added and **D** is the type of the new attribute. In contrast, we can remove attributes from a relation using the drop command.

```
alter table r drop A;
```

In SQL, an attribute can be dropped from an existing relation by using the *alter table* command. However, many database systems do not support this functionality, and instead only allow an entire table to be dropped.

Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses: select, from, and where. The query operates on the relations listed in the from clause, performs operations on them as specified in the where and select clauses, and produces a relation as the result.

<u>name</u>
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 21 - Result of "select name from instructor"

In the case of a query on a single relation, the example given is to find the names of all instructors in the university example. The instructor relation is listed in the from clause and the name attribute is specified in the select clause.

```
select name from instructor;
```

The basic structure of an SQL query consists of three clauses: select, from, and where. The select clause specifies the attributes of the relation to be displayed in the output, the from clause specifies the input relation, and the where clause is used for filtering the tuples.

```
select dept name from instructor;
```

SQL allows duplicate tuples in relations and in the results of SQL expressions. In order to force the elimination of duplicates, the keyword "*distinct*" is inserted after select.

```
select distinct dept name from instructor;
```

dept.name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 22 - Result of "select dept name from instructor"

The SQL keyword "*distinct*" can be used to eliminate duplicates in query results, while "*all*" can be used to specify that duplicates should be retained. The select clause in SQL queries can include arithmetic expressions that involve mathematical operators and constants or attributes of tuples.

```
select all dept name from instructor;
```

The where clause allows the selection of specific rows in the query result based on a specified predicate.

```
select ID, name, dept name, salary *1.1 from instructor;
```

The SQL language allows us to manipulate data in a relational database. We can use arithmetic expressions to modify the values of attributes or create new attributes, but this does not modify the original relation.

```
select name from instructor  
where dept.name = 'Comp. Sci.' and salary > 70000;
```

name
Katz
Brandt

Figure 23 - Result of "Find the names of all instructors in the Computer Science department who have salary greater than \$70,000."

SQL also provides special data types and arithmetic functions to operate on them. We can use the **WHERE** clause to filter the results of a query and retrieve only those tuples that satisfy a specified predicate.

name	dept.name	building
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 24 - The result of "Retrieve the names of all instructors, along with their department names and department building name."

Logical connectives such as **AND**, **OR**, and **NOT** can be used in the **WHERE** clause along with comparison operators to compare strings and arithmetic expressions. In queries involving multiple relations, we need to specify the relations to be accessed in the **FROM** clause and specify the matching condition in the **WHERE** clause. The **SELECT** clause is used to retrieve the desired attributes from the selected tuples.

This section discusses SQL queries that involve multiple relations. It explains that in such cases, the naming convention requires that relations in the from clause have distinct names. The select clause is used to list the desired attributes in the query result, the from clause is a list of the relations to be accessed, and the where clause is a predicate involving attributes of the relations in the from clause. The general form of an SQL query is given, which includes attributes, relations, and a predicate. If the where clause is omitted, the predicate is true.

```
select A1, A2, ..., An from r1, r2, ..., rm where P;
```

The order of the clauses in an SQL query must be select, from, and then where, but it can be easier to understand the operations if they are considered in the order from, where, and then select. The from clause generates tuples for the result relation of the from clause by defining a **Cartesian product** of the relations listed in the clause.

```
for each tuple t1 in relation r1  
for each tuple t2 in relation r2      ...  
for each tuple tm in relation rm  
  
Concatenate t1, t2, ..., tm into a single tuple t  
  
Add t into the result relation
```

This is done iteratively, creating a new tuple by concatenating each tuple in each relation, and then adding that tuple to the result relation. The result relation has all attributes from all the relations in the from clause, and since the same attribute name may appear in multiple relations, the name of the relation from which the attribute originally came must be prefixed before the attribute name.

The naming convention used in SQL queries involving multiple relations:

```
(instructor.ID, instructor.name, instructor.dept name,  
instructor.salary teaches.ID, teaches.course id, teaches.sec id,  
teaches.semester, teaches.year)
```

Attributes that appear in only one relation do not need to be pre-fixed with the relation name. However, when the same attribute name appears in multiple relations, the naming convention requires that the attribute be prefixed with the relation name to avoid ambiguity.

```
(instructor.ID, name, dept.name, salary teaches.ID, course.id, sec.id,
semester, year)
```

The **Cartesian product** of relations is defined as a set of tuples generated by iterating through each tuple in each relation listed in the from clause. The resulting relation has all the attributes from all the relations in the from clause. The article illustrates the **Cartesian product** of the instructor and teaches relations and explains that combining tuples from unrelated relations can result in an extremely large relation, making it rarely sensible to create such a **Cartesian product**.

<i>inst.ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course.id</i>	<i>sec.id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...

Figure 25 - The Cartesian product of the instructor relation with the teaches relation

To use the **WHERE** clause in an SQL query to restrict the output to meaningful results. The **WHERE** clause is used to combine tuples from different tables based on common attributes.

```
select name, course id from instructor, teaches  
where instructor.ID= teaches.ID;
```

SQL query combines tuples from the instructor and teaches tables to output instructor names and course IDs. The output only includes instructors who have taught some course, and instructors who have not taught any course are excluded.

```
select name, course id from instructor, teaches  
where instructor.ID= teaches.ID and instructor.dept name =  
'Comp. Sci.';
```

To add an extra predicate to the **WHERE** clause to further restrict the output:

<i>name</i>	<i>course.id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 26 - Result of "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

The sequence of steps involved in executing an SQL query, which involves generating a **Cartesian** product of the relations listed in the **FROM** clause, applying predicates specified in the WHERE clause, and outputting the selected attributes.

However, a real implementation of SQL optimizes the evaluation process to generate only the elements of the **Cartesian** product that satisfy the **WHERE** clause predicates. Also the consequences of omitting appropriate WHERE clause conditions can lead to the output of a huge relation.

The natural join operation in SQL produces a relation as a result by considering only those pairs of tuples with the same value on the common attributes that appear in the schemas of both relations. SQL also supports other ways of joining information from multiple relations.

The natural join operation in SQL combines information from two relations based on the matching of attributes with the same name.

```
select name, course id from instructor, teaches  
where instructor.ID= teaches.ID;
```

It produces a relation that contains only those tuples with the same value on the common attribute.

```
select name, course id from instructor natural join teaches;
```

The resulting relation has only those attributes that are common to the schemas of both relations, followed by those that are unique to the schema of each relation.

```
select A1, A2,..., An from r1 natural join r2 natural join ... natural join rm where P;
```

The natural join can be used in SQL queries with the from clause to combine multiple relations.

```
from E1, E2,..., En
```

The select and where clauses are then evaluated on the resulting relation.

```
select name, title from instructor natural join teaches, course  
where teaches.course id= course.course id;
```

The natural join is a useful tool for simplifying SQL queries and reducing the size of the output.

SQL operations, specifically natural join and rename operations.

```
select name, title from instructor natural join teaches natural join course;  
select name, title from (instructor natural join teaches) join course  
using (course id);
```

Additional Basic Operations

The natural join operation combines two tables by matching attributes with the same name, while the rename operation allows renaming of table attributes using the "as" clause.

```
select name, course id from instructor, teaches  
      where instructor.ID= teaches.ID  
            name, course id
```

Natural join while avoiding equating attributes erroneously and how the rename operation is useful in renaming relations.

```
name, course id  
old-name as new-name
```

SQL queries use these operations to obtain specific results.

```
select name as instructor name, course id from instructor, teaches  
      where instructor.ID= teaches.ID;  
  
select T.name, S.course id from instructor as T, teaches  
      as S  where T.ID= S.ID;
```

SQL queries and a few examples below of different operations that can be performed on character strings:

```
select distinct T.name from instructor as T, instructor as S  
      where T.salary > S.salary and S.dept name = 'Biology';
```

Also, the use of correlation names or table aliases in SQL. SQL strings are enclosed in single quotes and that the equality operation on strings is case sensitive.

```
select dept name from department  where building like '%Watson%';
```

SQL also permits a variety of functions on character strings, such as concatenation, substring extraction, and string length calculation. To perform pattern matching using the like operator and how to specify an escape character for special pattern characters.

```
like 'ab%cd%' escape '' matches all strings  
beginning with "ab%cd" like 'ab\cd%' escape ''  
matches all strings beginning with "ab"
```

Finally, some databases provide variants of the like operation which do not distinguish between lower and upper case, and SQL:1999 offers a similar to operation that provides more powerful pattern matching.

In SQL, the asterisk symbol (*) is used in the select clause to indicate "all attributes" of a table.

```
select instructor.* from instructor, teaches  
      where instructor.ID= teaches.ID;
```

The order by clause can be used to sort the tuples in the result of a query in ascending or descending order, and can be performed on multiple attributes.

```
select name from instructor  
      where dept name = 'Physics' order by name;
```

The between comparison operator simplifies where clauses that specify a value range, and the not between operator can also be used.

```
select * from instructor  
      order by salary desc, name asc;
```

To find instructor names and the courses they taught for instructors in the Biology department who have taught some course, an extra condition can be added to the where clause.

```
select name from instructor  
      where salary between 90000 and 100000;
```

instead of:

```
select name from instructor  
      where salary <= 100000 and salary >= 90000;
```

Below are various SQL operations such as attribute specification, ordering of tuples, where clause predicates, and set operations:

```
select name, course id from instructor, teaches  
      where instructor.ID= teaches.ID and dept name = 'Biology';  
  
select name, course id from instructor, teaches  
      where (instructor.ID, dept name) = (teaches.ID, 'Biology');  
  
select name, course id from instructor, teaches  
      where (instructor.ID, dept name) = (teaches.ID, 'Biology');
```

course.id
CS-101
CS-347
PHY-101

Figure 27 - The c1 relation, listing courses taught in Fall 2009

The asterisk symbol (*) is used in the select clause to indicate "all attributes."

```
select name, course id from instructor, teaches  
      where (instructor.ID, dept name) = (teaches.ID, 'Biology');
```

Set Operations

Ordering of tuples can be performed using the order by clause, and we can specify the sort order using desc for descending order or asc for ascending order.

```
select course id from section  
    where semester = 'Fall' and year= 2009;  
  
select course id from section  
    where semester = 'Spring' and year= 2010;
```

The between comparison operator can be used to simplify where clauses that specify a value between two values. The SQL operations union, intersect, and except are used for set operations.

```
select course id from section  
    where semester = 'Fall' and year= 2009  
        union (select course id from section where semester =  
    'Spring' and year= 2010);
```

The union operation eliminates duplicates automatically.

The three set operations in SQL are:

- Union
- Intersect
- Except

```
(select course id from section where semester = 'Fall' and year= 2009)

union all (select course id from section

where semester = 'Spring' and year= 2010);

(select course id from section

where semester = 'Fall' and year= 2009)

intersect (select course id from section

where semester = 'Spring' and year= 2010);
```

<u>course.id</u>
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
<u>PHY-101</u>

Figure 28 - The result relation for c1 union c2

The union operation combines the results of two queries, and it eliminates duplicates by default. However, if we want to retain duplicates, we must use union all instead of union.

course.id
CS-101

Figure 29 - The result relation for $c1 \text{ intersect } c2$

```
(select course id from section  
      where semester = 'Fall' and year= 2009)  
intersect all (select course id   from section  
      where semester = 'Spring' and year= 2010);
```

The intersect operation finds the common elements between two sets, and it automatically eliminates duplicates. But, if we want to keep duplicates, we must use intersect all instead of intersect. Finally, the except operation finds the difference between two sets, and it automatically eliminates duplicates.

```
(select course id from section  
      where semester = 'Fall' and year= 2009)  
except (select course id   from section  
      where semester = 'Spring' and year= 2010);
```

For all three operations, the number of duplicates in the result depends on the occurrence of duplicates in both input sets. The article provides examples to clarify the concepts.

The three set operations in SQL are:

- Union
- Intersect
- Except

```
(select course.id from section  
    where semester = 'Fall' and year= 2009)  
        union all (select course.id  from section  
            where semester = 'Spring' and year= 2010);  
  
(select course.id from section  
    where semester = 'Fall' and year= 2009)  
intersect (select course.id  from section  
    where semester = 'Spring' and year= 2010);
```

The union operation combines the results of two queries, and it eliminates duplicates by default. However, if we want to retain duplicates, we must use union all instead of union.

course.id
CS-347
PHY-101

Figure 30 - The result relation for c1 except c2

```
(SELECT course.id FROM section  
    WHERE semester = 'Fall' AND year = 2009)  
INTERSECT ALL (SELECT course.id  
    FROM section WHERE semester = 'Spring' AND year = 2010);
```

The intersect operation finds the common elements between two sets, and it automatically eliminates duplicates. But, if we want to keep duplicates, we must use intersect all instead of intersect.

Finally, the except operation finds the difference between two sets, and it automatically eliminates duplicates.

```
(select course id from section  
      where semester = 'Fall' and year= 2009)  
except (select course id   from section  
      where semester = 'Spring' and year= 2010);
```

For all three operations, the number of duplicates in the result depends on the occurrence of duplicates in both input sets. The article provides examples to clarify the concepts.

Null Values

In the realm of relational operations, the ubiquitous presence of null values poses unique challenges.

Arithmetical expressions involving operators such as +, -, *, and / can have their results thrown into disarray when one or more of their input values are null. Consider an expression like **r.A+5**; if the value of **r.A** happens to be null for a particular tuple, then the expression's result must also be null for that same tuple.

However, comparisons with nulls can be an even more formidable obstacle. For instance, if we take the comparison "**1 < null**," it would be fallacious to assert that it is true since the null value's meaning is unknown.

At the same time, it would be erroneous to claim that the expression is false, as that would result in "**not (1 < null)**" evaluating to true, which defies all sense. To navigate this conundrum, SQL designates the outcome of any comparison involving a null value as unknown, thus introducing a third logical value besides true and false.

```
select name from instructor where salary is null;
```

When using the select distinct clause, identical tuples must be removed. To accomplish this, SQL treats values of corresponding attributes from two tuples as the same if they are both non-null and equal in value or both null.

Therefore, two copies of a tuple such **as {('A', null), ('A', null)}** are considered identical, even if some of the attributes have null values. By using the distinct clause, only one copy of these identical tuples can be retained.

It is important to note that nulls are treated differently in predicates, where a comparison "**null=null**" yields unknown instead of true. This approach of treating tuples as identical is also employed in the set operations union, intersection, and except.

In short, dealing with null values in relational operations requires careful handling and an understanding of how SQL operates with unknown values.

Aggregate Functions

SQL provides a powerful set of tools for querying and manipulating data stored in relational databases.

One of the most important features of SQL is its support for aggregate functions, which allow users to perform computations on collections of values and return a single result.

The five built-in aggregate functions in SQL are average, minimum, maximum, total, and count, each of which takes a set or multiset of values as input and returns a single value.

```
select avg (salary) from instructor where dept name= 'Comp. Sci.';
```

Basic aggregation in SQL involves computing an aggregate function on a set of tuples and returning a single tuple as the result.

For example, we can use the AVG function to compute the average salary of instructors in the Computer Science department by writing a SQL query that selects the average salary from the instructor table and restricts the results to instructors in the Computer Science department.

```
select avg (salary) as avg salary from instructor where dept name= 'Comp. Sci.';
```

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

Figure 31 - Tuples of the instructor relation, grouped by the dept.name attribute

In more complex cases, we may need to group the tuples in the input relation into subsets based on the values of one or more attributes, and then compute an aggregate function on each subset.

This is accomplished using the **GROUP BY** clause in SQL. For example, we can use the **GROUP BY** clause to compute the average salary of instructors in each department by grouping the tuples in the instructor table by department name and then computing the average salary for each group.

```
select count (distinct ID) from teaches  
      where semester = 'Spring' and year = 2010;  
  
select count (*) from course;
```

<i>dept.name</i>	<i>avg.salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 32 - The result relation for the query "Find the average salary in each department"

When using the **GROUP BY** clause, it is important to ensure that any attribute that appears in the **SELECT** clause but is not part of the **GROUP BY** clause is enclosed in an aggregate function, to avoid ambiguity in the query.

Similarly, we can use the **DISTINCT** keyword in the **COUNT** function to eliminate duplicate values before performing the computation. These powerful features of SQL make it a valuable tool for managing and analyzing large datasets in a variety of contexts.

```
select dept.name, avg(salary) as avg.salary  
      from instructor group by dept.name;  
  
select avg(salary) from instructor;
```

dept.name	avg(avg.salary)
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 33 – The result relation for the query “Find the average salary of instructors in those departments where the average salary is more than \$42,000.”

Nested Subqueries

Structured Query Language (SQL) provides a powerful mechanism for nesting subqueries, a technique that allows select-from-where expressions to be embedded within other queries.

This capability is frequently employed to perform a range of operations, including set membership tests, set comparisons, and set cardinality evaluations. One common use of nested subqueries is to test for set membership using the in-connective, which tests whether a given tuple belongs to a relation.

This is achieved by constructing a set of values using a select clause and testing for membership using the in operator. Conversely, the not-in operator is used to test for the absence of set membership.

```
select distinct course id from section  
      where semester = 'Fall' and year= 2009 and course id in  
            (select course id from section  
              where semester = 'Spring' and year= 2010);
```

SQL's flexibility enables users to craft queries using different approaches to arrive at the same results.

For example, consider the query "*Find all the courses taught in both the Fall 2009 and Spring 2010 semesters.*" The not-in construct is similarly used to test for membership in the absence of a set. The ability to test for membership in arbitrary relations further extends the capabilities of SQL.

```
select distinct course id from section  
      where semester = 'Fall' and year= 2009 and course id not in  
            (select course id from section  
              where semester = 'Spring' and year= 2010);
```

SQL provides a powerful tool for the discerning data analyst - the nested subquery. Such a subquery is essentially a select-from-where expression that is nested within another query, and it can be used to perform tests for set membership, make set comparisons and determine set cardinality.

One particularly interesting use of nested subqueries is to test for set membership using the 'in' connective in the where clause. For example, to find all courses taught in both the Fall 2009 and Spring 2010 semesters, we can first find all courses taught in the latter semester using a subquery, and then nest that subquery in the where clause of an outer query to find those courses that were also taught in Fall 2009. The resulting query is both flexible and efficient, allowing the user to approach the problem in a way that seems most natural.

```
select distinct course id from section  
    where semester = 'Fall' and year= 2009 and course id not in  
        (select course id from section  
            where semester = 'Spring' and year= 2010);
```

Nested subqueries can also be used to compare sets, as in the query "*Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.*" This can be written using the > some construct, which generates the set of all salary values of instructors in the Biology department and compares them to the salary values of the instructors in the outer query.

```
select distinct T.name from instructor  
    as T, instructor as S  
    where T.salary > S.salary  
    and S.dept name = 'Biology';  
  
select name from instructor  
    where salary > all  
        (select salary from instructor  
            where dept name = 'Biology');
```

SQL offers a wide range of options for testing set membership and comparison using nested subqueries, making it an indispensable tool for data analysts in their quest for insights and knowledge.

SQL is the lingua franca, enabling users to execute complex queries and sift through mountains of data with ease. But beyond its standard comparison operators, SQL boasts a number of lesser-known, yet equally powerful features that can be harnessed to produce sophisticated results.

```
select dept name from instructor  
group by dept name  
having avg (salary) >= all  
(select avg (salary) from instructor group by dept name)
```

For instance, by using set comparisons, one can identify departments with the highest average salary, or courses taught in both the Fall 2009 and Spring 2010 semesters. These queries involve the use of subqueries, including correlated subqueries, and scoping rules to ensure that the right data is being accessed.

Additionally, SQL includes features for testing empty relations, checking for duplicate tuples, and testing the absence of duplicate tuples. With the "exists" construct, one can test whether a subquery has any tuples in its result, while the "unique" construct returns true if the argument subquery contains no duplicate tuples.

```
select T.course id from course s T  
where unique (select R.course id from section as R  
where T.course id= R.course id and R.year = 2009);
```

By utilizing these advanced features of SQL, users can gain greater insights into their data and extract more value from it. It is a testament to the power and versatility of this programming language that it remains a vital tool in the ever-expanding world of big data.

Modification of the Database

In the world of database management, the ability to **add**, **remove**, or **change** information is a critical aspect of data processing. While we have thus far focused on information extraction, the manipulation of data is an equally vital function, made possible through the power of SQL.

Deletion is a key operation that can be expressed in a way that is similar to a query, allowing for the removal of entire tuples. It is important to note that only whole tuples can be deleted; values on specific attributes cannot be removed.

```
delete from r where P;
```

The predicate in the “*where*” clause can be as complex as that of a select command’s “*where*” clause. The request “*delete from instructor*” is an example of deleting all tuples from the “*instructor*” relation. However, the “*instructor*” relation itself still exists but is emptied of all tuples.

```
delete from instructor  
where dept name in (select dept name  from department  
where building = 'Watson');
```

Insertion of data is achieved by either specifying a tuple to be inserted or writing a query that results in a set of tuples to be **inserted**. In both cases, the attribute values for inserted tuples must be members of the corresponding attribute’s domain, and the tuples inserted must have the correct number of attributes.

```
insert into student values ('3003', 'Green', 'Finance', null);
```

The tuple inserted by this request specified that a student with ID “3003” is in the Finance department, but the tot cred value for this student is not known.

Consider the query:

- The simplest insert statement involves inserting one tuple, with attribute values specified in the order in which the corresponding attributes are listed in the relation schema.
- In more complex cases, we may want to insert tuples on the basis of the result of a query. SQL allows this through the use of the “*insert into*” statement, followed by a “*select*” statement that specifies the set of tuples to be inserted. This is particularly useful when inserting data into a relation with a large number of attributes.

It is essential to evaluate the “*select*” statement fully before any insertions are carried out to avoid unexpected results.

```
select student from student where tot_cred > 45;
```

Deletion and insertion of data are the two most common operations used for modifying a database, and SQL provides a powerful and flexible way to accomplish these tasks. By using SQL, we can add, remove, or change data with ease, allowing us to better manage and analyze vast amounts of information.

```
update instructor set salary = salary *1.05 where salary < 70000;
```

SQL, the language of relational databases, offers a versatile set of tools for manipulating data in tables. One of the most frequently used operations is the update statement, which modifies the values of selected tuples in a given relation. To identify which tuples to update, SQL uses a where clause that can include any construct legal in the where clause of a select statement, including nested selects.

In the case of a nested select within an update statement, the relation being updated may be referenced. SQL tests all tuples in the relation to see whether they should be updated and carries out the updates afterward.

As a practical example, consider the request “*Give a 5 percent salary raise to instructors whose salary is less than average*”. This can be expressed in SQL as an update statement that includes a nested select to calculate the average salary of instructors.

```
update instructor set salary = salary *1.05  
    here salary < (select avg (salary) from instructor);  
  
update instructor  
    set salary = salary *1.03  
        where salary > 100000;  
  
update instructor  
    set salary = salary *1.05  
        where salary <= 100000;
```

In some cases, it may be necessary to perform multiple updates with different criteria. In these situations, SQL provides a case construct that allows for multiple conditions to be evaluated and updated within a single statement.

Additionally, scalar subqueries can be useful in the set clause of update statements, allowing for complex calculations to be performed on a per-tuple basis.

2.3 Intermediate SQL

This chapter delves deeper into SQL, covering complex query forms, view definitions, transactions, integrity constraints, detailed data definition, and authorization.

Join Expressions

In this section, we delve deeper into join operations in SQL beyond the natural join.

The natural join requires matching values across specified attributes, whereas the join operator we examine in this section enables us to specify an explicit join predicate, allowing us to broaden the criteria for matching tuples.

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>tot.cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 34 - The student relation

We use the student and takes relations, depicted in the Figures displayed above, respectively, to illustrate the examples discussed in this section. Of note is the null value for the grade attribute associated with a student with ID 98988 in the takes relation. This null value indicates that the grade for that course has not been awarded yet.

<i>ID</i>	<i>course.id</i>	<i>sec.id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	null

Figure 35 - The takes relation

We begin with the on condition, which is a form of joint that can accommodate a general predicate over the relations being joined. The on condition is written like a where clause predicate, except that the keyword on is used instead of where. As the name implies, the on condition appears at the end of the join expression.

To illustrate, consider the following query that employs the on condition:

```
Copy code select * from student
join takes on student.ID= takes.ID;
```

ID	name	dept.name	tot.cred	course.id	sec.id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null

Figure 36 - The result of student join takes on student.ID= takes.ID with second occurrence of ID omitted

The on condition stipulates that a tuple from the student relation matches a tuple from the takes relation if their ID values are equal. The resulting join expression is similar to the student natural join takes operation, except that the ID attribute is listed twice in the join result, once for the student and once for takes. However, the two ID attributes must have the same value.

Interestingly, the preceding query is equivalent to the following one:

```
select * from student, takes  
where student.ID= takes.ID;
```

As seen earlier, the relation name is used to disambiguate the attribute name ID, allowing us to refer to the two occurrences as students.ID and takes.ID, respectively.

A version of the query that displays the ID value only once is:

```
select student.ID as ID,  
name, dept name, tot cred, course id, sec id, semester, year,  
grade from student  
join takes on the student.ID= takes.ID;
```

Join types	Join conditions
inner join	natural
left outer join	on < predicate >
right outer join	using (A_1, A_2, \dots, A_n)
full outer join	

Figure 37 - join types and join conditions

The query result is depicted in the Figure.

While the on condition allows for more expressive join predicates than the natural join, it may seem redundant since an equivalent expression without the on condition can be written, with the predicate in the on clause moved to the where clause.

Views

In today's world of data management, it is of utmost importance to ensure that sensitive information is kept secure and out of the reach of unauthorized personnel.

In the realm of databases, this can be achieved through the use of views.

Views are virtual relations that can be defined in SQL through the **CREATE VIEW** command. They allow users to have a customized, personalized collection of relations that is better matched to their needs and intuition than the logical model. For instance, consider a clerk who needs to know an *instructor's ID, name, and department*, but should not have authorization to view the salary amount.

A view relation, such as '*faculty*', can be created for the clerk, containing only the necessary attributes.

```
select course.course id, sec id, building, room number from course,  
section where course.course id = section.course id  
and course.dept name = 'Physics'  
and section.semester = 'Fall'  
and section.year = '2009';
```

Views are not precomputed and stored, but are computed by executing the query whenever the view is used. Thus, the view relation is created whenever needed, on demand. This ensures that the information is always up to date and accurate, as the tuples are generated by computing the query result each time.

```
create view faculty as select ID, name, dept name from instructor;
```

Furthermore, it is possible to create multiple views on top of any given set of actual relations, allowing for a large number of views to be supported. Using the view name, users can refer to the virtual relation that the view generates and use it in SQL queries.

```
create view departments total salary(dept name, total salary) as select  
dept name, sum (salary) from instructor group by dept name;
```

Views in SQL provide a secure and flexible way of accessing and manipulating data, allowing users to have a customized and intuitive collection of relations that are always up to date and accurate.

To address the aforementioned challenge, some database systems offer the concept of materialized views. These views not only store the results of the query, but also keep them up-to-date with any changes to the underlying base relations.

However, the maintenance of materialized views comes with its own set of concerns, including storage costs and added overhead for updates. Despite these trade-offs, applications that frequently use views, or demand fast response times for large aggregates can benefit greatly from the use of materialized views.

```
create view instructor info as select ID, name, building
from instructor, department
where instructor.dept name= department.dept name;
```

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	<i>null</i>	<i>null</i>

instructor

<i>dept.name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
<i>null</i>	Painter	<i>null</i>

department

Figure 38 - Relations *instructor* and *department* after insertion of tuples

On the other hand, modifying the database through views is fraught with difficulties. This is because any change to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

For instance, the addition of a tuple through a view raises the question of how to assign a value for the salary attribute, given that it is not present in the view. Similarly, the insertion of a tuple through a view with multiple relations may require the use of nulls, which can lead to unexpected and undesired results.

```
insert into instructor info values ('69987', 'White', 'Taylor');
```

Given these complexities, most database systems limit the use of updates, insertions or deletions through views, and specify different conditions under which they permit them. The challenge of modifying databases through views has been a subject of considerable research, and further investigation is necessary to determine the most effective solutions.

Transactions

In the realm of database management, transactions serve as a vital tool to ensure the integrity of data in the face of errors or system failures. A transaction is a series of query and/or update statements, with the SQL standard mandating that the beginning of a transaction is triggered implicitly by the execution of an SQL statement. The completion of a transaction can occur in one of two ways: by **committing the transaction** via the "*commit work*" statement, which makes the updates performed by the transaction a permanent part of the database, or by **rolling back** the transaction via the "*rollback work*" statement, which undoes all the updates performed in the transaction and restores the database to its previous state.

In this way, transactions can be thought of as atomic operations that are indivisible - either all of the transaction's effects are reflected in the database, or none are. Transactions are especially important in applications such as banking or university management, where the updating of multiple pieces of data must occur simultaneously. If, for example, a power outage occurs after the subtraction of funds from one account in a banking application, but before the addition of funds to another account, the balances will be inconsistent.

It should be noted that some SQL implementations treat each SQL statement as a transaction in its own right, with individual statements being committed as soon as they are executed. To execute a transaction consisting of multiple SQL statements, the automatic commit of individual SQL statements must be turned off. Alternatively, multiple SQL statements can be enclosed between the keywords "*begin atomic*" and "*end*", as per the *SQL:1999 standard*.

While the specifics of implementing transactions may vary depending on the SQL implementation, it is clear that they are an essential tool for ensuring the reliability and consistency of data in complex applications. The intricacies of transactions and their implementation are explored further in later chapters.

Integrity Constraints

Integrity constraints are vital in ensuring that any changes made to a database do not compromise the system's data consistency. These constraints serve as safeguards against inadvertent damage to the database, which can occur due to various reasons, including human error. Examples of integrity constraints include ensuring that an instructor's name is not null, no two instructors have the same instructor ID, and a department's budget is greater than \$0.00.

While an integrity constraint can be any arbitrary predicate pertaining to the database, testing such constraints can be expensive. Therefore, most database systems allow users to specify integrity constraints that can be tested with minimal overhead. These constraints are usually identified during the database schema design process and declared as part of the create table command used to create relations. However, integrity constraints can also be added to an existing relation using the command "*alter table table-name add the constraint*."

```
name varchar(20) not null budget numeric(12,2) not null
```

There are several types of integrity constraints that can be included in the create table command, such as not **null**, **unique**, and **check**. The not null constraint, for instance, restricts the domain of certain attributes to exclude **null values**. This constraint prohibits the insertion of a null value for the attribute and generates an error diagnostic if attempted. Similarly, the unique constraint specifies that no two tuples in the relation can be equal on all the listed attributes, with candidate key attributes allowed to be null unless explicitly declared **not null**.

```
unique (Aj1, Aj2,..., Ajm)
```

Overall, the use of integrity constraints is crucial in maintaining data consistency in a database system, and their inclusion in the design process can help prevent costly errors and ensure the reliability of the system.

The SQL language provides a variety of mechanisms to ensure data integrity in relational databases, including the check clause and referential integrity constraints. When used in a relation declaration, the check clause allows the specification of a predicate that must be satisfied by every tuple in the relation, effectively creating a powerful type system that restricts attribute domains in ways that most programming-language type systems cannot.

For example, a clause such as “*check (budget > 0)*” would ensure that the value of the budget attribute in a relation called department is nonnegative.

```
create table section
  (course id varchar (8),
   sec.id varchar (8),
   semester varchar (6),
   year numeric (4,0),
   building varchar (15),
   room number varchar (7),
   time slot id  varchar (4),
   primary key
     (course id, sec id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring',
   'Summer')));
```

Referential integrity, on the other hand, ensures that a value appearing in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

Foreign keys can be used to specify referential integrity constraints in SQL, with the foreign key clause declaring that for each tuple in a relation, a specified attribute must exist in another relation. The referenced attribute can be a **primary key** or a **candidate key**, which is declared using either a primary key or a unique constraint.

```
dept name varchar(20) references department
```

While the SQL standard permits the use of subqueries in the predicate of a check clause, none of the widely used database products currently support this feature. Nonetheless, the check clause and referential integrity constraints are valuable tools for ensuring the correctness and consistency of relational data.

In the realm of relational databases, referential-integrity constraints are a critical element in maintaining consistency and accuracy.

These constraints act as guardians, ensuring that any updates or modifications to the database do not result in a violation of the data relationships. Typically, when a referential-integrity constraint is violated, the system's standard response is to reject the action, and the transaction performing the update is rolled back. However, with the use of foreign key clauses, a violation of the constraint can be handled differently.

```
create table classroom
    (building varchar (15),
     room number varchar (7),
     capacity numeric (4,0),
     primary key
        (building, room number))

create table department
    (dept name varchar (20),
     building varchar (15),
     budget numeric (12,2)
     check (budget >0),
     primary key (dept name))

create table course
    (course id varchar (8),
     title varchar (50),
     dept name varchar (20),
     credits numeric (2,0) check (credits >0),
     primary key
        (course id), foreign key (dept name) references department)
```

```

create table instructor
  (ID varchar (5), name varchar (20),
   not null dept name varchar (20),
   salary numeric (8,2),
   check (salary > 29000),
   primary key (ID),
   foreign key (dept name)
   references department)

```

Create table section

```

  (course id varchar (8),
   sec id varchar (8),
   semester varchar (6),
   check (semester in
   ('Fall', 'Winter', 'Spring', 'Summer')), year numeric (4,0),
   check (year > 1759 and year < 2100)

   building varchar (15),
   room number varchar (7),
   time slot id varchar (4),

   primary key (course id, sec id, semester, year),
   foreign key (course id)
   references course,
   foreign key (building, room number) references classroom)

```

By including a foreign key clause in the schema, developers can specify that instead of rejecting the action, the system must take measures to restore the constraint.

For instance, if a tuple is deleted from the referenced relation, causing a violation of the integrity constraint, the system will not reject the deletion. Instead, if a "*delete cascade*" clause is included in the foreign-key declaration, the system will propagate the deletion to the referencing tuples in the dependent relation.

Similarly, an "*update cascade*" clause will result in the system updating the referencing tuples to reflect the updated value.

It is worth noting that the use of null values in foreign keys can complicate the application of referential-integrity constraints. While null values are allowed for attributes of foreign keys, the system's response to them may not always be the correct choice.

Therefore, SQL provides developers with constructs that allow them to specify alternative behaviors when dealing with null values.

```
create table course
(
    ...
    foreign key (dept name) references department
        on delete cascade
        on update cascade,
    ...
);
```

Overall, the handling of referential-integrity constraints is a vital component of maintaining data accuracy and consistency. With the proper use of foreign key clauses and a clear understanding of how null values impact the constraints' application, developers can ensure that their relational databases function correctly and reliably.

In the world of databases, maintaining data integrity is of utmost importance. While the SQL standard offers constructs to specify integrity constraints, it must be noted that these constructs are not currently supported by most database systems. However, they can include subqueries in the check clause to verify referential-integrity constraints.

Complex check conditions and assertions can be effective tools **to ensure data integrity**. Nevertheless, they can be a double-edged sword due to the high cost of testing and maintaining them. **Assertions**, which express conditions that the database must always satisfy, can be used to articulate constraints that are not expressible using domain constraints or referential-integrity constraints.

When an assertion is created, it undergoes rigorous testing for validity. If valid, future modification to the database is allowed only if it does not violate the assertion. However, this testing can introduce significant overhead when dealing with complex assertions, and thus should be utilized with care.

Despite the limitations of most database systems in supporting sub-queries in the check clause predicate or the create assertion construct, equivalent functionality can still be implemented using triggers. The referential-integrity constraint on time slot ID, for example, can be implemented using triggers.

```
create assertion  
credits_earned_constraint  
  
check (not exists (select ID   from student  where tot_cred <> (select  
sum(credits)  
  
from takes natural join course  
  
where student.ID= takes.ID  and grade is not null and  
grade<> 'F' )
```

The world of databases may be complex, but it is an essential aspect of modern-day life. As such, it is important to remain vigilant when maintaining data integrity and to utilize the most effective tools available to do so.

SQL Data Types and Schemas

We explored integer types, real types, and character types, and now we shall unravel some of the lesser-known data types supported by SQL, in addition to creating basic user-defined types.

Such a set of data types is the date and time types. These allow for the storage of calendar dates and the time of day in hours, minutes, and seconds. A variant of the time type allows for the specification of fractional digits for seconds, and time zone information can also be stored along with the time.

Another data type is the timestamp, which combines date and time. It too has a variant that allows for the specification of the number of fractional digits for seconds, and time-zone information can be included as well.

```
date '2001-04-25' time '09:30:00'  
timestamp '2001-04-25 10:29:01.45'
```

To specify a date and time value, one can use the format year followed by month followed by day. A string can be converted to the appropriate data type using the cast expression.

```
reate table student  
(ID varchar (5),  
 name varchar (20) not null,  
 dept name varchar (20),  
 tot cred numeric (3,0)  
 default 0, primary key (ID));
```

SQL provides several functions to extract individual fields from a date or time value, and it allows comparison operations on all the types listed here. SQL also offers a data type called interval that enables computations based on dates and times.

Furthermore, SQL permits the specification of default values for attributes, as demonstrated by the create table statement. The default value is set to 0 for the tot cred attribute in this example, and this value is assigned when a tuple is inserted into the student relation without a value for the tot cred attribute.

```
insert into student(ID, name, dept name)
values ('12789', 'Newman', 'Comp. Sci.');
```

In essence, **SQL's data types** and **schemas** offer a range of options for the storage and manipulation of data, making it a powerful tool in the hands of those who know how to wield it.

In the world of database management, indexing is a crucial tool for efficient querying. With many queries referencing only a small proportion of records, it's wasteful for the system to scan through every tuple of relation to find the desired results. That's where index creation comes in.

```
Create index studentID index on student(ID);
```

An index is a data structure that allows a database system to efficiently find tuples in a relation that have a specific value for an attribute, without the need to scan through all the tuples. This is achieved by creating an index on the attribute, allowing the system to retrieve the desired tuples directly.

```
book review clob(10KB)
image blob(10MB)
movie blob(2GB)
```

In addition to single attribute indices, indices can also be created on a list of attributes, such as a combination of name and department name. While the SQL language does not provide syntax for creating indices, many databases offer support for index creation via SQL statements.

When it comes to storing large objects, such as high-resolution medical images or video clips, SQL provides large-object data types. These data types, known as **CLOB** and **BLOB**, allow for the storage of kilobyte to gigabyte-scale attributes.

However, retrieving entire large objects into memory can be impractical or inefficient. Instead, applications often use an SQL query to retrieve a locator for a large object and then manipulate the object using the locator. The locator can be used to fetch the large object in small pieces, much like reading data from a file using a read function call.

SQL provides support for user-defined types, enabling the creation of distinct types to enhance data integrity and prevent programming errors. This form of user-defined data types allows for the creation of unique data types that can be used as attributes of relations such as Dollars and Pounds, which are defined as decimal numbers with 12 total digits, two of which are placed after the decimal point.

```
create type Dollars as numeric(12,2) final;  
create type Pounds as numeric(12,2) final;
```

By utilizing strong type checking, SQL can identify and prevent programming errors such as assigning a value of type Dollars to a variable of type Pounds, which may be due to a programmer forgetting the differences in currency. Additionally, the use of domains, introduced in *SQL-92*, can add integrity constraints to an underlying type, such as not null and default values.

```
create table department dept name  
        varchar (20),  
        building varchar (15),  
        budget Dollars);
```

Domains differ from user-defined types in that they can have constraints and default values specified on them, while user-defined types are designed to be used in procedural extensions to SQL where it may not be possible to enforce constraints. Furthermore, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible, whereas user-defined types are strongly typed.

```
create domain DDollars as numeric(12,2) not null;  
create domain YearlySalary numeric(8,2)  
        constraint salary value test check(value >= 29000.00);  
create domain degree level varchar(10)  
        constraint degree level test  
        check (value in ('Bachelors', 'Masters', or 'Doctorate'));
```

SQL's support for user-defined types and domains provides a robust and flexible system for managing complex data structures and enforcing data integrity constraints.

Creating tables with similar schemas as existing tables can prove to be a challenging and time-consuming task for developers. However, SQL offers a useful tool in the form of a “*create table-like*” extension that can assist in this endeavor. By executing a statement such as “*create table temp instructor like the instructor*,” a new table can be generated with the same schema as an existing one.

```
create table temp instructor like instructor;
```

In addition, developers may find it useful to store the result of a complex query as a new table. The *SQL:2003* standard provides a simpler and more efficient technique for this task, enabling the creation of a temporary table containing the results of a query with a single statement. This method can be accomplished by executing a command such as “*create table t1 as (select * from instructor where dept name= ‘Music’) with data*.”

```
create table t1 as  
  (select *  
   from instructor  
   where dept name= ‘Music’)  
   with data;
```

It is worth noting that some database management systems may utilize slightly different syntax to execute these tasks. Nevertheless, the *create table ... as* statement bears resemblance to the *create view* statement in that both rely on queries for their definitions. The main difference is that the contents of a table are set when the table is created, whereas a view’s contents always reflect the current query result.

Early file systems stored all files in a single directory, making it difficult to name files uniquely. Similarly, early database systems had a single namespace for all relations, which led to coordination issues. To solve this problem, contemporary database systems provide a three-level hierarchy for naming relations: **catalogs**, **schemas**, and **SQL objects** such as relations and views. Each user has a default catalog and schema, and the combination is unique to the user. Different applications and users can work independently without worrying about name clashes.

```
catalog5.univ schema.course
```

The default catalog and schema are part of an SQL environment that is set up for each connection. We can create and drop schemas by means of *create schema* and *drop schema* statements. In most database systems, schemas are created automatically when user accounts are created. The schema name is set to the user account name, and it becomes the default schema for the user account. However, creation and dropping of catalogs is implementation-dependent and not part of the SQL standard.

Authorization

Amidst the labyrinthine universe of database management, Authorization looms large as a fundamental pillar upon which the entire infrastructure of access control rests. At its core, Authorization governs the granting of privileges to users for accessing, inserting, updating, or deleting data - each of which is deemed a distinct privilege.

These authorizations can be bestowed upon a user for a particular part of a database, such as a relation or a view, in combination or exclusively, according to the need of the hour.

However, the database system must ensure that each query or update that is submitted by the user is first checked against the granted authorizations, and if found unauthorized, is promptly rejected. It is not only data authorizations but also schema authorizations that can be granted, providing users with the ability to create, modify or delete relations, amongst other functionalities.

Furthermore, a user with some form of authorization can also bestow or revoke privileges to other users as deemed necessary.

Amidst the broad tapestry of Authorization lies the all-encompassing and ultimate authority vested in the hands of the **Database Administrator**. With a sweeping range of permissions, the Database Administrator is granted the power to create new users, restructure the database, and wield control over the entire infrastructure with utmost finesse, much akin to the authority of a superuser, administrator, or operator for an operating system.

The SQL standard is no stranger to privilege **authorization**, featuring **select**, **insert**, **update**, and **delete privileges** as the norm. In a single command, a user can grant all available privileges using the convenient shorthand of 'all privileges'. Whenever a new relation is created, the creator is automatically given all privileges on that relation, providing seamless integration into the system.

Authorization conferral is done through the use of the grant statement in the SQL data-definition language. The basic format of the statement includes a privilege list, a relation or view name, and the intended recipient user or role list. Roles are expounded on further in the subsequent section.

Select authorization is required for the reading of tuples in a relation. For example, granting Amit and Satoshi select authorization on the department relation allows them to read its tuples.

```
grant < privilege list >
    on <relation name or view name> to <user/role list>;
grant select on department to Amit, Satoshi;
```

Update authorization on a relation is granted on all attributes of the relation or on only a specific few, determined by the inclusion of the list of attributes in the grant statement. Using the grant statement, a user can give Amit and Satoshi update authorization only on the budget attribute of the department relation.

```
grant update (budget) on department to Amit, Satoshi;
```

Insert authorization permits users to insert tuples into a relation while deleting authorization grants users the ability to remove tuples. The '*public*' user designation refers to all present and future users of the system, so granting privileges to the public confers those privileges to all users.

```
revoke < privilege list >
    on <relation name or view name> from <user/role list>;
revoke select on department from Amit, Satoshi;
revoke update (budget) on department from Amit, Satoshi;
```

Although the SQL authorization mechanism allows for the **granting of privileges** on a relation or attributes of a relation, it does **not permit authorizations on specific tuples**. In order to revoke an authorization, a user may use the revoke statement, which follows the same basic format as the grant. However, revocation of privileges can become complicated if the user receiving the revocation has granted the same privilege to another user. Further details on this issue are explored in the following section.

The crucial importance of authorizations and access control becomes increasingly clear. In particular, the notion of roles has emerged as a powerful and elegant means of granting privileges to users in a more efficient and secure manner.

In a university setting, for instance, a variety of individuals may require access to various aspects of the database, such as *instructors*, *teaching assistants*, *students*, *deans*, and *department chairs*. Rather than having to manually assign authorizations to each **individual user**, roles can be created that encapsulate these **privileges** and be granted to users accordingly. This not only simplifies the administrative process but also minimizes the risk of security breaches by allowing for more granular control over who can access what.

Roles can be created and managed using SQL commands, such as the creation of a role with the “*Create role*” statement or the granting of privileges to a role with the “*grant*” statement. Moreover, the privileges of a user or role are not limited solely to those explicitly granted to them; they also inherit any privileges granted to roles that have been granted to them, creating a chain of permissions that can be managed and controlled at multiple levels.

```
create role instructor;
    grant select on takes to instructor;
    grant dean to Amit;
create role dean;
    grant instructor to dean;
    grant dean to Satoshi;
```

In short, the use of roles in access control is a versatile and essential tool in the arsenal of any database administrator, one that empowers them to grant and revoke privileges in a more efficient and secure manner.

In the realm of database management, authorization on views is a critical concern, as exemplified by a hypothetical scenario in which a staff member requires access to the salaries of faculty members in the Geology department, but is not authorized to view data pertaining to other departments. To address this issue, a view is created, aptly named “*geo instructor*”, that allows for access to only the necessary data while restricting access to sensitive information.

However, before processing any queries, the system must first verify that the user is authorized to access the requested data.

```
create view geo instructor as
    (select * from instructor
        where dept name = 'Geology');
    select * from geo instructor;
```

It is worth noting that creating a view does not automatically grant the creator all privileges on that view. In fact, the creator only receives privileges that do not exceed their existing authorization. For example, a user cannot be granted update authorization on a view without having update authorization on the underlying relations that were used to define the view. If a view is created on which no authorization can be granted, the system will deny the view creation request.

```
grant references (dept name) on department to Mariano;  
grant select on department to Amit with grant option;
```

The creation of functions and procedures, which can contain queries and updates, is also supported by SQL. By default, such functions and procedures inherit all the privileges of their creator, but this can be modified using the "*SQL security invoker*" clause introduced in *SQL:2003*.

This clause allows for the execution of functions and procedures under the privileges of the user who invokes them, as opposed to the privileges of the creator. This capability facilitates the creation of libraries of functions that can be executed under the same authorization as the invoker, enabling greater flexibility and security in database management.

```
revoke grant option for select on department from Amit;  
granted by current role
```

A primitive authorization mechanism for the database schema is specified by the SQL standard. According to this mechanism, only the schema owner can modify the schema, including the creation or deletion of relations, attributes, and indices.

However, users are permitted to declare foreign keys when creating relations, thanks to the SQL references privilege. This privilege is granted on specific attributes, allowing users to restrict deletion and update operations on the referenced relation.

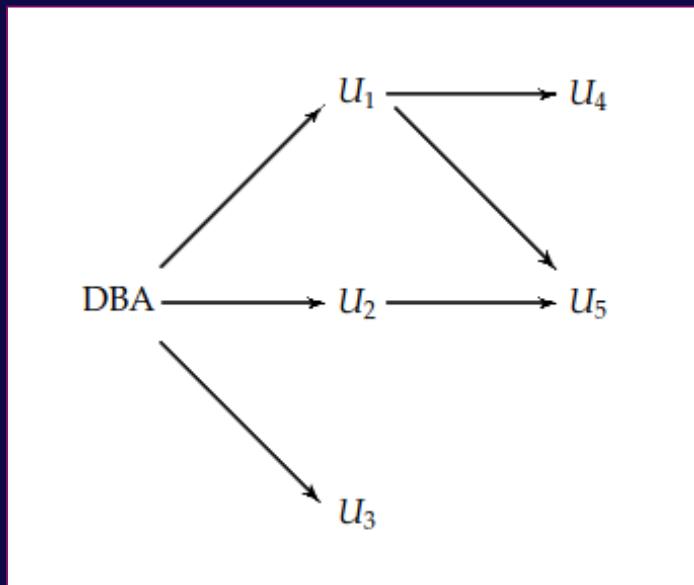


Figure 39 – Authorization-grant graph (U_1, U_2, \dots, U_5 are users and DBA refers to the database administrator)

But the ability to create foreign keys can restrict future activity by other users, as illustrated by the department relation example. If a user creates a foreign key referencing the department relation, then it is no longer possible to delete the referenced department without modifying the relation. Therefore, a reference privilege is necessary.

Users who are granted authorization may be allowed to pass on this authorization to other users, but by default, they are not authorized to grant the same privilege to others. To allow this, we use the "*with grant option*" clause when granting privileges, as shown in the example of granting select privileges on the department to Amit.

The creator of an object, such as a relation or a role, holds all privileges on the object, including the privilege to grant privileges to others.

Revoking authorization from a user may cause other users to lose the same privilege, a behavior known as cascading revocation. This can be prevented by using the "*restrict*" option in the revoke statement. While cascading revocation is often the default behavior, it can be inappropriate in some situations, such as when a user has granted privileges to another user. To avoid confusion and ensure that authorization is properly revoked, SQL ensures that authorization is revoked from both users, even if they have granted the same authorization to each other.

In the complex world of database management, understanding these authorization mechanisms is crucial for maintaining a secure and efficient system.

2.4 Advanced SQL

In this chapter, we delve into the more intricate features of the language. Our attention is drawn toward the pivotal issue of accessing SQL from a **general-purpose programming language** - an indispensable skill for constructing applications that employ a database to store and retrieve data.

We expound on the notion of procedural code execution within the database, an ingenious mechanism that either extends the SQL language to support procedural actions or enables functions defined in procedural languages to be executed within the database. We describe triggers, an exceptional tool that can automatically initiate designated actions when certain events, such as insertion, deletion, or update of tuples in a specified relation, occur.

Furthermore, we discuss the concept of recursive queries, a highly sophisticated technology that allows a query to reference itself. We also elucidate advanced aggregation features supported by SQL that facilitate complex analytical operations on large datasets. To conclude, we explicate the functionality of **online analytic processing (OLAP)** systems that enable the interactive analysis of vast datasets.

These advanced features are critical for constructing applications that rely on a database to store and retrieve data. They empower developers to automate actions and analyze massive datasets interactively, cementing SQL's status as a cornerstone of modern data management.

Accessing SQL From a Programming Language

SQL is a language that affords a potent and declarative means of querying data. As a result, composing SQL queries is usually an effortless endeavor when compared to the task of coding the same queries in a general-purpose programming language. However, it is worth noting that access to a general-purpose programming language is indispensable to a database programmer for two fundamental reasons.

Firstly, there are certain queries that cannot be expressed in SQL, given that the expressive power of SQL falls short of that of a general-purpose language such as *C, Java, or Cobol*. Consequently, for the purpose of composing such queries, SQL must be embedded within a more powerful language.

Secondly, non-declarative actions such as printing reports, user interactions, or transmission of query results to a graphical user interface cannot be achieved from within SQL. Given that querying or updating data is only one component of an application, other components written in general-purpose programming languages are indispensable. In this light, a means must be developed to fuse SQL with a general-purpose programming language for an integrated application.

Two main approaches to accessing SQL from a general-purpose programming language are *Dynamic SQL* and *Embedded SQL*. Dynamic SQL facilitates runtime construction and submission of an SQL query as a character string, while Embedded SQL involves identifying SQL statements at compile time using a preprocessor. The preprocessor submits SQL statements to the database system for recompilation and optimization before replacing the SQL statements in the application program with appropriate code and function calls prior to invoking the programming-language compiler.

One of the primary challenges in integrating SQL with a general-purpose programming language is the mismatch between the ways in which these languages manipulate data. While SQL operates on relations and returns relations as a result, programming languages usually operate on a variable at a time, and those variables correspond to the value of an attribute in a tuple in a relation. Thus, a mechanism must be developed to return the result of a query in a way that the program can handle.

This chapter examines two standards for connecting to an SQL database and performing queries and updates. The first standard is JDBC, an application program interface for the *Java* language. The second standard, **ODBC**, is an application program interface originally developed for the *C* language, but later extended to other languages such as *C++, C#, and Visual Basic*.

The Java Database Connectivity (**JDBC**) standard is an important tool that enables Java programs to connect with database servers. By providing an **application programming interface (API)**, **JDBC** allows Java developers to take advantage of database functionality while writing Java code. The API provides a range of capabilities, including opening and closing connections, executing statements, and processing results.

In order to use JDBC, developers must first *import java.sql.** package, which contains the interface definitions for JDBC functionality. Once this is done, the **JDBC API** can be used to execute SQL queries and interact with databases in a seamless and efficient manner. With its powerful functionality and ease of use, **JDBC** has become an essential tool for Java developers who need to work with databases.

```

{
try
{
Class.forName ("oracle.jdbc.driver.OracleDriver");
Connection conn =
    DriverManager.getConnection("jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                               userid, passwd);
Statement stmt = conn.createStatement();
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics',
98000)");
}
catch (SQLException sqle)
{
    System.out.println("Could not insert tuple. " +
sqle);
}
ResultSet rset = stmt.executeQuery(
    "select dept name, avg (salary) "+ " from instructor
"+ " group by dept name");
while (rset.next())
{
    System.out.println(rset.getString("dept name") + " "
+ rset.getFloat(2));
    stmt.close();
    conn.close();
}
catch(Exception sqle)
{
System.out.println("Exception : " + sqle);
}
}

```

Opening a connection to a database from a Java program is a crucial first step in the process of executing SQL statements. It requires a careful selection of the database to use, which can be an instance of Oracle running on a local machine or a PostgreSQL database running on a remote one. This initial connection is established using the **getConnection** method of the **DriverManager** class, which takes three crucial parameters.

The first parameter specifies the URL or machine name of the server, along with some additional information, including the *communication protocol* and *port number*. However, the actual protocol used to exchange information with the database is not defined by the **JDBC standard** and may vary depending on the driver being used.

To access the database from Java, the JDBC driver must be loaded first, which is achieved by invoking `Class.forName` with a concrete class implementing the `java.sql.Driver interface`. It is important to note that each database product that supports JDBC provides its own **JDBC driver**, which must be dynamically loaded before connecting to the database.

The second and third parameters of the **getConnection** method require a database user identifier and password, respectively. However, the need to specify a password within the **JDBC code** presents a significant security risk, particularly if an unauthorized person gains access to the Java code. Therefore, the process of establishing a connection to a database from a Java program requires a meticulous approach and attention to detail. By carefully selecting the appropriate database and driver, and taking the necessary security precautions, developers can ensure that their Java programs execute SQL statements smoothly and securely.

Once a connection to the database is established, *the Java program* gains the ability to send SQL statements for execution to the database system. To achieve this, the program makes use of an instance of the `Statement` class. It's worth noting that the `Statement` object is not the SQL statement itself, but rather an interface that facilitates the invocation of methods that execute SQL statements.

In the example provided, a *Statement handle (stmt)* is created on the *connection (conn)*, providing the program with the necessary means to execute SQL statements. To execute a statement, the program uses either the **executeQuery** method or the **executeUpdate** method, depending on the nature of the SQL statement. If the statement is a query statement that returns a result set, the **executeQuery** method is used.

Conversely, non-query statements, such as **update**, **insert**, **delete**, **create a table**, etc., are executed using the **executeUpdate** method.

The given example employs the stmt.executeUpdate method to execute an update statement that inserts data into the instructor relation. This method returns an integer value that corresponds to the number of tuples inserted, updated, or deleted. **DDL** statements, however, produce a return value of zero.

To address any exceptions that may arise during **JDBC calls**, the program employs a try-catch construct. This enables the program to handle any errors and communicate appropriate messages to the user.

By leveraging **Statement objects** and the appropriate execute methods, Java programs can effectively execute SQL statements and interact with a database system with ease.

Retrieving the result of a query is a critical aspect of data processing. The process involves executing a query and fetching the resulting set of tuples into a **ResultSet object**, which is then accessed one tuple at a time. In the Java programming language, this is done using the executeQuery method on a Statement object.

The **ResultSet object** provides a range of methods for fetching attributes from the fetched tuple, including getString and getFloat. These methods can be invoked using either the attribute name or the position of the desired attribute within the tuple.

However, it is crucial to close the connection and statement objects at the end of the program to avoid exceeding the limit of connections imposed on the database. Failure to do so may prevent the application from opening any more connections, disrupting the data processing pipeline.

```
PreparedStatement pStmt = conn.prepareStatement( "
    insert into instructor values(?, ?, ?, ?, ?)");
    pStmt.setString(1, "88877");
    pStmt.setString(2, "Perry");
    pStmt.setString(3, "Finance");
    pStmt.setInt(4, 125000);
    pStmt.executeUpdate();
    pStmt.setString(1, "88878");
    pStmt.executeUpdate();
```

To execute a prepared statement, which is a SQL statement with one or more parameters, the program can use an instance of the class **PreparedStatement**. In the example provided, a **PreparedStatement** object is created and used to insert tuples into the instructor relation. The **setString** and **setInt** methods are used to set the parameter values before executing the update using the **executeUpdate** method.

Retrieving the result of a query is a critical aspect of data processing. The process involves executing a query and fetching the resulting set of tuples into a **ResultSet** object, which is then accessed one tuple at a time. In the Java programming language, this is done using the **executeQuery** method on a Statement object.

The **ResultSet** object provides a range of methods for fetching attributes from the fetched tuple, including **getString** and **getFloat**. These methods can be invoked using either the attribute name or the position of the desired attribute within the tuple.

However, it is crucial to close the connection and statement objects at the end of the program to avoid exceeding the limit of connections imposed on the database. Failure to do so may prevent the application from opening any more connections, disrupting the data processing pipeline.

```
select * from instructor where name = 'X' or 'Y' = 'Y'
```

To execute a prepared statement, which is a SQL statement with one or more parameters, the program can use an instance of the class **PreparedStatement**. In the example provided, a **PreparedStatement** object is created and used to insert tuples into the instructor relation. The **setString** and **setInt** methods are used to set the parameter values before executing the update using the **executeUpdate** method.

Java Database Connectivity (JDBC) provides a robust and flexible way to connect Java applications to databases. One notable feature of JDBC is the **CallableStatement** interface, which enables the invocation of SQL-stored procedures and functions. **CallableStatement**, similar to **PreparedStatement** for queries, allows for the use of functions and procedures in Java applications. By registering out parameters with the **registerOutParameter()** method, we can retrieve the data types of function return values and procedure out parameters with the get methods.

This functionality provides Java applications with an easy way to interact with databases using stored procedures and functions.

```
ResultSetMetaData rsmd = rs.getMetaData();

for(int i = 1; i <= rsmd.getColumnCount(); i++)

{

    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));

}
```

Furthermore, JDBC's metadata features provide Java programs with the means to retrieve metadata about the database at runtime. This approach makes Java applications more robust to changes in the database schema. The **ResultSet** interface's **getMetaData()** method returns a **ResultSetMetaData** object containing metadata about the result set, which includes information such as the number of columns and the name and type of each column. The **DatabaseMetaData** interface provides even more metadata about the database system, allowing Java applications to obtain information such as product names, version numbers, and supported features.

```
DatabaseMetaData dbmd = conn.getMetaData();

ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");

// Arguments to getColumns: Catalog, Schema-pattern, Table-
// pattern,                                // and Column-Pattern

// Returns: One row for each column; row has a number of
// attributes                               // such as COLUMN NAME, TYPE NAME

while( rs.next())

{

    System.out.println(rs.getString("COLUMN NAME"),
    rs.getString("TYPE NAME"));

}
```

The metadata interfaces can be leveraged for a wide range of tasks, such as writing a database browser or generic code to display rows in a relation. Additionally, JDBC provides many other useful features, such as updatable result sets and transaction handling. By utilizing JDBC, Java developers can seamlessly integrate databases into their applications and take full advantage of their capabilities.

In a world where data reigns supreme, accessing and manipulating vast amounts of information is a task of paramount importance. To this end, the **Open Database Connectivity (ODBC)** standard provides a vital tool for applications seeking to open a connection with a database, send queries and updates, and retrieve results. It is a veritable lifeline for a vast array of applications, from graphical user interfaces to statistics packages to spreadsheets.

Each database system supporting **ODBC** provides a library that must be linked with the client program. In this way, **ODBC** acts as a bridge between client and server, facilitating the smooth and efficient communication of information. **The ODBC API** provides a range of functions for tasks such as finding all the relations in the database and finding the names and types of columns of a query result or a relation in the database.

The program first allocates an SQL environment and then a database connection handle, using ODBC-defined types such as **HENV**, **HDBC**, and **RETCODE**.

From there, the program opens the database connection by calling SQLConnect, passing parameters such as the connection handle, the server to which to connect, the user identifier, and the password for the database.

```
void ODBCexample() { RETCODE error; HENV env; /* environment / HDBC
conn; / database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL NTS, "avi", SQL NTS,
"avipasswd", SQL NTS);
{
    char deptname[80];
    float salary;
    int lenOut1, lenOut2;
    HSTMT stmt;
    char * sqlquery = "select dept name, sum (salary)
                      from instructor
                      group by dept name";
    SQLAllocStmt(conn, &stmt);
    error = SQLExecDirect(stmt, sqlquery, SQL NTS);
    if (error == SQL SUCCESS)
    {
        SQLBindCol(stmt, 1, SQL C CHAR, deptname , 80, &lenOut1);
        SQLBindCol(stmt, 2, SQL C FLOAT, &salary, 0 ,
&lenOut2);
        while (SQLFetch(stmt) == SQL SUCCESS)
        {
            printf ("%s %g\n", deptname, salary);
        }
    }
    SQLFreeStmt(stmt, SQL DROP);
}
SQLDisconnect(conn);
SQLFreeConnect(conn);
SQLFreeEnv(env);

}
```

Once the connection is established, SQL commands can be sent to the database via **SQLExecDirect**, with C language variables bound to attributes of the query result. **SQLBindCol** facilitates this task, with the second argument identifying the position of the attribute in the query result and the third indicating the type conversion required from SQL to C. On each fetch, SQLFetch retrieves the next result tuple, with attribute values stored in corresponding C variables. The program then prints out these values.

A good programming style requires that the result of every function call be checked for errors. At the end of the session, the program frees the statement handle, disconnects from the database, and frees up the connection and SQL environment handles.

```
EXEC SQL < embedded SQL statement >;
```

To use embedded SQL, a program must be processed by a preprocessor prior to compilation. This preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow for runtime execution of database accesses. Unlike **JDBC** or **ODBC**, where SQL statements are interpreted at runtime, some SQL-related errors can be caught at compile time when using embedded SQL.

The **EXEC SQL** statement is used to identify embedded SQL requests to the preprocessor, and the **SQL INCLUDE SQLCA** statement is used to identify the place where the preprocessor should insert special variables used for communication between the program and the database system. Before executing any SQL statements, the program must connect to the database using the **EXEC SQL** and connect to the server using a username using a password statement.

```
EXEC SQL BEGIN DECLARE SECTION;  
    int credit amount;  
EXEC SQL END DECLARE SECTION;
```

Variables of the host language can be used within embedded SQL statements but must be preceded by a colon (**:**) to distinguish them from SQL variables. Embedded SQL statements are similar in form to regular SQL statements but have several important differences. To write a relational query, we use the declare cursor statement, which creates a cursor for the query. The program must use the open and fetch statements to obtain the result tuples.

```
EXEC SQL  
    declare c cursor for  
        select ID, name  
        from student  
        where tot cred > :credit amount;
```

A single fetch request returns only one tuple, so the program must contain a loop to iterate over all tuples. **Embedded SQL** assists the programmer in managing this iteration, as the cursor is set to point to the first tuple of the result when the program executes an open statement on a cursor. Each time it executes a fetch statement, the cursor is updated to point to the next tuple of the result. The close statement is used to tell the database system to delete the temporary relation that held the result of the query.

Embedded SQL expressions for database modification do not return a result and are somewhat simpler to express. A database modification request takes the form **EXEC SQL**.

Overall, embedded SQL is a powerful tool for accessing and manipulating database systems, and its use can improve the efficiency and reliability of database operations.

```
EXEC SQL < any valid update, insert, or delete>;
```

Functions and Procedures

While the language comes with a number of pre-built functions, developers can also create their own specialized functions and procedures, store them in the database, and call them from SQL statements. With these custom functions, developers can add business logic to their databases, such as rules for how many courses a student can take in a given semester or how many majors they can be enrolled in.

But the benefits of custom functions don't stop there. Functions can also be used with specialized data types, such as images and geometric objects, where they can perform tasks like checking whether two line segments overlap or comparing two images for similarity. And by storing these functions within the database, developers can ensure that multiple applications have access to them and can change the business rules in one central location without needing to modify other parts of their application.

```
create function dept_count(dept_name varchar(20))  
    returns integer  
begin  
    declare d_count integer;  
    select count(*) into d_count  
        from instructor  
            where instructor.dept_name= dept_name  
    return d_count;  
end
```

SQL supports the creation of functions, procedures, and methods, which can be defined using either the procedural component of SQL or an external programming language like *Java*, *C*, or *C++*. And while different database systems may implement these syntaxes in slightly different ways, the concepts we've discussed remain relevant across implementations. So whether you're using *Oracle*, *Microsoft SQL Server*, or *PostgreSQL*, the power of custom functions and procedures is within your reach.

In the realm of the structured query language (SQL), the power of programming language constructs is readily available. To explore this domain, one must delve into the **Persistent Storage Module (PSM)** - a part of the SQL standard that outlines the language constructs available for writing procedures and functions.

SQL functions in particular, are discussed in detail. For example, define a function that, given the name of a department, returns the count of instructors in that department.

```
create function instructors of (dept name varchar(20))  
    returns table (  
        ID varchar (5),  
        name varchar (20),  
        dept name varchar (20),  
        salary numeric (8,2))  
  
return table  
(select ID, name, dept name, salary  
from instructor  
where instructor.dept name = instructor of.dept name);
```

Variables are declared using a declare statement, and SQL's support for almost all conditional statements makes it an extremely flexible language. From while statements to repeat statements and conditional statements like if-then-else, SQL's flexibility is unparalleled.

```
if boolean expression  
    then statement or compound statement  
    elseif boolean expression  
        then statement or compound statement  
    else statement or compound statement  
end if
```

SQL's PSM provides an incredible suite of tools that give it the power of a general-purpose programming language. With its support for functions and procedures, SQL provides great flexibility in developing applications. Its vast range of language constructs allows developers to write complex code that can handle almost any scenario.

SQL allows for functions to be defined in languages like *Java*, *C#*, *C*, or *C++*. These functions can be more efficient than SQL-defined functions and can perform computations that SQL cannot. However, external language procedures and functions must handle null values in parameters and return values, communicate failure/success status, and deal with exceptions. The exact mechanisms for handling these situations depend on the database system used.

```
create procedure dept count proc
    ( in dept name varchar(20), out count integer)
    language C external name '/usr/avi/bin/dept count proc'
create function dept count (dept name varchar(20))
    returns integer language C external name '/usr/avi/bin/dept
    count'
```

Although functions defined in programming languages and compiled outside the database system can be loaded and executed with the database-system code, this method carries significant risk.

Alternatively, if the code is written in a "safe" language such as *Java* or *C#*, it can be executed in a sandbox within the database query execution process itself. The sandbox allows the Java or C# code to access its memory area while preventing it from reading or updating the memory of the query execution process or accessing files in the file system. This method greatly reduces function call overhead by avoiding interprocess communication.

Today, several database systems support external language routines running in a sandbox within the query execution process, such as **Oracle** and **IBM DB2**, which allow Java functions to run as part of the database process. **Microsoft SQL Server** allows procedures compiled into the **Common Language Runtime (CLR)** to execute within the database process, written in languages like *C#* or *Visual Basic*. **PostgreSQL** also allows functions defined in several languages, including *Perl*, *Python*, and *Tcl*.

Triggers

Triggers are powerful mechanisms that can automatically execute certain actions in response to a modification of the database. To design a trigger mechanism, one must specify the trigger event and the conditions for execution, as well as the actions to be taken when the trigger fires. Once implemented, the database system assumes responsibility for executing the trigger whenever the specified event occurs and the conditions are met.

Triggers are often utilized to enforce integrity constraints that cannot be expressed using standard SQL constraints. Additionally, they can serve as valuable tools for automating tasks or alerting humans when certain conditions are met. Take for example a scenario where a student is enrolled in a course - a trigger can be designed to update the student's total credits whenever a new tuple is inserted into the "*takes*" relation. Similarly, a warehouse may use a trigger to automatically place an order when the inventory level of an item falls below the minimum level.

However, it's important to note that trigger systems are limited in their ability to perform updates outside of the database. Therefore, in the aforementioned warehouse example, a separate process must be created to periodically scan the "*reorders*" relation and place orders as needed. Some database systems do offer built-in support for sending emails from SQL queries and triggers, but the above approach remains the most common solution.

As we delve into the implementation of triggers in SQL, we are presented with a syntax that is recognized as standard, but the reality is that most databases utilize nonstandard versions of this syntax. Though the syntax we examine may not be upheld on such systems, the underlying concepts are universally applicable. In this regard, it is necessary to consider nonstandard trigger implementations, which we will discuss in greater detail later on in this section.

```
create trigger timeslot
check1 after insert
on section
referencing new row as nrow
for each row
when (nrow.time slot id not in (
    select time slot id
    from time slot) /* time slot id not present in time slot */
begin
    rollback
end;
create trigger timeslot check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time slot id not in (
    select time slot id
    from time slot)/*lasttuple for time slot id deleted
from time slot */
    and orow.time slot id in (
        select time slot id
        from section)) /* and time slot id still referenced from
section*/
begin
    rollback
end;
```

To illustrate the efficacy of triggers in enforcing referential integrity on the **time slot id attribute** of the section relation. The first trigger definition is initiated after any insert on the relation section, and it verifies that the time slot id value being inserted is valid. The referencing new row as clause establishes a transition variable called nrow, which retains the value of an inserted row after the insertion.

Furthermore, the when statement specifies a condition. Consequently, the system executes the remainder of the trigger body solely for tuples that satisfy the condition. The begin atomic...end clause is employed to collect several SQL statements into a single compound statement. However, in the given example, only one statement is included, which rolls back the transaction that caused the trigger to be executed. In this manner, any transaction that breaches the referential integrity constraint is automatically rolled back, ensuring that the data in the database satisfies the constraint.

```
create trigger reorder after update of amount on inventory
referencing old row as orow,
new row as nrow
for each row
when nrow.level <= (select level
from minlevel
where minlevel.item = orow.item) and orow.level > (select level
from minlevel
where minlevel.item = orow.item)
begin atomic
insert into orders
(select item, amount
from reorder
where reorder.item = orow.item);
end;
```

The trigger is only executed when the grade attribute is updated from a value that is either **null** or '**F**', to a grade that indicates successful completion of the course. The update statement is normal SQL syntax, except for the use of the variable nrow.

Triggers can be less efficient and less transparent than other techniques, and can even create unintended complications.

One example of a better alternative to triggers is the use of materialized views, which many modern database systems support. By contrast, triggers would require writing complex code to maintain the view, which could make it difficult for a database user to understand the constraints in the database. Similarly, while triggers can be used to maintain replicas of databases, built-in facilities for replication are now widely available, which often make triggers unnecessary.

```
select course id, sec id, semester, year, count(ID) as total students  
from takes group by course id, sec id, semester, year;
```

In addition to these issues, triggers can also cause problems when loading data from backup copies or replicating database updates on backup sites. In such cases, triggers may execute the triggered action more than once, which can lead to an infinite chain of triggering. As a result, database systems may limit the length of such chains or flag them as an error.

Despite these challenges, triggers remain a useful tool in many cases. However, they should be written with great care to avoid errors and complications. When possible, alternative techniques such as stored procedures should be considered instead. Ultimately, the choice of whether or not to use triggers depends on the specific needs and circumstances of the database in question.

course.id	prereq.id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 40 - The prereq relation

Recursive Queries**

"findAllPrereqs(cid)" take the course id of the course as a parameter (cid), computes the set of all direct and indirect prerequisites of that course, and returns the set. The function uses three temporary tables: **c_prereq**, **new_c_prereq**, and **temp**. The repeat loop then adds all courses in the **new_c_prereq** table to the **c_prereq** table. Next, it computes the prerequisites of all those courses in new_c_prereq, except those that have already been found to be prerequisites of cid, and stores them in the temporary table temp.

Finally, it replaces the contents of **new_c_prereq** by the contents of temp. The repeat loop continues until it finds no new (indirect) prerequisites.

It is worth noting that the use of the except clause in the function ensures that the function works even in the (abnormal) case where there is a cycle of prerequisites. This is particularly useful as cycles are possible in other applications. For example, in relation to "**flights(to, from)**" which says which cities can be reached from which other cities by a direct flight, the function can be adapted to find all cities that are reachable by a sequence of one or more flights.

```
create function findAllPrereqs(cid varchar(8))

    -- Finds all courses that are prerequisite (directly or indirectly)
for

returns table (course id varchar(8))

--The relation prereq(course id, prereq id) specifies which course is
-- directly a prerequisite for another course.

begin

create temporary table c_prereq (course id varchar(8));

        --table c_prereq stores the set of courses to be returned

create temporary table new_c_prereq (course id varchar(8));

        --table new_c_prereq contains courses found in the previous iteration

create temporary table temp (course id varchar(8));

        --table temp is used to store intermediate results

insert into new_c_prereq

    select prereq id

        from prereq
```

```

        where course id = cid;

repeat

    insert into c prereq

        select course id

        from new c prereq;

    insert into temp

        (select prereq.course id

        from new c prereq, prereq

        where new c prereq.course id = prereq.prereq

id

    )

except (

    select course id

    from c prereq

);

delete from new c prereq;

insert into new c prereq

    select *

    from temp;

delete from temp;

until not exists (select * from new c prereq) end repeat;

return table c prereq;

end

```

Overall, this innovative approach to computing the transitive closure of the relation "prereq" has significant implications for a wide range of hierarchical structures, including organizations and machines, where transitive closure can be used to find all parts in a system.

Using recursion in SQL, we can, for example, define the set of courses that are prerequisites for a particular course by recursively identifying courses that are prerequisites for other courses, until we arrive at the desired course. This approach can be expressed using the "*with recursive*" clause.

```
with recursive c prereq(course id, prereq id) as (
    select course id, prereq id
    from prereq
    union
    select prereq.prereq id, c prereq.course id from prereq, c prereq
    where prereq.course id = c prereq.prereq id
)
select *
from c prereq;
```

However, there are certain constraints to be aware of when constructing a recursive view. The recursive query must be monotonic, meaning that if more tuples are added to the view relation, the recursive query should return at least the same set of tuples as before, and possibly return additional tuples. Constructs such as aggregation do not exist in subqueries, and the set differences can cause the query to be nonmonotonic and should therefore be avoided.

Despite these constraints, recursive views remain a valuable tool in the database programmer's arsenal, providing a powerful means of expressing transitive closure with ease and efficiency. By applying an iterative procedure, we can compute a fixed point instance of the view relation, containing exactly the tuples necessary for expressing the desired transitive closure. So the next time you find yourself struggling with transitive closure in SQL, remember the power of recursion and the convenience of recursive view definitions.

Advanced Aggregation Features

In the field of database management, aggregation support in SQL is a powerful tool that can tackle most common tasks with ease. However, there are some complex tasks that prove to be difficult to implement efficiently with the basic aggregation features. This is where the ingenious additions to SQL come into play.

In this section, we explore the ranking feature added to SQL that enables users to find the position of a value in a larger set, a common operation in data analysis. For instance, in the academic world, professors may wish to rank their students based on their *grade-point average (GPA)*. The rank 1 goes to the student with the highest GPA, the rank 2 to the student with the next highest *GPA*, and so on. SQL's ranking feature allows users to accomplish this task with ease.

In addition to finding the rank, users can also find the percentile in which a value in a (multi)set belongs. While SQL's basic constructs can express such queries, they are often difficult to execute and inefficient. Programmers may resort to writing the query partly in SQL and partly in a programming language.

To illustrate the use of ranking in SQL, we can examine the example of a university's student grades (*ID, GPA*) view. The following query uses the rank function to give the rank of each student based on their *GPA*:

```
select ID, rank() over (order by (GPA) desc) as s_rank from student grades;
```

It is important to note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra order by clause is required to get them in sorted order. The rank function gives the same rank to all tuples that are equal on the order by attributes. In contrast, the dense rank function does not create gaps in the ordering. It assigns the same rank to tuples with the same *GPA* and increments the rank for the next *unique GPA value*.

Ranking can also be performed within partitions of the data. For example, if we want to rank students by department rather than across the entire university, we can use the partition by clause.

The following query gives the rank of students within each department:

```
select ID, dept name,  
       rank () over (  
           partition by dept name order by GPA desc) as dept rank  from dept  
grades  
       order by dept name,  
       dept rank;
```

Multiple rank expressions can be used within a single select statement. Moreover, when ranking (*possibly with partitioning*) occurs along with a group by clause, the group by clause is applied first, and partitioning and ranking are done on the results of the group by. This allows users to obtain aggregate values that can then be used for ranking.

While SQL's ranking functions can be used to find the top n tuples by embedding a ranking query within an outer-level query, several database systems provide nonstandard SQL extensions that simplify the job of the optimizer. For instance, some databases allow a clause limit n to be added at the end of an SQL query to specify that only the first n tuples should be output.

SQL's ranking feature is an innovative addition to its suite of tools that offers users a simple and efficient way to perform complex tasks. By leveraging SQL's ranking functions, users can easily rank their data based on various criteria, making data analysis a breeze.

The concept of windowing in databases is a powerful tool for computing aggregate functions over ranges of tuples. By specifying a window, which may overlap with other windows, one can compute aggregate values for specific time intervals, as well as analyze trends in various data sets.

One practical use of windowing is in trend analysis, such as analyzing sales figures or stock market trends. While it is relatively straightforward to compute aggregates over a fixed window of time, such as a *3-day period*, it becomes more challenging to do so for every *3-day period*. This is where the windowing feature of SQL comes in handy.

```
select year, avg(num credits)  
       over (order by year rows 3 preceding)  
       as avg total credits  from tot credits;
```

SQL allows for the computation of aggregate values over specific windows by using the “*rows preceding*” or “*rows following*” syntax. For example, a query could be written to compute the average number of credits taken by students over the previous three years, sorted by year. Additionally, the “*range between*” syntax can be used to specify a range based on the value of the order attribute.

```
select year, avg(num credits)
    over (order by year rows unbounded preceding) as avg total
  credits from tot credits;

select year, avg(num credits)
    over (order by year rows between 3 preceding and 2 following)
        as avg total credits
from tot credits;
```

Furthermore, windowing can be used to partition data by specific attributes, such as department names, in order to compute aggregate values for each subset of data. This is done by including a “*partition by*” clause in the SQL query.

```
select dept name, year, avg(num credits)
    over (partition by dept name
          order by year rows between 3 preceding and current row)
        as avg total credits from tot credits dept;
```

Overall, the windowing feature in SQL provides a powerful tool for analyzing trends and computing aggregate values over specific windows of data. Its flexibility and ease of use make it an essential tool for data analysis and decision-making in a wide variety of fields.

OLAP

The art of data analysis has come a long way since the days of spreadsheet applications. In today's world, businesses are looking for more sophisticated ways to make sense of their vast amounts of data. Enter the online **analytical processing (OLAP)** system, a revolutionary tool that allows analysts to view summaries of multidimensional data in an interactive and speedy fashion.

```
sales (item name, color, clothes size, quantity)
```

item.name	color	clothes.size	quantity
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2

Figure 41 - An example of sales relation

OLAP systems are available in many forms, including those that come bundled with database products like **Microsoft SQL Server** and **Oracle**, as well as standalone tools. While spreadsheet applications can handle smaller amounts of data, **OLAP systems** are necessary for larger datasets that require efficient preprocessing and online query processing support from databases.

To better understand the power of **OLAP**, let's consider a hypothetical scenario. A clothing store wants to know what types of clothes are popular among their customers. By using a sales relation with attributes such as item name, color, and clothes size, managers can group their data and view summaries in a cross-tabulation (*or pivot table*) format. The table shows total quantities for different combinations of item name and color, with clothes size being a summary across all values.

Multidimensional data, with measure and dimension attributes, is essential for this type of analysis. **OLAP systems** allow managers to quickly group and view data in a cross-tabulation format, providing valuable insights for business decisions. By extending SQL to support these tasks, **OLAP systems** are at the forefront of data analysis technology.

		color			
		dark	pastel	white	total
item_name	skirt	8	35	10	53
	dress	20	10	5	35
	shirt	14	7	28	49
	pants	20	2	5	27
	total	62	54	48	164

Figure 42 – Cross tabulation of sales by item name and color

The figure displays a 3D data cube with dimensions: item_name (skirt, dress, shirt, pants, all), color (dark, pastel, white, all), and clothes_size (small, medium, large, all). The values represent sales counts, with some cells containing summary values like 16, 18, 45, etc.

		item_name								
		skirt	dress	shirt	pants	all				
		2	5	3	1	11				
		4	7	6	12	29				
		2	8	5	7	22				
color		dark	8	20	14	20	62	4	16	
		pastel	35	10	7	2	54	34	18	
		white	10	8	28	5	48	9	45	
		all	53	35	49	27	164	21	42	
			skirt	dress	shirt	pants	all	small	medium	large
							all			
										clothes_size

Figure 43 – Cross tabulation of sales by item name and color

Use of cross-tabs has become ubiquitous. These tables, which summarize data by showing the intersection of two attributes, are a simple and powerful tool for understanding complex data sets. Most cross-tabs include summary rows and columns, which provide totals for the cells in each row or column.

But what happens when a data set has more than two attributes? Enter the data cube, a multi-dimensional version of the cross-tab that can be visualized as an n-dimensional cube. Each cell in the data cube contains a value, just like a cross-tab, but now the cell is identified by values for multiple dimensions.

The number of different ways to group tuples for aggregation can be vast. For instance, a table with three dimensions for item name, color, and clothes size results in a cube with a size of **$3 \times 4 \times 3 = 36$** . Including summary values, the cube is **$4 \times 5 \times 4$** with a size of **80** . Aggregation can be performed with grouping on each of the **$2n$ subsets** of the n dimensions.

OLAP systems take the concept of the data cube to the next level, allowing data analysts to interactively select attributes for their cross-tabs. Analysts can pivot their views to create two-dimensional cross-tabs on different attributes or slice the data to view a single value across multiple dimensions.

Drilling up and down hierarchies allows analysts to view data at different levels of granularity, which can be organized into a hierarchy. For example, the hierarchy of location can go from city to state to country to region, while clothing items can be grouped into categories like menswear or womenswear.

The beauty of **OLAP systems** is the ability to view data at any level of granularity, from **fine-grained data** to **coarser-grained data**. With the power of these tools, data analysts can gain insights and understanding from even the most complex of data sets.

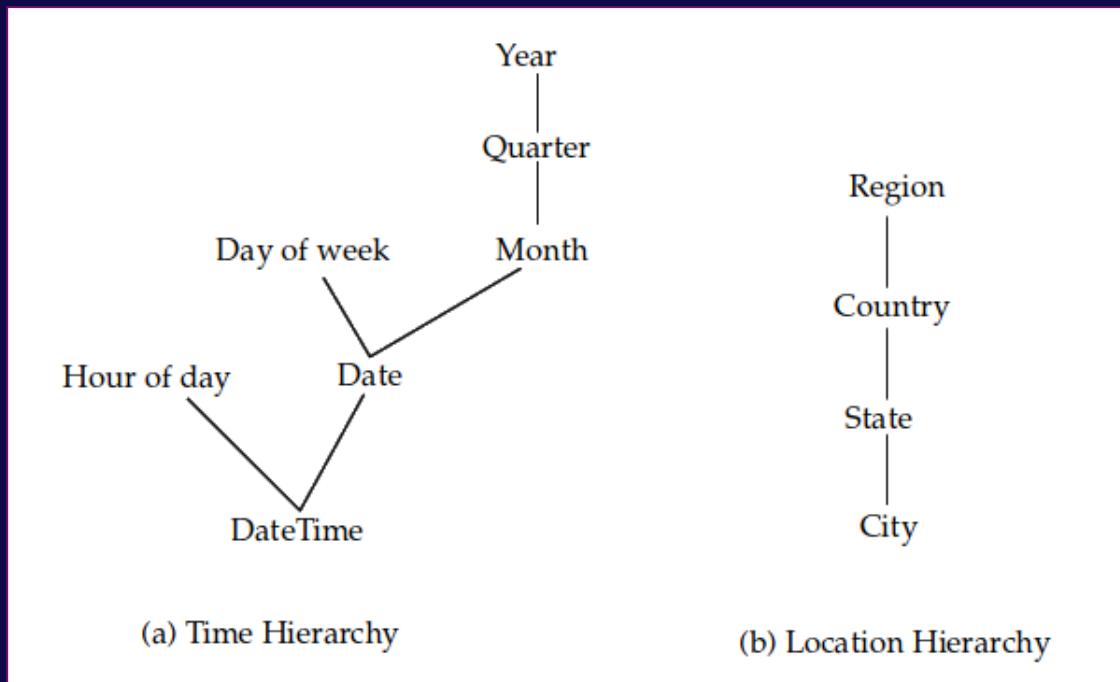


Figure 44 - Hierarchies on dimensions

In the realm of **database management and online analytical processing (OLAP)**, the intricacies of cross-tab and relational tables have been a topic of much discussion. A cross-tab, in particular, is an entity that differs from the typical relational tables stored in databases, as its columns are contingent on the data present within it. The malleability of its structure, while desirable for display purposes, poses a problem when it comes to data storage.

To address this issue, a cross-tab can be represented in a relational form with a fixed number of columns or by introducing a special value, "all," to denote subtotals.

clothes_size: all		category	item_name	color			total
				dark	pastel	white	
womenswear	skirt		skirt	8	8	10	53
	dress		dress	20	20	5	35
	subtotal		subtotal	28	28	15	88
menswear	pants		pants	14	14	28	49
	shirt		shirt	20	20	5	27
	subtotal		subtotal	34	34	33	76
	total			62	62	48	164

Figure 45 - Cross tabulation of sales with hierarchy on item name.

item.name	color	clothes.size	quantity
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	white	all	28
shirt	all	all	49
pants	dark	all	20
pants	pastel	all	2
pants	white	all	5
pants	all	all	27
all	dark	all	62
all	pastel	all	54
all	white	all	48
all	all	all	164

Figure 46 - Relational representation of the data

Furthermore, hierarchies can be represented in a relational format as well, such as the relationship between womenswear and menswear categories and their respective items. OLAP systems were first implemented using multidimensional arrays in memory, which led to the advent of multidimensional **OLAP (MOLAP)** systems. Subsequently, **OLAP** facilities were integrated into relational systems to form relational **OLAP (ROLAP)** systems. Hybrid systems, also known as hybrid **OLAP (HOLAP)** systems, arose from storing some summaries in memory and the remaining data in a relational database.

```
select * from sales pivot
( sum(quantity)  for color in ('dark','pastel','white') )
order by item name;
```

To compute data cubes, a **naïve** algorithm requires multiple scans of the relation, which can be optimized by computing an aggregation on a set of attributes from another aggregate with a larger set of attributes. The number of groupings on subsets of dimension attributes in data cubes grows exponentially, making pre-computation and storage of all possible groupings infeasible. Instead, a selected set of groupings can be precomputed and stored, while others can be computed on demand.

<i>item.name</i>	<i>clothes.size</i>	<i>dark</i>	<i>pastel</i>	<i>white</i>
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2

Figure 47 - Result of SQL pivot operation on the sales relation

<i>item_name</i>	<i>quantity</i>
skirt	53
dress	35
shirt	49
pants	27

Figure 48 - Query result

The nuances of cross-tab and relational tables, as well as OLAP systems, have been the focus of many studies in the realm of database management. Despite the challenges posed by the malleability of cross-tab structures and the exponential growth of groupings in data cubes, researchers have developed optimized methods to address these issues and enhance the efficiency of OLAP systems.

Online Analytical Processing (OLAP) continues to garner widespread attention. One notable implementation of OLAP is SQL, which boasts a pivot clause that enables the creation of cross-tabs. Supported by a variety of SQL implementations, such as Microsoft SQL Server and Oracle the pivot clause is especially handy for generating a pivot table.

The pivot clause enables the specification of values from an attribute, which are then displayed as attribute names in the pivoted result. It is noteworthy that while the pivot clause does not compute subtotals on its own, it can be used in combination with the group by construct to achieve the desired result.

SQL also supports generalizations of the group by constructing through the cube and rollup operations. The cube and rollup constructs allow for the execution of multiple groups by queries in a single query, with the result returned as a single relation.

Data cube relations can be incredibly large, as evidenced by the cube query, which generates **80** tuples for just **3** possible colors, **4** possible item names, and **3** sizes. To produce a more readable result, the relation of Figure uses “*all*” in place of null. To achieve the same result, the “*all*” value must be substituted for null in the query.

item_name	color	quantity
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5

Figure 49 – Data result

The concept of using the decode function in SQL queries has been explored in depth, with its ability to substitute values in an attribute of a tuple being particularly noteworthy.

By comparing a given value against a set of match values and corresponding replacement values, the decode function can effectively replace attribute values as needed. However, it does not function optimally when dealing with null values, as predicates on nulls ultimately result in false. To circumvent this issue, the grouping function can be employed, returning **1** if its argument is a **null value** generated by a cube or rollup, and **0** otherwise.

This allows for the substitution of all values in place of nulls.

```
select decode(grouping(item name), 1, 'all', item name) as item name  
      decode(grouping(color), 1, 'all', color)  
      as color sum(quantity) as quantity  
  from sales group by cube(item name, color);
```

In addition to the decode function, the rollup construct has also been discussed. Similar to the cube construct, rollup generates fewer groups by queries and can be useful for hierarchies when groups are of frequent practical interest.

```
select item name, color, clothes size,  
      sum(quantity) from sales  
  group by rollup(item name),  
          rollup(color, clothes size);
```

For example, a location hierarchy could be grouped by region, country, state, and city, allowing for a sequence of "*drilling down*" for further detail. Multiple rollups and cubes can also be used in a single group by clause, with the resulting groupings depending on the attributes used in each rollup or cube. While neither clause allows for complete control over the generated groupings, more restricted groupings can be generated using the grouping construct in the having clause.

2.5 Formal Relational Query Languages

In this latest chapter, we examine the very foundations upon which these powerful tools are built. In this illuminating chapter, we explore three formal languages that provide the theoretical framework for SQL and other relational query languages. The first is relational algebra, a powerful tool that underpins the SQL query language and enables it to manipulate and process vast amounts of data with ease. Following this, we delve into the tuple relational calculus and the domain relational calculus, two declarative query languages that are rooted in the realm of mathematical logic. Together, these three formal languages offer a deep understanding of the principles behind modern relational databases, making this chapter an essential resource for anyone looking to gain complete mastery of the subject.

The Relational Algebra

In this chapter, we delve into relational algebra, which serves as a procedural query language in the relational database model. With a set of operations that take one or two relations as input and produce a new relation as output, relational algebra offers a variety of useful tools for database querying. The fundamental operations in relational **algebra include select, project, union, set difference, Cartesian product, and rename**. Additionally, there are other operations available such as set intersection, natural join, and assignment.

course.id	prereq.id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 50 - The instructor relation

Unary operations such as select, project, and rename operate on one relation, while binary operations such as union, set difference, and Cartesian product operate on pairs of relations. The select operation, denoted by the lowercase **Greek letter sigma (S)**, selects tuples that satisfy a given predicate. In order to select tuples from the instructor relation where the instructor is in the "Physics" department, one would write dept name = "Physics" (*instructor*).

To find all instructors with a salary greater than \$90,000, one would write $\text{salary} > 90000$ (*instructor*). The selection predicate may include comparisons between two attributes, and multiple predicates can be combined with the connectives and (\wedge), or (\vee), and not (\neg).

Iteration Number	Tuples in c1
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Figure 51 - Result of $R \text{deptname} = \text{"Physics"}$ (*instructor*)

It should be noted that the term **selects** in relational algebra has a different meaning than in SQL, where it corresponds to what is referred to as **where** in SQL. This difference in interpretation may lead to confusion, and thus, it is important to differentiate between the two.

The project operation serves as a valuable tool to extract desired information from a relation. Specifically, the project operation allows for the creation of a new relation that retains only selected attributes of the original relation, effectively filtering out any unnecessary data. As a unary operation, projection is denoted by the uppercase **Greek letter pi (π)** and takes as its argument the relation to be projected. It is important to note that since a relation is a set, any duplicate rows will be eliminated from the resulting relation.

In the case of more complex queries, relational-algebra operations can be composed together to form a relational-algebra expression, akin to the composition of arithmetic operations. In this regard, the result of a relational-algebra operation is of the same type as its inputs, a relation, allowing for seamless integration of operations into a cohesive expression.

Another key operation in relational algebra is the union operation, represented by the **symbol U**. This operation is essential when combining two relations to form a larger relation that contains all the tuples in both relations, ensuring compatibility between the relations. However, it is crucial to ensure that the relations being combined have the same arity and domains to avoid confusion and errors.

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

Figure 52 - The section relation

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 53 - Courses offered in either Fall 2009, Spring 2010 or both semesters

In essence, the use of relational-algebra operations provides a powerful toolset for managing and analyzing data in a relational database, allowing for efficient data extraction and manipulation.

In the vast landscape of relational databases, two fundamental operations play a crucial role in combining and comparing data from different sources: the set-difference operation and the **Cartesian-product operation**. These operations, both part of the relational algebra, are essential tools in the arsenal of database professionals, enabling them to extract valuable insights and make informed decisions based on disparate sets of data.

The set-difference operation, denoted by the symbol “-”, allows us to find tuples that exist in one relation but not in another. By applying this operation, we can discover all courses taught during the Fall 2009 semester but not in the Spring 2010 semester. However, it is important to note that for this operation to be valid, the relations involved must have the same arity and identical domains for each attribute. These conditions ensure that the set difference operation produces meaningful and relevant results.

The **Cartesian-product operation**, denoted by the symbol “ \times ”, is another essential operation in the world of relational databases. This operation enables us to combine information from any two relations, making it possible to create powerful new datasets that contain valuable insights. However, because the same attribute name may appear in both relations, it is necessary to devise a naming scheme that distinguishes between these attributes. We can accomplish this by attaching the name of the relation from which the attribute originates. Moreover, it is essential to ensure that the relations used in this operation have distinct names to avoid any ambiguity.

<i>course.id</i>	<i>sec.id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room.number</i>	<i>time.slot.id</i>
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 54 - The *teaches* relation

The combination of these operations can be used to extract meaningful data from large datasets. For example, by applying the Cartesian-product operation to the instructor and teaches relations, we can find the names of all instructors in the Physics department, along with the course ID of all the courses they taught. While the **Cartesian-product operation** may produce some redundancy in the resulting dataset, it remains a powerful tool for database professionals seeking to extract useful information from large and complex datasets.

course.id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 55 - Result of $\text{instructor} \times \text{teaches}$

In the realm of relational algebra, a crucial task is to assign a name to the results of expressions, as unlike database relations, they lack a distinct identifier. Fortunately, the rename operator, symbolized by the lowercase **Greek letter rho**, offers a means to do just that. Given a relational algebra expression E , the operator $\text{rx}(E)$ generates the outcome of expression E under the name x . Interestingly, a relation r is also considered a trivial relational algebra expression, and thus the rename operation can be applied to r to obtain the same relation under a new name.

Another form of the rename operation involves an expression **E** with arity **n**.

In this case, **rx(A₁,A₂,...,A_n)(E)** yields the result of **E** under the name **x** and with the attributes renamed to **A₁, A₂,..., A_n**. An example of renaming a relation involves the query *"Find the highest salary in the university."* Our strategy involves computing a temporary relation of non-maximum salaries and then taking the set difference between the **rsalary(instructor)** relation and the temporary relation to obtaining the desired outcome.

<i>inst.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course.id</i>	<i>sec.id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Physics	95000	10101	cs-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	cs-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
<hr/>								
12121	Wu	Physics	95000	10101	cs-101	1	Fall	2009
12121	Wu	Physics	95000	10101	cs-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
<hr/>								
15151	Mozart	Physics	95000	10101	cs-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	cs-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
<hr/>								
22222	Einstein	Physics	95000	10101	cs-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	cs-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009

Figure 56 - Result of the subexpression

To compute the temporary relation, a mechanism to differentiate between two salary attributes is necessary. The rename operation comes to the rescue, enabling one reference to the instructor relation to be renamed, permitting the relation to be referenced twice without ambiguity. The temporary relation comprises salaries that are not the largest, and the resulting expression is given by:

```
Ninstructor.salary (Qinstructor.salary  
< d.salary (instructor × pd (instructor)))
```

To find the highest salary in the university, the expression:

```
Nsalary (instructor) = Ninstructor.salary  
(instructor.salary < d.salary (instructor × Rd (instructor)))
```

To construct a **general expression** in relational algebra, we utilize smaller sub-expressions. Two relational-algebra expressions, **E1** and **E2**, can be used to construct a range of relational-algebra expressions, including union, difference, cross-product, selection with a predicate, projection with a list of attributes, and renaming.

One of these new operations is a set intersection, denoted by the symbol "**∩**". By utilizing set intersection, we can write simpler queries for finding the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters, as well as for other similar cases.

```
Pcourse id (Qsemester = "Fall" ∧ year=2009 (section)) ∩  
Pcourse id (Qsemester = "Spring" ∧ year=2010 (section))
```

Another new operation is the **natural join**, denoted by the symbol "**⋈**". This operation allows us to combine certain selections and a **Cartesian product** into one operation. The natural-join operation forms a **Cartesian product** of its two arguments, performs selection-forcing equality on those attributes that appear in both relation schemas and removes duplicate attributes. The resulting relation is smaller and contains only those tuples that give information about an instructor and a course that the instructor actually teaches. The natural join is an essential operation that simplifies many queries involving **Cartesian products**.

The **relational-algebraic expression**, in all its elegance, can be made even more palatable through the use of the assignment operation. This operation, denoted by the symbol \leftarrow , enables the parts of the expression to be assigned to temporary relation variables for the sake of convenience. In a manner akin to assignment in a programming language, the result of the expression to the right of the \leftarrow is assigned to the relation variable on the left of the \leftarrow , with no relation being displayed to the user. The beauty of this approach lies in its ability to create a sequential program of assignments followed by an expression that displays the value of the query's result. It is important to note that assignments in relational algebra queries must always be made to a temporary relation variable, as assignments to permanent relations constitute a modification of the database. Though it does not add any additional power to the algebra, the assignment operation remains a valuable tool for expressing complex queries with ease.

- The outer-join operation proves to be a vital extension of the join operation by catering to instances of missing information. When tuples in either of the relations being joined are not part of the join's result due to non-matching, the outer join operation comes to the rescue. There are three variations of the outer join operation, namely left outer join, right outer join, and full outer join, each with a distinct approach to preserving lost tuples.
- The left outer join operation takes all the tuples in the left relation that do not have a matching tuple in the right relation, fills in the attributes from the right relation with null values, and adds these tuples to the natural join result. On the other hand, the right outer join operation pads tuples from the right relation with nulls and adds them to the result of the natural join when there is no match in the left relation. The full outer join operation performs both left and right outer joins, adding tuples from the left relation and the right relation that do not match the result of the join.

It is worth noting that outer-join operations have the potential to generate results containing null values. Thus, it becomes essential to specify how different relational-algebra operations treat null values. However, it is fascinating to observe that outer-join operations can be expressed by the basic relational-algebra operations. For example, the left outer join operation, $r \times s$, can be expressed as $(r \times s) \cup (r - PR(r \times s)) \times \{(null, \dots, null)\}$, where the constant relation $\{(null, \dots, null)\}$ is on the schema $S - R$.

Overall, the outer-join operation plays a pivotal role in preserving lost tuples and filling the gaps in the join operation.

In the realm of relational algebra operations, two extensions provide further functionality that is not possible with the basic set of operations. These are the generalized projection and aggregate operations, which offer the ability to incorporate **arithmetic** and **string functions**, as well as **aggregate functions** such as *min* and *average* respectively.

- The **generalized-projection operation** takes an algebraic expression **E** and a list of arithmetic expressions **F₁** through **F_n** that consist of constants and attributes of **E** and generates a new relationship that includes the projected attributes with any arithmetic operations applied to them. For example, the expression "*P*(*ID, name, dept name, salary ÷ 12(instructor)*)" would yield a new relation containing the *ID, name, department name, and monthly salary of each instructor*.
- **Aggregate operations**, on the other hand, take a collection of values and return a single value based on a given function. Common aggregate functions include sum, average, count, min, and max, and can be applied to multisets of values where duplicates are not eliminated, or to sets of values where duplicates are removed. The calligraphic **G** symbol denotes the use of aggregation, with a subscript indicating the specific function to be applied. For instance, "*Gsum(salary)(instructor)*" returns a relation with a single attribute containing the sum of all instructors' salaries.

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>	<i>course.id</i>	<i>sec.id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
33456	Gold	Physics	87000	null	null	null	null
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
58583	Califieri	History	62000	null	null	null	null
76543	Singh	Finance	80000	null	null	null	null
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 57 - Tuples of the instructor relation, grouped by the dept name attribute

These operations provide further power to the relational-algebra language, enabling more complex queries and analyses to be performed on relational databases.

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

Figure 58 - The result relation for the query "Find the average salary in each department"

The Tuple Relational Calculus

To express a query in the tuple relational calculus, we use the notation $\{t \mid P(t)\}$, which represents the set of all tuples t such that predicate P is true for t . By using $t[A]$ to denote the value of tuple t on attribute A , and $t \in r$ to denote that tuple t is in relation r , we can write queries such as "*Find the ID, name, dept name, salary for instructors whose salary is greater than \$80,000*" as $\{t \mid t \in \text{instructor} \wedge t[\text{salary}] > 80000\}$.

To express more complex queries, we need to use the "**there exists**" construct from mathematical logic. For instance, to find the instructor ID for each instructor with a salary greater than \$80,000, we use the notation $\{t \mid \exists s \in \text{instructor } (t[\text{ID}] = s[\text{ID}] \wedge s[\text{salary}] > 80000)\}$, which reads as "*The set of all tuples t such that there exists a tuple s in relation instructor for which the values of t and s for the ID attribute are equal, and the value of s for the salary attribute is greater than \$80,000.*"

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Figure 59 - Names of all instructors whose department is in the Watson building

Similarly, queries involving multiple relations can be expressed using multiple "there exists" clauses connected by the logical operator "and."

For example, to find the names of all instructors whose department is in the Watson building, we can use the notation $\{t \mid \exists s \in \text{instructor } (t[\text{name}] = s[\text{name}] \wedge \exists u \in \text{department } (u[\text{dept name}] = s[\text{dept name}] \wedge u[\text{building}] = "Watson"))\}$.

Finally, we can use the **logical operator** "or" to find the set of all courses taught in the *Fall 2009* semester, the *Spring 2010* semester, or both. If we want only those course id values for courses that are offered in both the *Fall 2009* and *Spring 2010* semesters, we simply change the "or" to "and" in the preceding expression.

With these powerful tools at our disposal, we can now tackle more **complex queries**, such as "*Find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester,*" using the "*not*" symbol to exclude the courses offered in both semesters.

The tuple relational calculus offers a concise and expressive language for querying relational databases, allowing us to uncover insights and patterns hidden in the data.

- **Tuple-relational calculus** is a powerful tool for expressing queries. However, it is not without its limitations. In order to ensure that queries are both meaningful and computable, we must impose certain constraints on their structure.
- At the heart of tuple-relational calculus is the concept of a *formula*, which, takes the form $\{t|P(t)\}$. Here, P is a formula consisting of atoms, which can be either relational expressions or comparisons involving tuple variables and constants. Tuple variables may be bound or free, depending on whether they are quantified by a logical operator.
- In order to avoid the problem of infinite relations, we must ensure that all expressions are "*safe*," meaning that their domain (*the set of values referenced by the formula*) is finite. This is accomplished by imposing restrictions on the structure of the formula, such as **disallowing expressions** that generate infinite relations.
- The expressive power of languages is a topic of great interest and significance. In particular, the tuple relational calculus, when restricted to safe expressions, has been found to be equally expressive as the basic relational algebra. This equivalence holds for all fundamental relational **algebraic operations, including union, difference, cartesian product, selection, and projection**. It is important to note that this result excludes extended relational operations, such as generalized projection and aggregation.
- Every **relational algebraic expression** that employs only the basic operations has an equivalent expression in the tuple relational calculus. Moreover, the converse is also true, and every **tuple-relational-calculus expression** has an equivalent **relational-algebra expression**. While the proof of this equivalence is beyond the scope of our present discussion.
- The tuple relational calculus does not have any direct equivalent of the **aggregate operation**, but it can be **extended to support aggregation**. Furthermore, extending the tuple relational calculus to handle arithmetic expressions is a straightforward matter.

The Domain Relational Calculus

Domain Relational Calculus, a prominent method for querying databases, diverges from its sibling, **Tuple Relational Calculus**, by utilizing domain variables that represent values from an attribute's domain rather than an entire tuple's values. Despite their differences, **Domain and Tuple Relational Calculus** are closely linked. The former acts as the theoretical foundation for the commonly employed QBE language, while the latter forms the basis of the widely utilized SQL language.

The domain relational calculus is an important tool for expressing queries. This form of calculus uses domain variables that take on values from an attribute's domain, rather than values for an entire tuple. To better understand this calculus, it's helpful to consider its formal definition.

- An atom is a formula. If **P1** is a formula, then so are $\neg P1$ and $(P1)$. If **P1** and **P2** are formulae, then so are **P1 v P2**, **P1 \wedge P2**, and **P1 \Rightarrow P2**. If **P1(x)** is formulaic **x**, where **x** is a free domain variable, then $\exists x (P1(x))$ and $\forall x (P1(x))$ are also formulae. It's worth noting that this calculus serves as the theoretical basis of the popular **QBE language**, just as relational algebra serves as the basis for SQL. By understanding the formal definition of the domain relational calculus and the rules for building up formulae, one can effectively query databases and retrieve the information they need.
- As we delve further into the world of database systems, we encounter a realm where queries reign supreme. In the pursuit of extracting information from databases, it is essential to understand the intricacies of the domain of relational calculus. This calculus, much like its tuple-based counterpart, involves a language for expressing queries over relations. In the pursuit of clarity, we provide examples of domain-relational-calculus queries that mirror those written in tuple calculus, highlighting their similarities.

However, we must note a crucial difference between these two forms of calculus. In tuple calculus, we can readily bind a variable to a relation using the symbol " \in ". In contrast, domain calculus does not bind variables to tuples but rather to domain values. This subtlety is significant because the domain of a variable is unconstrained until the subformula " \in instructor" restricts it to a specific domain.

3 DATABASE DESIGN

In a world where information is king, the design and management of database systems are critical to the success of modern enterprises. These systems handle vast amounts of data that are intimately intertwined with the operations of the organization, whether it be for the production of information products or the support of essential devices and services.

The first two chapters of this section delve into the essential building blocks of database schema design. Specifically, the chapter explores the **entity-relationship (E-R) model**, a high-level data model that identifies and organizes basic objects, or entities, and the relationships between them. This model represents a crucial first step toward developing a comprehensive and effective database schema.

However, relational database design - the creation of a relational schema - is equally important and often more complex. The chapter introduces several "*normal forms*" that help distinguish well-designed schemas from those that may be prone to inconsistencies and inefficiencies. These forms provide a framework for optimizing query performance while maintaining data integrity.

To design a complete and effective database application environment that meets the needs of a modern enterprise, a host of additional issues must be addressed. The chapter delves into some of the most critical of these topics, including the **design of user interfaces** and **multi-layered system architectures**, which form the foundation for **large-scale applications**. Additionally, the chapter provides a comprehensive exploration of security at both the application and database levels, ensuring that sensitive data is protected from unauthorized access or compromise.

3.1 Database Design and the E-R Model

As we delve deeper into the intricacies of database systems, it becomes imperative to comprehend the nuances of database schema design. In order to effectively capture the essence of the database and its interrelated components, it is crucial to have a solid understanding of the **entity-relationship (E-R) model**. This model is a potent tool for identifying the entities that are part of the database and the relationships that bind them together.

This chapter elucidates the key concepts of the **E-R model**, highlighting its significance in relation to a robust database schema design. The crux of the matter is to transform the **E-R design** into a set of relation schemas and to capture the constraints that govern those relations. The next chapter deals with the analysis of the set of relation schemas, using a broader range of constraints to determine whether it is a good or bad database design.

It is imperative that one has a solid understanding of the fundamental principles of database design. These two chapters not only provide a comprehensive overview of the process but also equip the reader with the necessary knowledge to create a robust and effective database schema design.

Overview of the Design Process

In the complex and multifaceted task of creating a database application, the needs of users take center stage. This entails designing the database schema, as well as the programs that access and update the data, and a security scheme to regulate access. In this chapter, the focus is on the design of the database schema, while also briefly outlining some of the other tasks that lie ahead.

To design a complete database application environment that meets the needs of the enterprise being modeled, one must be attuned to a wide range of issues. These issues have a bearing on various design choices at the *physical*, *logical*, and *view levels*. Therefore, attention must be paid to these aspects of the expected use of the database in order to create an effective and comprehensive design.

The process of designing a database application is an intricate and multifaceted task that involves various phases. While creating a **small-scale application** may allow for a straightforward approach to designing the database schema, real-world applications are far more complex and necessitate extensive interaction between the database designer and application users to understand and represent the data requirements of the enterprise being modeled. The first phase of database design involves characterizing the data needs of the prospective database users, which entails interacting extensively with domain experts and users to obtain a comprehensive specification of user requirements.

Once the user requirements are established, the designer then selects a data model and translates these requirements into a conceptual schema of the database. The entity-relationship model is a common method for representing the conceptual design, which specifies the entities, attributes, relationships, and constraints of the database. At this stage, the designer's focus is on describing the data and their relationships, rather than specifying physical storage details.

The conceptual schema also indicates the functional requirements of the enterprise, which describe the operations or transactions that will be performed on the data. The next phase involves mapping the **high-level conceptual data model** to a **logical data model**, typically the relational data model, and mapping the conceptual schema defined using the entity-relationship model into a relation schema. Finally, the resulting system-specific database schema is used in the physical-design phase, which specifies the physical features of the database, including the form of file organization and choice of index structures.

It is crucial to design the database schema with care, as changes to the logical schema may affect a number of queries and updates scattered across application code. The physical schema can be changed relatively easily after an application has been built, but logical schema changes require greater effort and should be avoided if possible. Therefore, the database design phase should be conducted thoroughly before building the rest of the database application to ensure that all data requirements are satisfied and redundant features are removed.

The art of designing a database schema is not to be underestimated, as it requires careful consideration of the many entities and relationships within an enterprise. These entities must be identified and represented in the design, with each one related to the others in a way that captures the complexity of the enterprise. However, in doing so, we must be wary of two major pitfalls that can lead to poor design: redundancy and incompleteness.

Redundancy occurs when information is needlessly repeated, leading to potential inconsistencies if updates are not made uniformly. For example, if we store the course identifier and title with each course offering, we redundantly store the title with every offering. Instead, it suffices to store only the identifier with each offering and associate the title with the identifier in a course entity. Incompleteness, on the other hand, occurs when certain aspects of the enterprise cannot be modeled due to inadequate design. If we only have entities corresponding to course offerings and not to courses themselves, we cannot represent information about a new course unless it is already being offered. This leads to problematic design and workarounds that are often unattractive and impractical.

Avoiding bad designs is just the beginning, however. There may be a plethora of good designs to choose from, each with its own set of trade-offs. For instance, when a customer buys a product, is the sale itself a relationship between the customer and the product, or is the sale an entity that relates to both the customer and the product? The choice made here can have important implications for the overall design.

Ultimately, database design is a challenging problem that requires both scientific and artistic sensibilities. It is a task that demands a deep understanding of the enterprise being modeled, as well as an appreciation for the elegance and simplicity of good design.

The Entity-Relationship Model

The Entity-Relationship (E-R) data model is a powerful tool in the field of database design. Its purpose is to provide a clear and concise representation of the logical structure of a database by mapping real-world entities and relationships onto a conceptual schema. This has made it a widely-used and popular approach for database design.

- The E-R data model is based on three fundamental concepts: *entity sets*, *relationship sets*, and *attributes*. These concepts allow for the creation of a comprehensive schema that accurately represents the interactions and meanings of an enterprise. In addition, the E-R model has a graphical representation, the E-R diagram, which helps designers visualize the structure of the database.
- The E-R data model has revolutionized the field of database design, providing a powerful framework for accurately representing complex relationships and entities in a logical and efficient manner. Its widespread use is a testament to its effectiveness and usefulness in the field.
- The entity-relationship (E-R) data model is an invaluable tool for creating a conceptual schema that accurately maps the interactions of real-world enterprises. Comprised of three core concepts, namely entity sets, relationship sets, and attributes, the E-R model is a powerful framework for designing databases. In particular, entity sets represent distinct "*things*" or "*objects*" in the real world that can be identified by a set of properties, or attributes. Whether concrete (*e.g. people or books*) or abstract (*e.g. courses or reservations*), entity sets share the same attributes and are grouped into collections, or extensions.

For instance, a university database may include the entity sets of instructors and students, with attributes such as ID, name, department, and salary for instructors, and ID, name, major, and GPA for students. These entity sets need not be disjoint; a person can be both an instructor and a student, or neither. Attributes describe the properties possessed by each member of an entity set, with each entity having its own unique value for each attribute. The ID attribute is typically used to uniquely identify entities within a set, and databases consist of a collection of entity sets that may include dozens of distinct collections.

Ultimately, the entity-relationship model is a powerful tool for conceptualizing complex databases and designing schema that accurately reflects the interactions of real-world enterprises. Through the use of entity sets, attributes, and relationship sets, database designers can create an intuitive and robust data model that captures the essence of complex data structures.

The figure consists of two separate tables side-by-side. The left table is labeled "instructor" and contains the following data:

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

The right table is labeled "student" and contains the following data:

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

Figure 60 - Entity sets *instructor* and *student*

The notion of a relationship takes center stage. A relationship is a connection between several entities that helps to make sense of the data being modeled. At its core, a relationship is a mathematical relation on $n \geq 2$ (*possibly nondistinct*) entity sets. When multiple relationships of the same type exist, they can be grouped together as a relationship set.

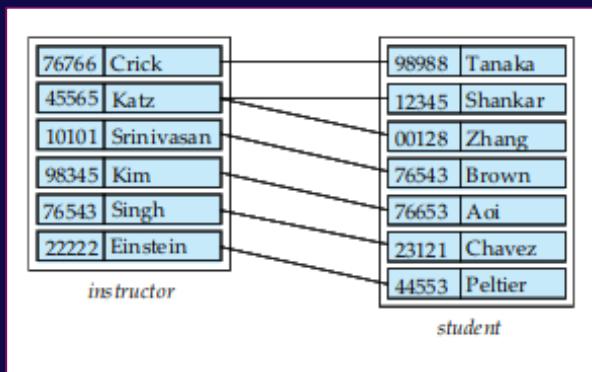


Figure 61 - Relationship set *advisor*

Consider, for example, the two entity sets of instructor and student, which can be linked together through a relationship set called *advisor*. This relationship set captures the connection between a specific instructor and a specific student, thus formalizing the advisory role of the instructor to the student. Similarly, a relationship set can link the entity sets of students and sections to represent the enrollment of students in specific course sections.

The concept of participation plays a critical role in relationship sets. This refers to the involvement of entity sets in the relationship set. Each entity set plays a role in the relationship, which helps clarify the meaning of the relationship. In cases where the entity sets in the relationship set are not distinct, explicit role names are necessary to specify how the entity participates in a relationship instance.

Descriptive attributes can also be associated with relationships, such as the date an instructor became an advisor to a student. A relationship instance must be uniquely identifiable from its participating entities, without the use of descriptive attributes.

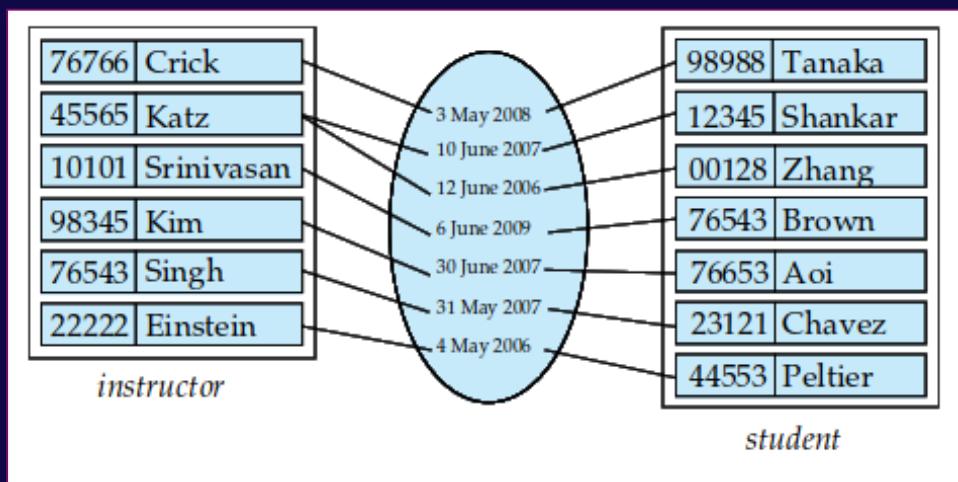


Figure 62 - Date as attribute of the advisor relationship set

In many cases, multiple relationship sets involving the same entity sets can exist. Binary relationship sets are the most common, but there are cases where a relationship set may involve more than two entity sets. The modeling of relationships in database systems is a complex task, but a crucial one for creating accurate and useful databases.

In the realm of database design, the attributes of an entity set play a crucial role in representing real-world objects and their characteristics. In the context of the E-R model, each attribute is associated with a domain of permitted values, which defines the set of valid values for that attribute. For example, the domain of the "course id" attribute might be a set of text strings of a particular length, while the domain of the "semester" attribute might be a set of strings from the set **{Fall, Winter, Spring, Summer}**.

In the E-R model, an attribute is a function that maps from an entity set into a domain, allowing each entity to be described by a set of *(attribute, data value)* pairs. These pairs represent the values of each attribute for a given entity, providing a significant portion of the data stored in the database.

Attributes in the E-R model can be characterized by several types, including simple and composite attributes, single-valued and multivalued attributes, and derived attributes. Simple attributes represent indivisible, atomic values, while composite attributes can be divided into subparts. **Single-valued attributes** have a single value for each entity, while multivalued attributes can have a set of values for a specific entity. Derived attributes can be derived from other attributes or entities, with the value being computed when required.

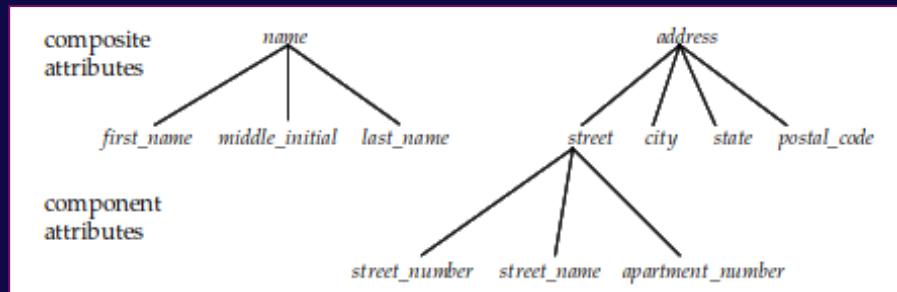


Figure 63 - Composite attributes instructor name and address

An attribute can also take a null value, indicating that the entity does not have a value for that attribute. Null values can signify “*not applicable*” or “*unknown*,” and they play an important role in handling missing or incomplete data.

Overall, the attributes of an entity set provide a rich and nuanced representation of the characteristics of real-world objects, enabling powerful database modeling and analysis capabilities.

Constraints

Constraints play a crucial role in ensuring data integrity and consistency. An essential aspect of constraints is mapping cardinalities, which express the relationship between entities in a database. While mapping cardinalities are most commonly used for binary relationship sets, they can also apply to more complex relationship sets involving multiple entity sets.

For a binary relationship set, the mapping cardinality can be classified as **one-to-one**, **one-to-many**, **many-to-one**, or **many-to-many**. The choice of mapping cardinality depends on the specific real-world situation that the relationship set represents. Take, for example, the advisor relationship set in a university. If a student can only be advised by one instructor, and an instructor can advise several students, then the relationship set from instructor to student is **one-to-many**. On the other hand, if a student can be advised by multiple instructors, as is the case with joint advising, the relationship set becomes **many-to-many**.

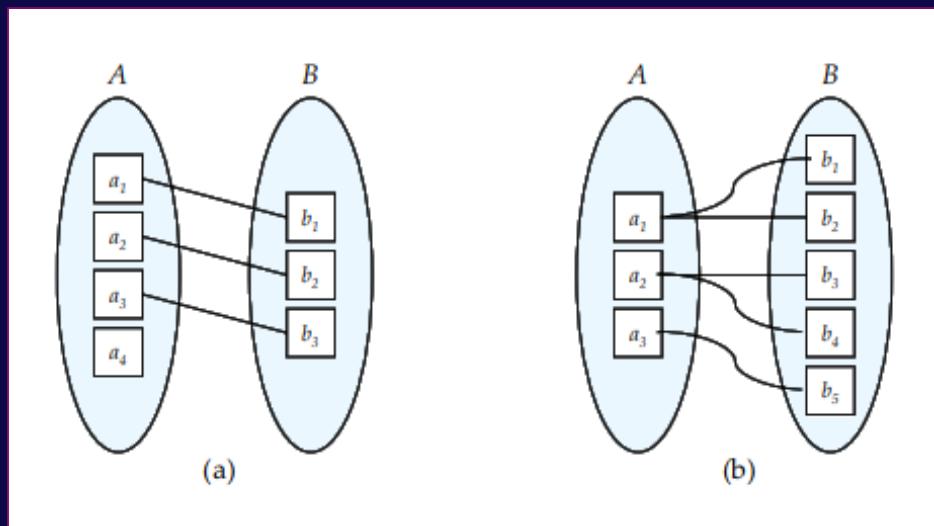


Figure 64 - Mapping cardinalities. (a) One-to-one. (b) One-to-many

Mapping cardinalities are a powerful tool for designing databases that accurately reflect the relationships between entities in the real world. By understanding mapping cardinalities, designers can ensure that their databases maintain data integrity and consistency, allowing for more accurate and efficient data processing.

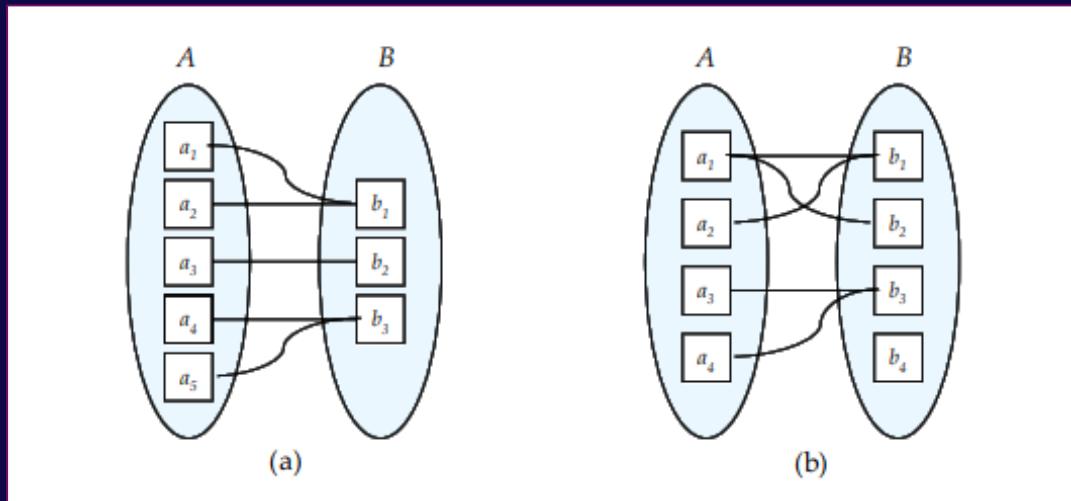


Figure 65 - Mapping cardinalities. (a) Many-to-one. (b) Many-to-many

The art of database design involves more than just defining entities and their relationships. One key aspect is setting constraints to ensure that the contents of the database adhere to certain standards. In this vein, mapping cardinalities and participation constraints serve as powerful tools to govern relationships among entity sets.

Mapping cardinalities are ratios that describe how many entities can be associated with another entity through a relationship set. While binary relationship sets are the most amenable to mapping cardinalities, they can also be applied to more complex sets involving more than two entity sets. For binary relationship sets, four mapping cardinalities exist **one-to-one**, **one-to-many**, **many-to-one**, and **many-to-many**. The choice of which mapping cardinality to use depends on the particulars of the real-world scenario being modeled.

Participation constraints, on the other hand, refer to the degree to which an entity set participates in a relationship set. If every entity in an entity set participates in at least one relationship in a relationship set, the participation is total. If only some entities in the entity set participate, the participation is partial. For instance, in the advisor relationship set, we would expect every student entity to have at least one corresponding instructor entity, making the participation total for students. In contrast, instructors need not advise any students, rendering participation partial for instructors.

By setting participation constraints, database designers can ensure **data consistency** and **reduce the risk of data corruption**.

The ability to distinguish entities from one another is of paramount importance. Entities may seem distinct on a conceptual level, but from a database perspective, it is crucial to express their differences in terms of their attributes. To this end, we require a set of attribute values that can uniquely identify each entity. Thus, the notion of a key, as previously defined for relation schemas, applies to entity sets as well.

A key for an entity is a set of attributes that can distinguish one entity from another. The concepts of superkey, candidate key, and primary key, which are fundamental to relation schemas, are also applicable to entity sets. Keys are essential to identifying relationships uniquely, and thereby distinguishing one relationship from another.

For instance, let us consider a relationship set **R**, involving entity sets **E₁, E₂,..., E_n**. If **primary-key(E_i)** denotes the set of attributes that form the primary key for entity set **E_i**, then the composition of the primary key for relationship set R is determined by the set of attributes associated with **R**. If **R** has no attributes associated with it, then the set of attributes **primary-key(E₁) ∪ primary-key(E₂) ∪... ∪ primary-key(E_n)** describes a unique relationship in set **R**.

On the other hand, if **R** has attributes **a₁, a₂,..., a_m** associated with it, then the set of attributes **primary-key(E₁) ∪ primary-key(E₂) ∪... ∪ primary-key(E_n) ∪ {a₁, a₂,..., a_m}** describes an individual relationship in set R. In both of the above cases, the set of attributes **primary-key(E₁) ∪ primary-key(E₂) ∪... ∪ primary-key(E_n)** forms a superkey for the relationship set.

However, if the attribute names of primary keys are not unique across entity sets, they are renamed to distinguish them. If an entity set participates more than once in a relationship set, the role name is used instead of the name of the entity set to form a unique attribute name.

The structure of the primary key for the relationship set is determined by the mapping cardinality of the relationship set. The primary key of advisor, a relationship set involving entity sets instructor and student with attribute date, is the union of the primary keys of instructor and student if the relationship set is **many-to-many**. If the relationship is **many-to-one** from student to instructor, then the primary key of the advisor is simply the primary key of the student. Similarly, if an instructor can advise only one student, then the primary key of the advisor is simply the primary key of the instructor. For **one-to-one relationships**, either **candidate key** can be used as the **primary key**.

While the choice of the primary key for **non-binary relationships** is straightforward when no cardinality constraints are present, it becomes more complicated in the presence of such constraints. However, as we have not yet discussed how to specify cardinality constraints on **non-binary relationships**, we defer this discussion to later chapters.

Removing Redundant Attributes in Entity Sets

In designing a database using the **E-R model**, selecting the appropriate entity sets and their corresponding attributes is crucial. The **choice of attributes** lies in the hands of the designer, who possesses a good understanding of the structure of the enterprise. However, the relationship sets among various entities may result in redundant attributes, which should be removed from the original entity sets.

For instance, consider the entity sets instructor and department. Both contain the attribute dept name, with the dept name serving as the primary key for the department entity set. Therefore, the dept name is redundant in the instructor entity set and should be removed. While this may appear counterintuitive, treating the connection between instructors and departments uniformly as a relationship makes the **logical relationship explicit** and **avoids premature assumptions**.

Similarly, the student entity set is related to the department entity set through the relationship set student dept, eliminating the need for a dept name attribute in the student entity set. Another example involves the entity sets section and time slot, which share the attribute time slot id. Since time slot id is the primary key for the entity set time slot, it is redundant in the entity set section and should be removed.

Finally, redundant attributes can also exist within the same entity set. For instance, the entity set section contains the attributes building and room number, which are also present in the related entity set classroom. Thus, these attributes are redundant and should be removed from the section entity set.

By avoiding redundant attributes in **entity-relationship design**, we can create a streamlined and efficient database. For the university example discussed in this article, a comprehensive list of entity sets and relationship sets has been provided, showcasing a well-designed database free of redundant attributes.

Entity-Relationship Diagrams

Entity-Relationship Diagrams (ERDs) are graphical representations of the logical structure of a database that utilize simple and clear components to achieve widespread usage. ERDs consist of entity sets, relationship sets, and attributes, which are linked together by lines and arrows to specify the nature of the relationships between entities.

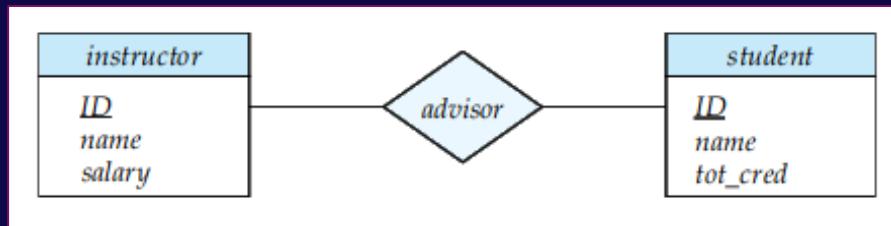


Figure 66 - E-R diagram corresponding to instructors and students

By mapping cardinality, **ERDs** can further indicate the number of times each entity participates in relationships within a relationship set. It is important to note that the placement of arrows in a relationship set indicates the nature of the relationship between the entity sets. **One-to-one**, **one-to-many**, **many-to-one**, or **many-to-many** relationships can be represented with either a directed or undirected line, depending on the cardinality.

ERDs can also specify minimum and maximum cardinalities, with minimum values of 1 indicating total participation, maximum values of 1 indicating at most one relationship, and maximum values of * indicating no limit.

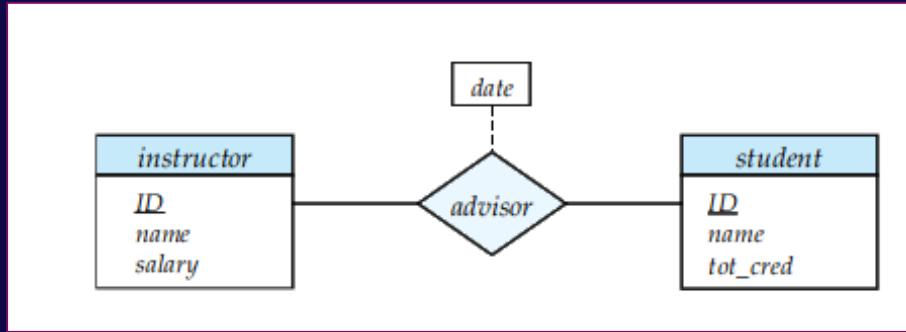


Figure 67 - E-R diagram with an attribute attached to a relationship set

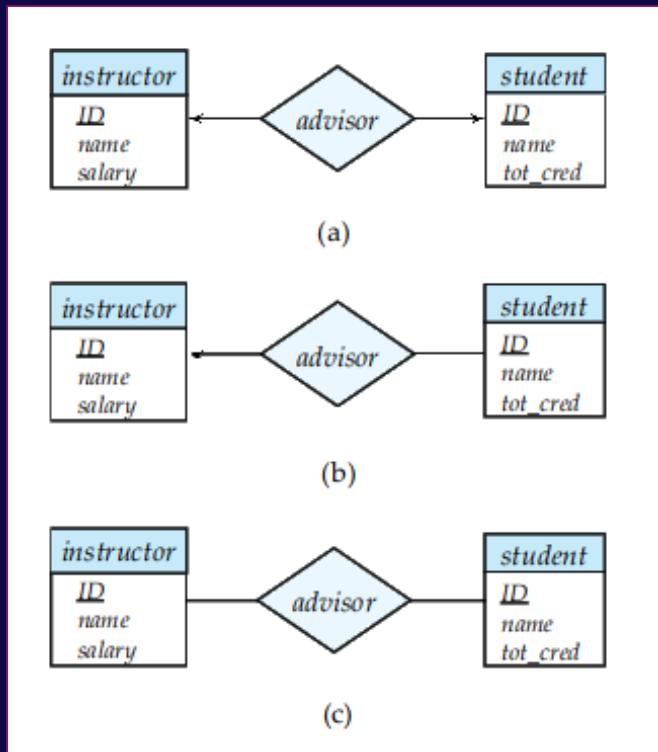


Figure 68 - Relationships. (a) One-to-one. (b) One-to-many. (c) Many-to-many

Overall, **ERDs** provide a useful tool for visualizing and designing complex databases.

To aid in understanding this concept, a visual representation of composite attributes in the **E-R notation** is. Here, the attribute "*instructor*" is replaced by the composite attribute "*name*", which consists of three component attributes: first name, middle initial, and last name.

Another example is given in the case of adding an address attribute to the instructor entity set. The composite attribute "*address*" is defined, as consisting of four component attributes: *street*, *city*, *state*, and *zip code*. Further complexity arises as the street attribute itself is a composite attribute, comprised of three component attributes: *street number*, *street name*, and *apartment number*.

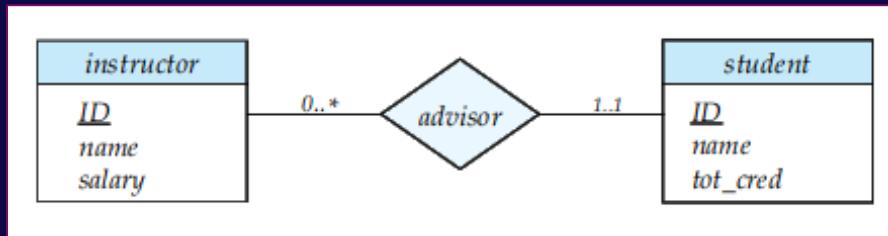


Figure 69 - Cardinality limits on relationship sets

As if this wasn't enough, the figure above also depicts two other attribute types: multivalued and derived attributes. The former, denoted by "**{phone number}**", indicates an attribute that can have multiple values for a single entity.

The latter, represented by “**age()**”, is an attribute that is not stored in the database but instead calculated from other attributes.

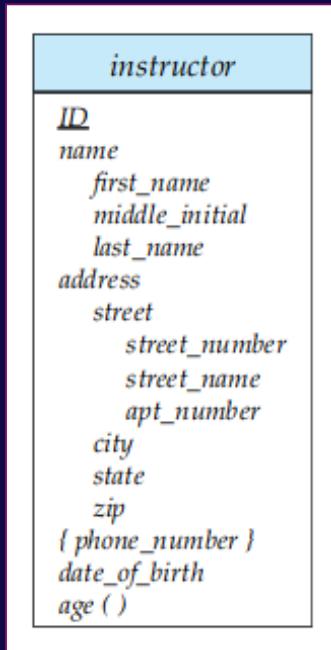


Figure 70 - E-R diagram with composite, multivalued, and derived attributes

In **E-R diagrams**, roles can be identified by marking the lines connecting diamonds to rectangles. The image presented in the figure above demonstrates this approach, where the course entity set and the prereq relationship set are linked by role indicators such as course id and prereq id.

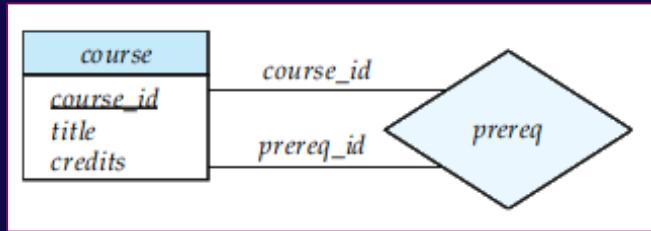


Figure 71 - E-R diagram with role indicators

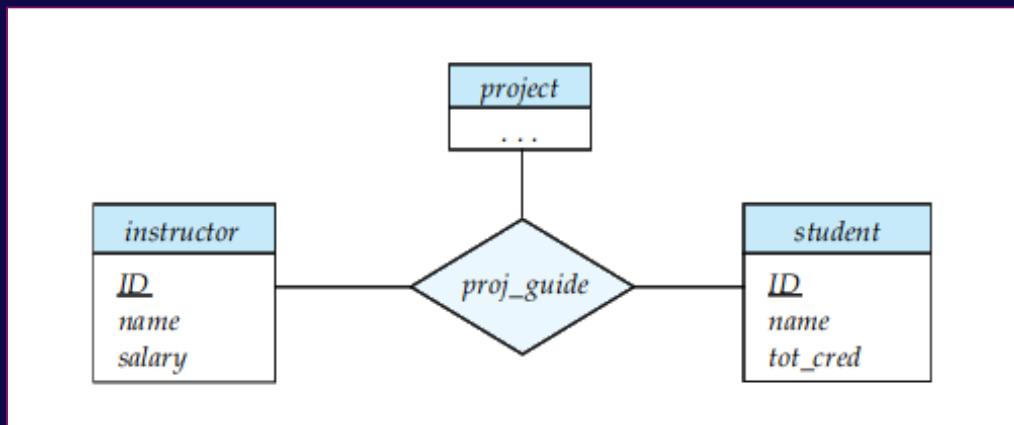


Figure 72 - E-R diagram with a ternary relationship

In the realm of **entity-relationship (E-R)** diagrams, roles can be designated by the labels attached to lines linking diamonds to rectangles. This is illustrated in the figure above, where the roles of course id and prereq id are indicated by labels on the line between the course entity set and the prereq relationship set.

Nonbinary relationship sets, which involve more than two entity sets, can also be depicted in **E-R diagrams**. The figure showcases an example involving three entity sets: *instructor*, *student*, and *project*. These sets are interconnected through the proj guide relationship set.

The concept of weak entity sets stands out as a particularly nuanced and intricate topic, one that requires a keen eye for detail and a deep understanding of the interrelationships between data entities. At its core, a weak entity set is an entity set that lacks the necessary attributes to form a **primary key**, and as such, it must be associated with another entity set, known as the identifying or owner entity set, to derive its unique identity.

To fully appreciate the complexity of this concept, we must delve deeper into the inner workings of weak entity sets. Take, for example, the case of a section entity, which is uniquely identified by a course identifier, semester, year, and section identifier. At first glance, section entities appear to be related to course entities, and so we create a relationship set between section and course entity sets.

However, upon closer inspection, we find that the information in this relationship set is redundant, as the section already has an attribute that identifies the course with which it is related. To address this redundancy, we could simply remove the relationship, but doing so would result in an implicit relationship between section and course, which is not desirable.

Instead, we must find a way to store the necessary information in a more efficient and meaningful way. One option is to remove the **course ID attribute** from the section entity and only store the remaining attributes. However, this approach presents a new problem, as the section entity would no longer have enough attributes to identify a specific section entity uniquely.

To solve this issue, we treat the relationship between section and course as a special relationship that provides the extra information required to identify section entities uniquely. This is where the concept of the weak entity set comes into play.

A weak entity set is an entity set that lacks sufficient attributes to form a **primary key** and must be associated with an identifying entity set to derive its **unique identity**. The identifying entity set is said to own the weak entity set that it identifies, and the relationship associating the two entity sets is known as the **identifying relationship**.

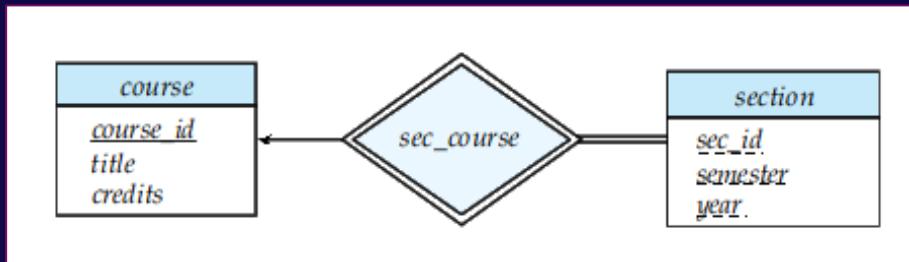


Figure 72 - E-R diagram with a weak entity set

In the case of section and course, the course is the identifying entity set for section, and the relationship between the two, known as sec course, is the **identifying relationship**. The discriminator of a weak entity set is a set of attributes that allows us to distinguish among all the entities in the weak entity set that depend on one particular strong entity.

In the case of the section, the discriminator consists of the attributes *sec id*, *year*, and *semester*. The **primary key** of a weak entity set is formed by the primary key of the **identifying entity set** plus the **weak entity set's discriminator**.

As we can see, the concept of weak entity sets is a nuanced and complex topic that requires a deep understanding of database design principles. However, by mastering this concept, we can create more efficient and meaningful database structures that better represent the relationships between data entities.

In a stunning demonstration of conceptual modeling, the authors present an **E-R diagram** that encapsulates the intricate workings of a university enterprise. This diagram, which is equivalent to the textual description of the university **E-R model** presented earlier in the text, is enriched with several additional constraints that underscore the complexity of this domain.

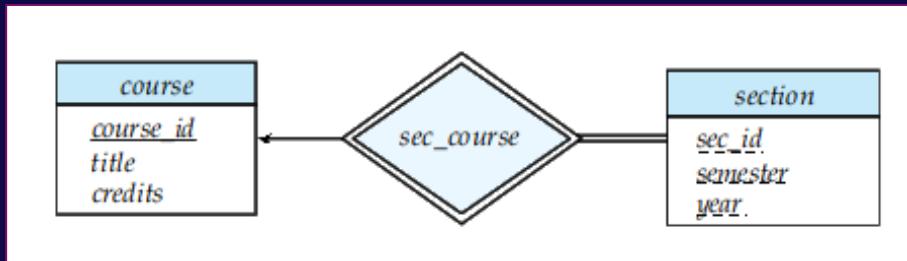


Figure 73 - E-R diagram for a university enterprise

Of particular note is the constraint that mandates each instructor to have exactly one associated department, which is represented by a double line between instructor and inst dept, indicating total participation of instructor in inst dept. Moreover, the diagram reveals that each instructor can have at most one associated department, as depicted by the arrow from inst dept to department.

Similarly, the course and student entities have double lines to *relationship sets course dept* and *stud dept*, respectively, while the section is now a weak entity set, with attributes *sec id*, *semester*, and *year* forming the discriminator. In a masterful stroke of design, the sec course is the identifying relationship set that relates the weak entity set section to the strong entity set course.

Notably, the relationship set takes has a descriptive attribute grade, and each student has at most one advisor. The authors tantalize readers with the promise that, they will unveil how this **E-R diagram** can be harnessed to derive the various relation schemas used in the enterprise. Truly, a tour de force of conceptual modeling that leaves readers eagerly anticipating what lies ahead.

Reduction to Relational Schemas

In the world of database design, the **E-R model** and the relational database model stand out as the most significant and widely used paradigms. While both models are abstract and logical representations of real-world enterprises, they rely on similar design principles, allowing for seamless conversion between them.

The **E-R model** uses entity sets and relationship sets to define the database structure, while the relational model uses relation schemas, which can be viewed as tables in a database. The conversion of **E-R design** to relational design requires that for each entity set and relationship set, there is a corresponding relation schema with a unique name.

Constraints are an essential aspect of database design, and the **E-R model** provides a framework for defining constraints that can be mapped to the relational model. In this regard, this section delves into how **E-R schemas** can be represented by relation schemas and how constraints from the **E-R design** can be mapped to constraints on relation schemas.

As such, the conversion of **E-R models** to relational models enables the use of a more straightforward and flexible approach to database design and management. This technique enhances data integrity, accuracy, and consistency, making it an invaluable tool for modern-day data scientists.

In the world of database management, the conversion of **E-R designs** into relational designs is a crucial step toward creating efficient and effective database systems. This process involves representing an **E-R schema** by relation schemas and mapping constraints from the **E-R model** to those in the relational model.

In this vein, let us explore how strong entity sets with simple descriptive attributes can be represented as relation schemas. If we have a strong entity set **E** with attributes **a1**, **a2**, ..., and **an**, we can create a schema called **E** with **n** distinct attributes to represent it. Here, since student ID is the primary key of the entity set, it is also the primary key of the relation schema.

All the strong entity sets, except the time slot, only have simple attributes. Thus, we can derive the following schemas from these strong entity sets: **classroom** (*building, room number, capacity*), **department** (*dept name, building, budget*), **course** (*course id, title, credits*), **instructor** (*ID, name, salary*), and **student** (*ID, name, tot cred*).

Note that the instructor and student schemas differ from those used in previous chapters as they do not contain the attribute dept name. However, we will revisit this issue shortly.

The handling of strong entity sets with nonsimple attributes can be a challenge. We create separate attributes for each component attribute of a composite attribute, rather than creating a separate attribute for the composite attribute itself. For example, the schema generated for the instructor entity set includes separate attributes for first name, middle name, and last name, rather than creating a separate attribute for the name.

Though not explicitly represented in the relational data model, they can be represented as “*methods*” in other data models such as the **object-relational data model**. And for multivalued attributes, we create relation schemas with an attribute corresponding to the **primary key** of the entity set or relationship set of which the multivalued attribute is a part.

In practice, this means that we create a relational schema for the **multivalued attribute phone number** of the instructor entity set, with the **primary key** of the instructor as the corresponding attribute. Each phone number of an instructor is then represented as a unique tuple in relation to this schema. And we create a **foreign-key constraint** on the relation schema generated from the multivalued attribute, with the attribute generated from the **primary key** of the entity set referencing the relation generated from the entity set.

We can drop the relation schema for the entity set and retain only the relation schema with the attribute corresponding to the **primary-key attribute** and the **multivalued attribute**. For example, the time slot entity set can be represented by a single schema created from the multivalued composite attribute, with time slot id as the **primary key**.

Drawing upon an encyclopedic knowledge of database design, the author skillfully articulates the mechanics of representing weak entity sets by way of a **relation schema**, complete with a **foreign-key** constraint that ensures the existence of a corresponding tuple representing the associated strong entity. The author then takes us through an illustration of this process by way of the weak entity set section in the **E-R diagram**, painting a vivid picture of how the **primary key** of the course entity set and the discriminator of the section combine to form the primary key of the schema, while also creating a **foreign-key constraint** on the section schema.

Moving on to the representation of relationship sets, the author masterfully explains the mechanics of choosing a primary key for binary relationship sets, deftly navigating the complexities of one-to-one, many-to-one, and one-to-many relationship sets to arrive at a lucid and insightful conclusion. Through a series of carefully crafted examples, the author walks us through the intricacies of creating foreign-key constraints on relation schema R, deftly leveraging the primary keys of entity sets to create a coherent and well-organized system.

Redundancy can be a hindrance rather than a help. In particular, relationships between weak and strong entity sets can lead to the superfluous schema, as the relationship sets themselves contain no descriptive attributes. This is further complicated by the fact that the primary key of a weak entity set includes the primary key of the strong entity set.

A striking example of this phenomenon can be seen in the **E-R diagram**, where the weak entity set section is dependent on the strong entity set course through the relationship set sec course. The sec course schema, which includes attributes such as *course id*, *sec id*, *semester*, **and** *year*, is redundant when compared to the section schema, which contains the same attributes among others. In fact, every combination of (*course id*, *sec id*, *semester*, *year*) in a sec course relation can also be found in the section schema, and vice versa. As such, the relationship set schema between weak and strong entity sets is generally unnecessary.

However, in cases where a **many-to-one relationship** set exists between entity sets **A** and **B**, and the participation of **A** in the relationship is total, we can combine the **A** and **AB** schemas to form a single schema.

For example, the schemas *instructor* and *department* correspond to the entity sets **A** and **B**, respectively, and can be combined to form the *inst dept* schema.

Similarly, the schemas *student* and *department* can be combined to form the *stud dept* schema, while the schemas *course* and *department* can be combined to form the *course dept* schema.

Entity-Relationship Design Issues

The notions of entity sets and relationship sets are fraught with imprecision. Indeed, the possibilities for defining a set of entities and the relationships that bind them are manifold. In this section, we delve into the fundamental issues that undergird the construction of an **entity-relationship (E-R)** database schema, with a more detailed exploration of the design process.

One salient question that arises is whether to utilize entity sets or attributes. Take, for instance, the instructor entity set, which could be augmented with an additional attribute - a *phone number*. However, it is entirely plausible to argue that a phone number should be treated as an entity in its own right, with its own corresponding attributes such as **location** (e.g. *office*, *home*) and **type** (e.g. *mobile*, *IP phone*).

In this scenario, the attribute of a phone number would not be added to the instructor entity set, but rather a new entity set of phones would be created, along with a relationship set between instructors and phones, aptly named "*inst phone*," as seen in Figure below:

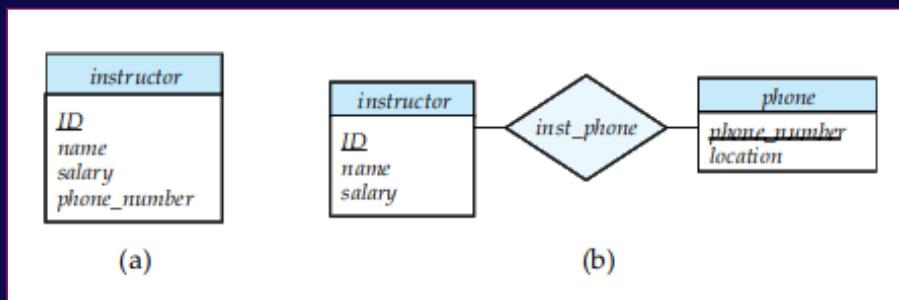


Figure 74 - Alternatives for adding phone to the instructor entity set

But what exactly differentiates these two definitions of an instructor? Well, treating a phone number as an attribute implies that instructors have precisely one phone number apiece. In contrast, treating phones as an entity allows instructors to have multiple phone numbers (even zero) associated with them. That being said, phone numbers could be defined as a multivalued attribute, enabling multiple phones per instructor. So what makes treating a phone as an entity preferable? The answer lies in the extra information one might want to preserve about a phone, such as its location, type, or all individuals who share the same phone. It follows, then, that treating the phone as an entity better models this scenario and is thus more general than treating it as an attribute.

Of course, one might ask what exactly constitutes an attribute versus an entity set. Alas, the answer is not so **clear-cut**. The distinctions between the two hinge on the structure of the real-world enterprise being modeled and on the semantics associated with the attribute at hand.

A common mistake that people make in **E-R database schema design** is to utilize the primary key of an entity set as an attribute of another entity set, instead of using a relationship. For instance, it would be incorrect to model the **ID** of a student as an attribute of an instructor, even if each instructor advises only one student.

The use of entity sets versus relationship sets can be a tricky issue. The question of whether to represent an object as an entity set or a relationship set is not always clear-cut and requires careful consideration. As depicted in the figure, the takes relationship set can be used to model the situation where a student takes a course. However, an alternative approach would be to create a **course-registration** record for each course that each student takes, and represent it as an entity set called "*registration*". This entity set would be related to exactly one student and one section, creating two relationship sets: *one to relate course-registration records to students, and another to relate them to sections*. While both approaches accurately represent the university's information, using the takes relationship set is more concise and efficient.

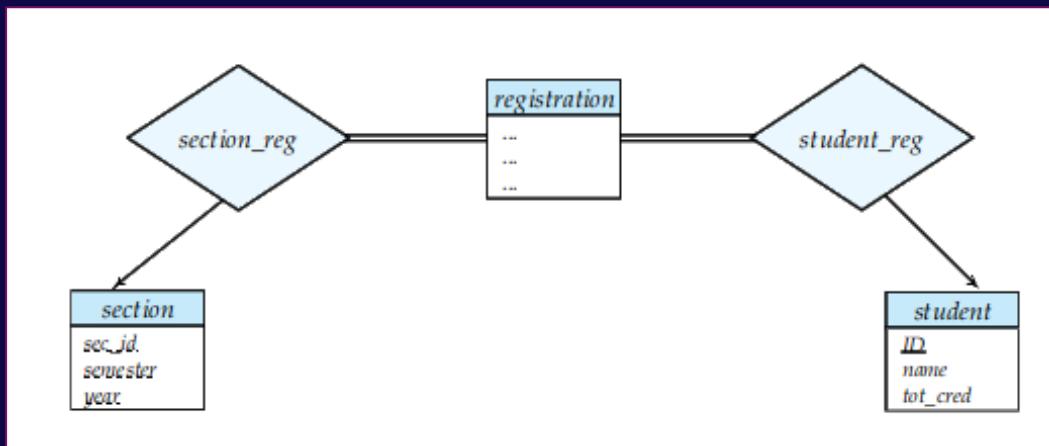


Figure 75 - Replacement of takes by registration and two relationship sets

One way to determine whether to use an entity set or a relationship set is to use a relationship set to describe an action that occurs between entities. This approach can also be helpful in deciding which attributes may be more appropriately expressed as relationships.

In addition, it is important to consider whether to use binary or **n-ary relationship sets**. While relationships in databases are often binary, some relationships that appear to be nonbinary could actually be better represented by several binary relationships. For example, instead of creating a ternary relationship called a parent to relate a child to his/her mother and father, two binary relationships called mother and father could be created. Using binary relationship sets is preferable in this case, as it provides a record of a child's mother even if the father's identity is unknown.

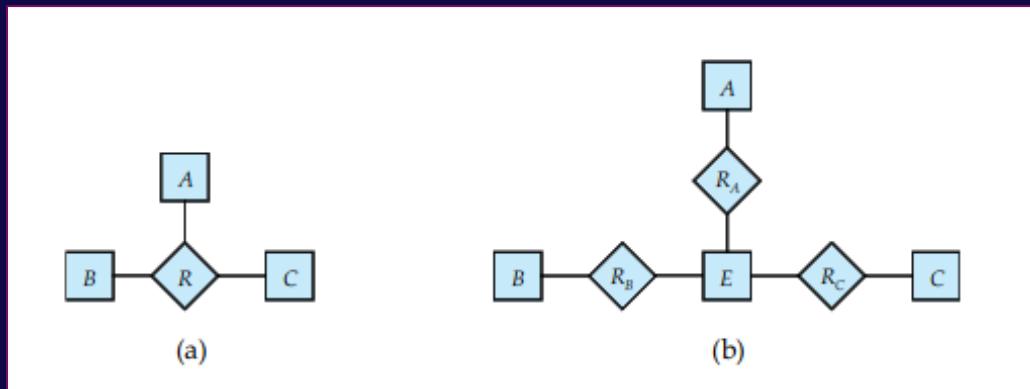


Figure 76 - Ternary relationship versus three binary relationships

It is always possible to replace a nonbinary (n -ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. This can be done by creating an entity set to represent the relationship set, and then creating three relationship sets to replace the original ternary relationship set. However, this process can increase the complexity of the design and overall storage requirements.

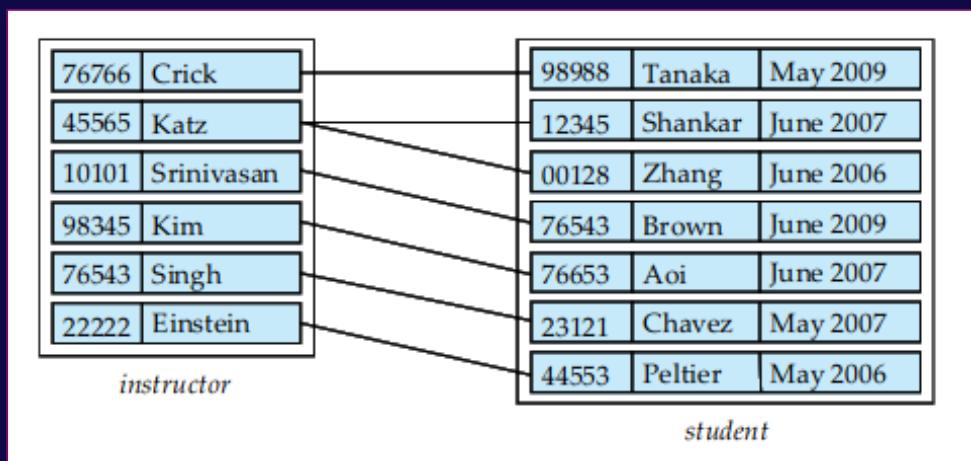


Figure 77 - Ternary relationship versus three binary relationships

In some cases, an **n-ary relationship set** may be necessary to show that several entities participate in a single relationship. Moreover, there may not be a way to translate constraints on a ternary relationship into constraints on binary relationships. For example, consider the relationship set *proj guide*, which relates instructors, students, and projects. While it cannot be directly split into binary relationships between instructors and projects, and between instructors and students, doing so may not be very natural.

The placement of relationship attributes can be a perplexing challenge. A cardinality ratio of a relationship can have a profound impact on where to place these attributes. In some cases, attributes of **one-to-one** or **one-to-many** relationship sets can be associated with one of the participating entity sets, instead of the relationship set itself.

For instance, let's take the case of an **advisor-student relationship**, where one instructor may advise several students, but each student can only be advised by a single instructor. In this scenario, the attribute 'date', which specifies the date on which an instructor became the advisor of a student, can be associated with the student entity set, rather than with the advisor relationship set. This is because each student entity participates in a relationship with at most one instance of an instructor. However, this design decision should reflect the characteristics of the enterprise being modeled.

In contrast, the choice of attribute placement is more straightforward for **many-to-many relationship sets**. In such cases, the attribute must be associated with the **many-to-many relationship** set itself, rather than with any of the participating entities. This is because the attribute is determined by the combination of the participating entity sets, rather than by either entity separately.

Therefore, the placement of relationship attributes in a database design must be thoughtfully considered to accurately represent the characteristics of the enterprise being modeled.

Extended E-R Features

The basic **E-R model** has been the bedrock of database design for decades, but as with any paradigm, there are some aspects that can be better expressed with extensions. In this section, we will explore some of the extended **E-R features** that can help us model complex databases more effectively.

- One such feature is specialization, which allows us to create more specific entity sets from a general entity set. For example, we can create a "*student*" entity set that specializes from the "*person*" entity set, with additional attributes such as "*major*" and "*class standing*."
- Another extension is a generalization, which allows us to group similar entity sets into a more general entity set. For example, we can group the "*student*" and "*instructor*" entity sets into a more general "*personnel*" entity set.

Higher-and lower-level entity sets allow us to organize entity sets into hierarchies. Attribute inheritance is a related feature that allows attributes to be inherited by lower-level entity sets from higher-level entity sets.

Finally, aggregation is an extension that allows us to treat a collection of entities as a single entity. For example, we can treat a department and all of its associated courses as a single entity.

To illustrate these extended **E-R features**, we will use a more elaborate database schema for a university, which includes an entity set called "*person*" with attributes such as *ID*, *name*, and *address*. With these tools at our disposal, we can model even the most complex databases with clarity and precision.

In the realm of database modeling, entities within an entity set may have distinct attributes that set them apart from other entities in the same set. Enter the concept of specialization, an extended feature of the **E-R model** that allows for subgroupings of entities within a set. For example, the entity set “*person*” may be further classified as an employee or a student, each with their own set of attributes beyond those of the parent entity set.

The university setting provides ample opportunity for specialization. Students can be further categorized as graduate or undergraduate, each with their own unique attributes. Graduate students may have an office assigned to them, while undergraduate students are assigned to a residential college. Similarly, employees can be classified as instructors or secretaries, with their own set of specialized attributes and relationships to other entity sets.

An entity set can be specialized by more than one distinguishing feature, resulting in overlapping specializations, or by at most one feature, resulting in disjoint specializations. This concept is visually represented in an **E-R diagram** by a hollow arrowhead pointing from the specialized entity set to the other entity set. This relationship is called the **ISA relationship** and is akin to stating that an instructor “*is an*” employee.

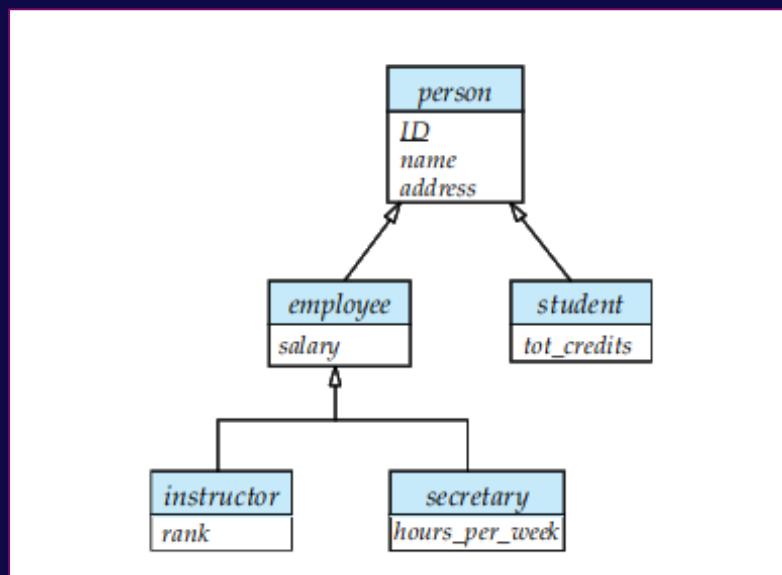


Figure 78 - Specialization and generalization

It opens the door to subclass relationships, which are crucial to accurately modeling complex data structures.

The refinement of an initial entity set into successive levels of entity subgroupings represents a crucial top-down process in which important distinctions are made explicit. However, the design process may also proceed in a bottom-up fashion, whereby multiple entity sets are synthesized into a **higher-level entity** set on the basis of common features.

Upon closer inspection, the designer may recognize that the *instructor* and *secretary* entity sets share some key attributes, such as their identifier, name, and salary attributes. This commonality can be expressed through generalization, which is a containment relationship that exists between a higher-level entity set and one or more lower-level entity sets.

To illustrate this concept, let's consider the example of an employee entity set that contains both instructors and secretaries as lower-level entity sets. In order to create a **generalization**, the attributes that are conceptually the same must be **given a common name and represented by the higher-level entity of a person**. In this way, the attribute names *ID*, *name*, and *address* can be used to create a cohesive and unified schema that accurately reflects the needs of the user.

It's worth noting that **higher-** and **lower-level** entity sets can also be referred to as *superclass* and *subclass*, respectively. For all practical purposes, generalization is simply the inverse of specialization, and both processes are typically employed in combination during the course of designing an **E-R schema** for an enterprise. In terms of the E-R diagram itself, there is no need to distinguish between specialization and generalization; rather, new levels of entity representation are distinguished or synthesized as needed to fully express the database application and the user requirements.

While the two approaches may differ in terms of their starting point and overall goal, they ultimately serve the same purpose of creating a robust and effective database schema. The specialization emphasizes differences among entities within a set by creating distinct lower-level entity sets, while generalization synthesizes entity sets *into a single, higher-level entity* set based on shared attributes and relationship sets. Through the careful application of these techniques, designers can create databases that are efficient, effective, and optimized for the needs of their users.

This crucial property of higher- and lower-level entities created by specialization and generalization brings a level of coherence and structure to an otherwise complex and multifaceted model.

Consider, for instance, the student and employee entities, which inherit the attributes of a person, and are therefore described by their *ID*, *name*, and *address*, as well as additional attributes such as tot cred and salary, respectively. And this inheritance, as we learn, applies through all tiers of lower-level entity sets, thus reinforcing the coherence of the model.

But inheritance is not limited to attributes alone; **lower-level entity** sets also inherit participation in the relationship sets in which their **higher-level entities participate**. Take, for example, the person entity set that participates in a relationship with the department. This relationship is implicitly inherited by the *student*, *employee*, *instructor*, and *secretary* entity sets, which also participate in the person dept relationship with the department.

As we explore the implications of ER modeling, it becomes clear that whether the model is arrived at through specialization or generalization, the outcome is largely the same: a **higher-level** entity set with attributes and relationships that apply to **all lower-level** entities, and lower-level entity sets with distinctive features that apply only within their own particular sets.

Constraints, however, can be placed on a particular generalization to model an enterprise more accurately. These constraints can involve determining which entities can be members of a **lower-level entity set**, as in **condition-defined** or **user-defined sets**, or whether entities may belong to more than one **lower-level** set, as in **disjoint** or **overlapping** sets.

The completeness constraint on a generalization or specialization can also be specified, indicating whether an entity in the **higher-level** set must belong to at least one of the lower-level sets.

- The **Entity-Relationship (E-R) model** has proven to be a valuable tool. However, as with any model, it has its limitations. One such limitation is its inability to express relationships among relationships, as illustrated by the ternary relationship proj guide between an instructor, student, and project.
- To capture the evaluation reports required for this relationship, an entity called evaluation is created with a **primary key evaluation id**. To link evaluations to the (*student*, *project*, *instructor*) combination, a quaternary relationship set eval for between *instructor*, *student*, *project*, and *evaluation* is created. However, combining the relationship sets *proj guide* and *eval for* into a single relationship is not advisable, as some combinations may not have an associated evaluation.
- To avoid redundancy, the concept of aggregation is introduced. Aggregation is an abstraction that treats relationships as **higher-level entities**. In this case, the

relationship set proj guide is treated as a higher-level entity set called **proj guide**, with a binary relationship eval between **proj guide** and **evaluation** representing which (*student, project, instructor*) combination an evaluation is for.

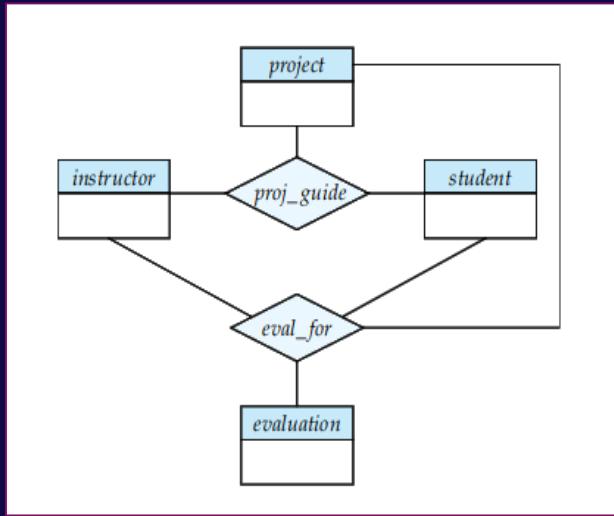


Figure 79 - E-R diagram with redundant relationships

- The extended **E-R features** can then be translated into relation schemas. Two methods of designing relation schemas for an E-R diagram that includes generalization are discussed, with foreign-key constraints being a potential issue in the second method.

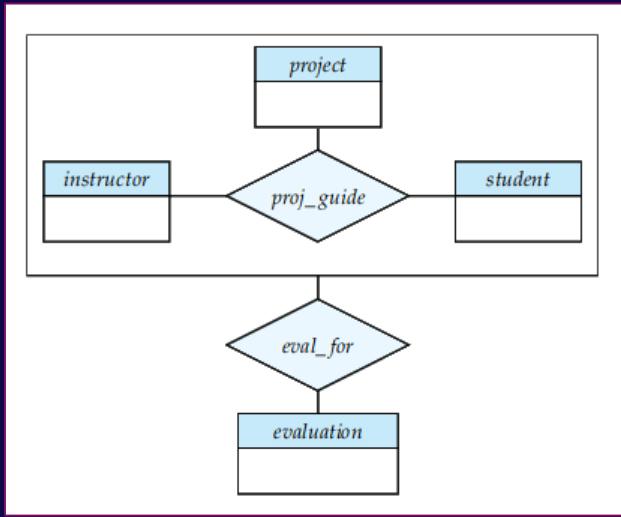


Figure 80 - E-R diagram with aggregation

In conclusion, the E-R model has proven to be a valuable tool in database design, but its limitations must be considered when dealing with complex relationships.

The representation of aggregation can often pose a challenge. With the aid of the Figure - *E-R diagram with aggregation*, we can easily craft a schema for the relationship set eval that lies between the aggregation of proj guide and the entity set evaluation.

To achieve this, we include an attribute for each attribute in the **primary keys** of the entity set **evaluation** and the **relationship** set **proj guide**, as well as any descriptive attributes that may exist for the relationship set eval. We then proceed to transform the relationship sets and entity sets within the aggregated entity set according to the well-established rules we have already defined.

The **primary key** of the aggregation is the **primary key** of its defining relationship set, and there is no need for a separate relation to representing the aggregation. Instead, the relationships created from the defining relationship are utilized. With these fundamental principles in mind, navigating the complexities of database design can be a far less daunting task.

Alternative Notations for Modeling Data

In the quest for the most intuitive and effective notation for modeling data, a number of alternatives have been proposed, each with its own unique set of symbols and conventions. Among the most widely used are the **Entity-Relationship (E-R)** diagram and the **Unified Modeling Language (UML)** class diagram. These notations play a crucial role in the database schema design process, providing an intuitive and easy-to-understand visual representation of the data model of an application.

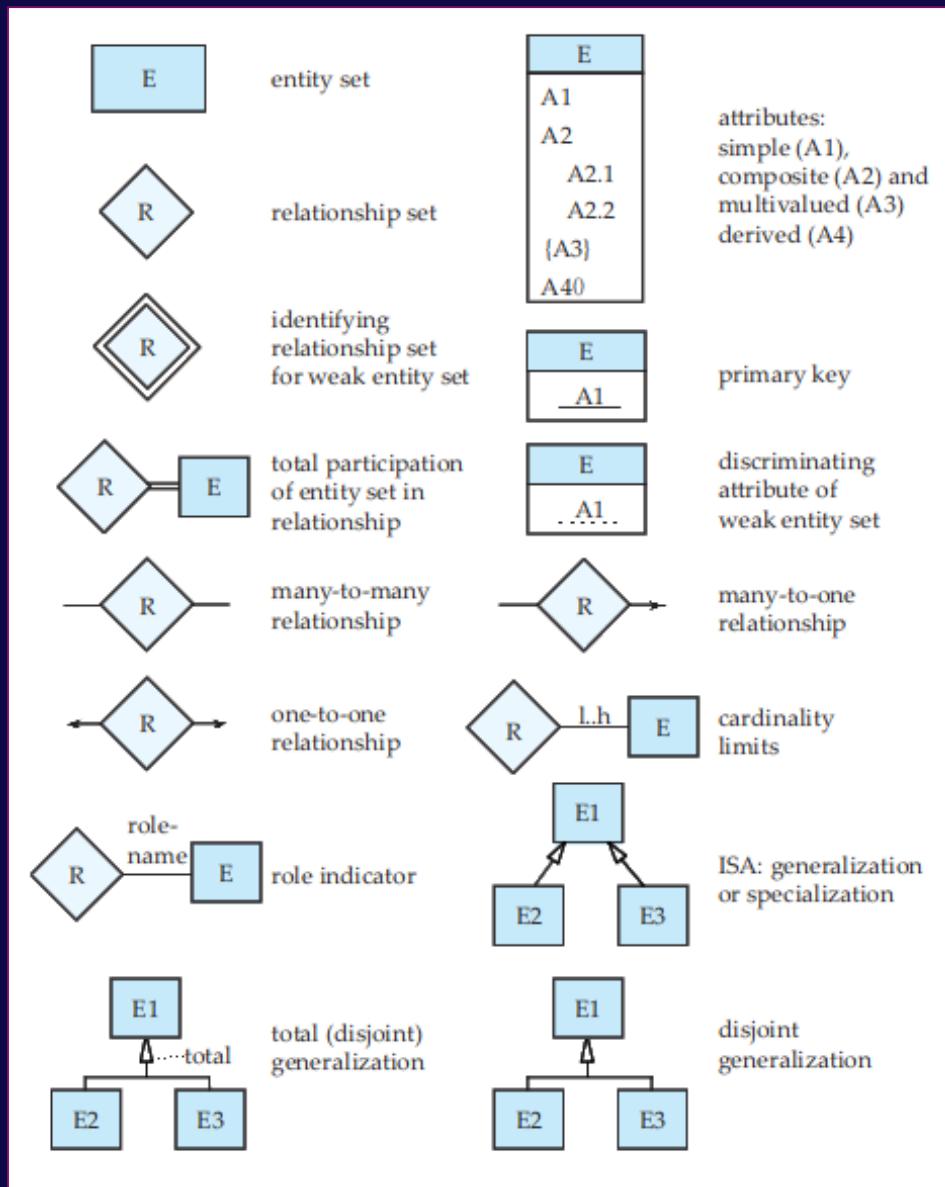


Figure 81 - Symbols used in the E-R notation

To provide a basis for comparison, we examine some of the alternative **E-R diagram** notations as well as the **UML class diagram** notation. The set of symbols used in our **E-R diagram** notation is summarized in the Figure above.

Alternative E-R diagram notations include different ways of representing entity and relationship attributes, as well as cardinality constraints and generalization. Some use ovals to represent attributes, with primary key attributes underlined, while others represent relationship attributes by connecting ovals to the diamond representing the relationship. Cardinality constraints can be depicted using various symbols, such as the "crow's-foot" notation, or labels such as "*" and "1."

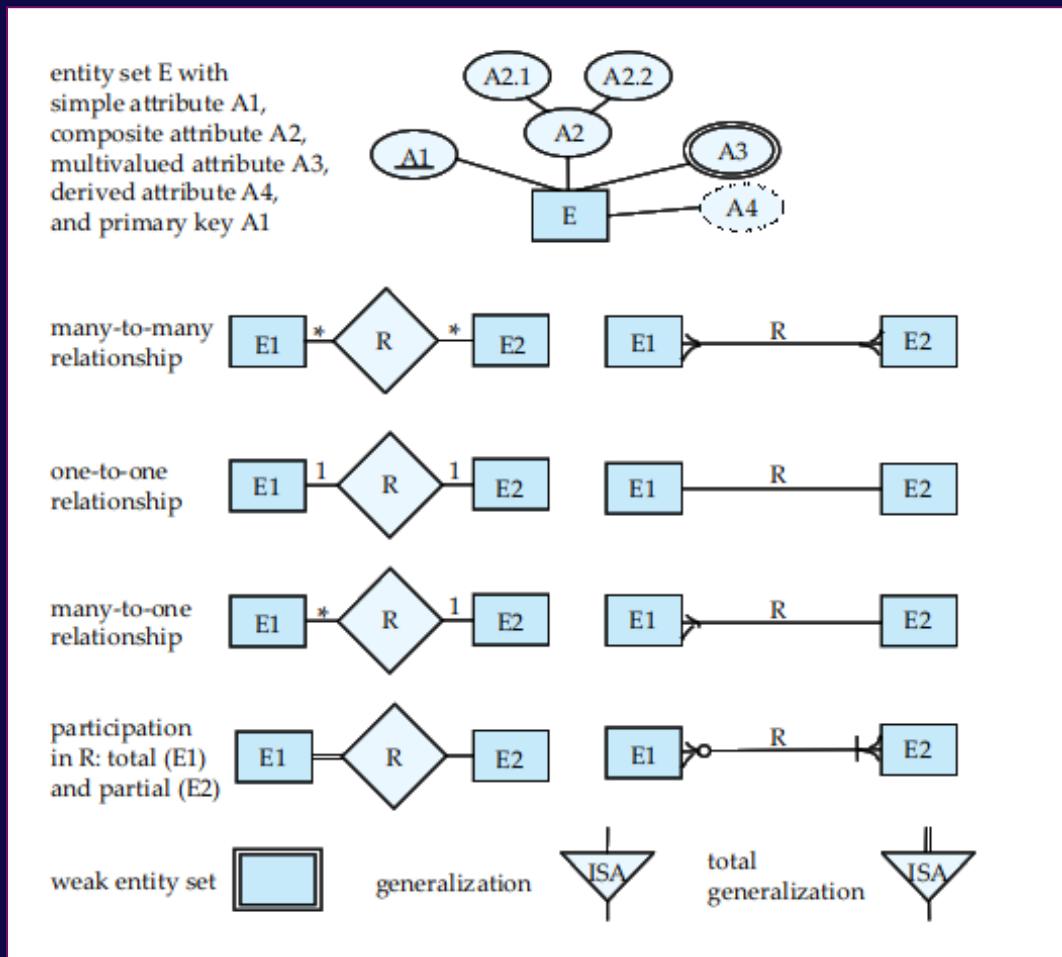


Figure 82 - Alternative E-R notations

Our new notation, which is closer to the UML class diagram notation, provides a more compact representation of attributes and is supported by many **E-R modeling tools**. However, it is important to note that different tools may offer various notational variants.

While **E-R diagrams** are crucial for modeling data representation in a software system, they are only one component of an overall system design. **UML** provides a standard for creating specifications of various components of a software system, including models of user interactions, functional modules, and their interactions. As such, **UML** plays a vital role in the design and development of complex software systems.

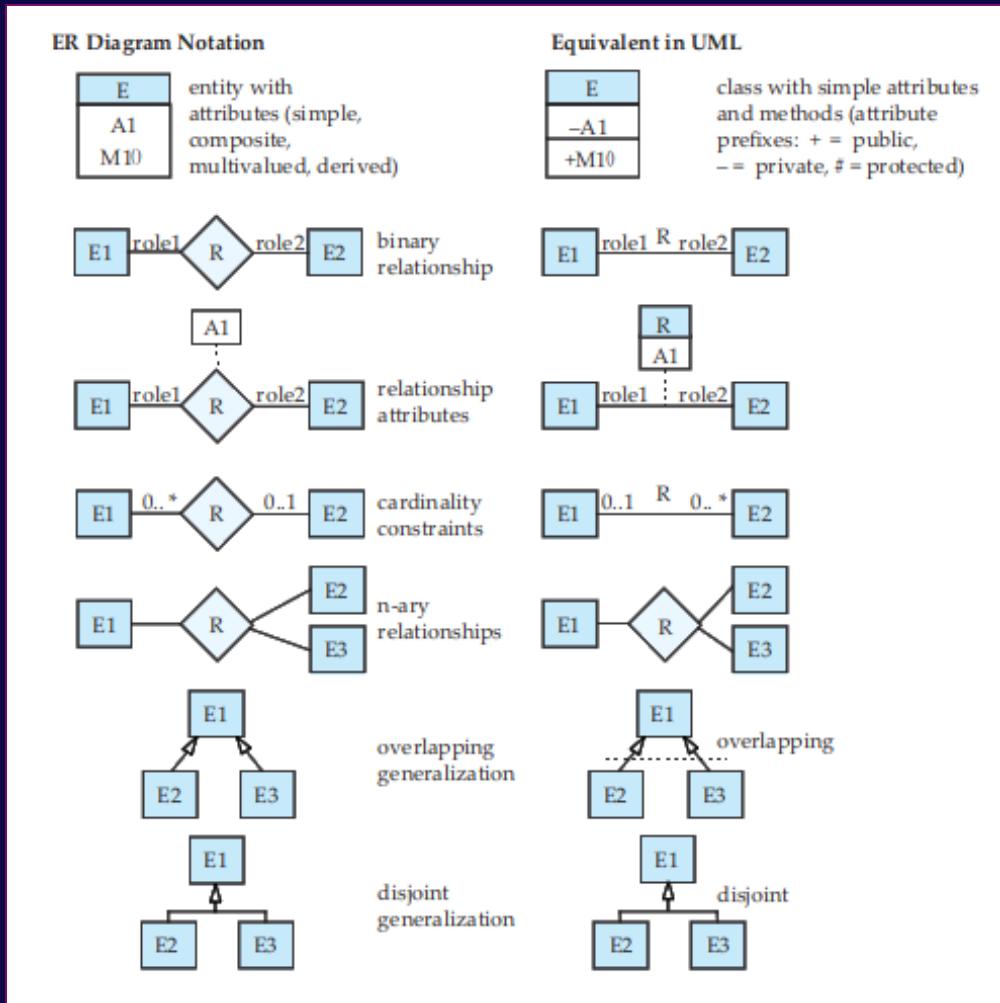


Figure 83 - Symbols used in the UML class diagram notation

Other Aspects of Database Design

In exploring the complexities of database design, we must not fall prey to the misguided notion that schema design is the **be-all** and **end-all**. There are, in fact, a multitude of factors that must be taken into account when considering the optimal design for a database. We shall only briefly survey a few of these factors here but will delve more deeply into them in future chapters.

One crucial aspect of database design is the implementation of data constraints, which can be expressed using SQL. Constraints serve to ensure consistency in data preservation and can be centrally located and updated, making them more reliable than relying on individual application programs to enforce constraints. Furthermore, the explicit expression of constraints allows for the use of unique identifiers in designing relational database schemas, such as social security numbers, which can be used to link data related to a particular individual across multiple relations.

Performance is another critical consideration in database design, as it affects the efficiency of users and processes interacting with the system. Throughput, which measures the number of queries or updates processed per unit of time, and response time, which measures the time a single transaction takes from start to finish, are the **primary metrics** for performance. While throughput is the focus of systems processing large numbers of transactions, response time is critical for systems interacting with people or time-critical systems. The type of queries that are expected to be most frequent, as well as the relative mix of update and read operations, are important factors in the choice of indices to speed query evaluation.

Authorization requirements are also a crucial consideration in database design. SQL allows access to be granted to users on the basis of components of the logical design of the database, so relation schemas may need to be decomposed to facilitate access rights. Division of data becomes even more critical when the data are distributed across systems in a computer network.

In short, there are myriad factors to consider when designing a database beyond schema design, and a holistic approach is necessary to ensure optimal performance and functionality.

3.2 Relational Database Design

In this chapter, the intricate art of relational database design is brought to the fore. Akin to the **E-R model**, this pursuit aims to create a schema that obviates redundancy while facilitating easy access to stored data. The key to achieving this objective is the crafting of schemas that conform to the appropriate normal form. To accomplish this, an understanding of the enterprise being modeled is vital, and this can be gleaned from a well-crafted **E-R diagram** or other relevant sources of information.

With this in mind, the chapter introduces a formal method of relational database design predicated on the notion of functional dependencies. Using this method, the team defines normal forms in terms of functional dependencies and other forms of data dependencies. Before delving into the intricacies of this design approach, the chapter scrutinizes the relational design conundrum, focusing on the schemas derived from an **extant entity-relationship design**. In sum, this chapter offers a masterclass in the design of relational databases.

Features of Good Relational Designs

In exploring the intricacies of relational database design, we must tread carefully and consider the subtle nuances of our design choices. The allure of creating larger schemas to reduce the complexity of our queries may seem tempting, but we must be wary of the consequences that come with storing redundant information. As we learned from the university database example, the inst dept schema presents a double-edged sword that risks creating inconsistency and hindering the creation of new departments.

```
classroom(building, room.number, capacity)
department(dept.name, building, budget)
course(course.id, title, dept.name, credits)
instructor(ID, name, dept.name, salary)
section(course.id, sec.id, semester, year, building, room.number, time.slot.id)
teaches(ID, course.id, sec.id, semester, year)
student(ID, name, dept.name, tot.cred)
takes(ID, course.id, sec.id, semester, year, grade)
advisor(s.ID, i.ID)
time.slot(time.slot.id, day, start.time, end.time)
prereq(course.id, prereq.id)
```

Figure 84 - Schema for the university database

Similarly, the decision to split a schema into smaller, more specialized schemas requires a nuanced approach that takes into account the unique rules and functional dependencies of the database.

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept.name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Figure 85 - The inst dept table

We cannot simply rely on **ad-hoc methods** of identifying repetition in the data as it may be a mere coincidence or a special case that does not generalize to other scenarios.

Instead, we must turn to the fundamental rules and constraints that govern the database and use functional dependencies to guide our schema decomposition.

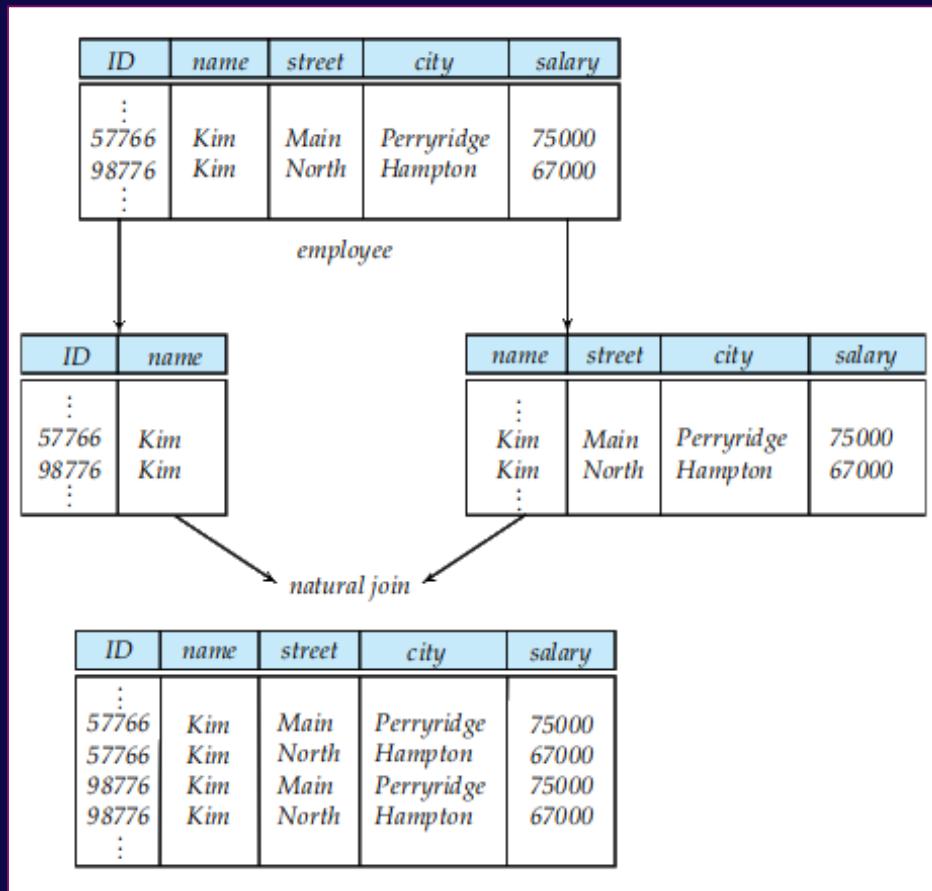


Figure 86 - Loss of information via a bad decomposition

In short, the art of relational database design requires a delicate balance between simplicity, efficiency, and accuracy. We must constantly weigh the trade-offs between larger and smaller schemas, redundant and non-redundant information, and specialized and generalized functionality. Only through a thoughtful and meticulous approach can we hope to create databases that are both reliable and effective.

Atomic Domains and First Normal Form

The concept of atomic domains and the first normal form is a fundamental and essential principle. When designing tables from **entity-relationship models**, which allow for attributes with substructure such as multivalued or composite attributes, we must eliminate this **substructure** in the **relational model**. This is achieved by creating a separate attribute for each component of a composite attribute and by creating one tuple for each item in a multivalued set.

In the relational model, we formalize the idea that attributes have **no substructure** and define an **atomic domain** as one where the elements are considered indivisible units. A relation schema is in **first normal form (1NF)** if all attributes have atomic domains. For instance, sets of names or identification numbers with department codes and unique employee numbers are examples of non-atomic domains that would violate **1NF**.

Using set-valued attributes can lead to redundant storage and inconsistencies, which can cause problems when updating data. While some types of non-atomic values, such as composite-valued attributes, can be useful, they should be used with care. Forcing a first normal form representation in domains with complex entity structures can create unnecessary burdens on application programmers, who must write code to convert data into an atomic form, and runtime overhead. Nonetheless, modern database systems support various types of non-atomic values.

While some non-atomic values may have utility in certain circumstances, the potential for inconsistencies and redundancy requires careful consideration when using them in database design.

Decomposition Using Functional Dependencies

In the realm of relational database design, the evaluation of whether a given schema should be decomposed can be achieved via a formal methodology that is grounded in the concepts of keys and functional dependencies. In order to discuss algorithms for database design, it is necessary to use notation that pertains to arbitrary relations and their schema, rather than focusing only on examples. This notation is reminiscent of that introduced in the relational model.

A	B	C	D	
a ₁	b ₁	c ₁	d ₁	
a ₁	b ₂	c ₁	d ₂	
a ₂	b ₂	c ₂	d ₂	
a ₂	b ₃	c ₂	d ₃	
a ₃	b ₃	c ₂	d ₄	

Figure 87 - Sample instance of relation r

In the context of this notation, certain sets of attributes can be used to uniquely identify tuples in a given relation, and these sets are known as superkeys. **Keys**, **candidate keys**, and **primary keys** can be seen as types of real-world constraints that are commonly represented formally. Functional dependencies, on the other hand, allow for constraints that identify the values of certain attributes within a schema.

A functional dependency is said to hold on to a given schema if it is satisfied in every legal instance of that schema. If a set of attributes is a **superkey** for a relation schema, then the functional dependency between that set and the entire set of attributes of that schema holds. This means that for every pair of tuples in any legal instance of the relation that shares the same **superkey**, their entire attribute sets will be identical.

The relationship between **superkeys** and **functional dependencies** is demonstrated with the example of a schema involving departments and instructors, in which the functional dependency between the department name and the budget holds, indicating that the department name can be used to uniquely identify the budget of a given department.

Ultimately, the concepts of keys and functional dependencies are crucial in the evaluation of whether a given relational schema should be decomposed.

<i>building</i>	<i>room.number</i>	<i>capacity</i>
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 88 - An instance of the classroom relation

As we delve deeper into the world of database design, we come across the **Boyce-Codd normal form (BCNF)**, a highly desirable state that eliminates all redundancy based on functional dependencies. While there may still be other forms of redundancy remaining, BCNF eradicates any redundancy that can be discovered through functional dependencies.

For a relation schema **R** to be considered in **BCNF** with respect to a set **F** of functional dependencies, there must be at least one of the following criteria:

- A functional dependency is trivial, meaning that the dependent attribute is a subset of the determinant.
- The determinant is a superkey for schema **R**.
- To demonstrate this concept, we revisit the relational **schema inst_dept (ID, name, salary, dept_name, building, budget)**, which we previously saw was not in **BCNF** due to the functional dependency $dept_name \rightarrow budget$. As $dept_name$ is not a **superkey**, we decompose *inst_dept* into instructor and department. Upon doing so, we discover that the instructor is in **BCNF**, with ID being the **primary key** and a **superkey** for the schema. Similarly, the department is also in **BCNF**, with $dept_name$ being the primary key and a superkey for the schema.
- When a schema is not in **BCNF**, we must decompose it to remove the redundancy. To do so, we replace **R** in our design with two schemas, (*Lambda U Beta*) and (*R - (Beta - Lambda)*). In the case of *inst_dept*, we replace it with (*dept_name, building, budget*) and (*ID, name, dept_name, salary*). This process continues until all resulting schemas are in **BCNF**.

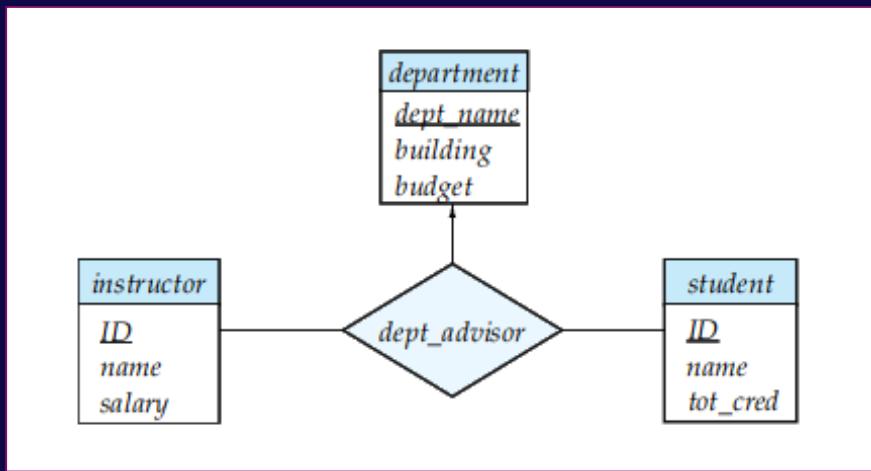


Figure 89 - The dept advisor relationship set

While **BCNF** is highly desirable, it may hinder the efficient testing of certain functional dependencies if the decomposition is not done properly. This is because testing constraints each time the database is updated can be costly. Therefore, designing the database in a way that allows for efficient constraint testing is crucial.

Functional-Dependency Theory

In a world driven by data, the ability to reason systematically about **functional dependencies** is a **fundamental skill** for anyone working with databases. **Functional-Dependency Theory**, as exemplified in the groundbreaking work of *Armstrong*, provides a powerful framework for reasoning about these dependencies and ensuring that database schemas conform to desired normal forms.

The crux of the theory lies in the concept of logical implication: given a set of functional dependencies F on a schema, we can prove that certain other functional dependencies hold on the schema. In other words, we can determine which dependencies are "*logically implied*" by F . However, this process can be time-consuming, particularly for large sets of dependencies.

```
 $F^+ = F$ 
repeat
  for each functional dependency  $f$  in  $F^+$ 
    apply reflexivity and augmentation rules on  $f$ 
    add the resulting functional dependencies to  $F^+$ 
  for each pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$ 
    if  $f_1$  and  $f_2$  can be combined using transitivity
      add the resulting functional dependency to  $F^+$ 
until  $F^+$  does not change any further
```

Figure 90 - A procedure to compute F^+

To address this issue, *Armstrong's axioms* provide a set of rules for inferring additional dependencies from a given set. These axioms are both sound and complete, meaning they generate all correct dependencies and no incorrect ones. Moreover, they can be used to prove the correctness of additional rules, such as the **Union rule**, **Decomposition rule**, and **Pseudotransitivity rule**.

```
result :=  $\alpha$ ;
repeat
  for each functional dependency  $\beta \rightarrow \gamma$  in  $F$  do
    begin
      if  $\beta \subseteq result$  then  $result := result \cup \gamma$ ;
    end
  until ( $result$  does not change)
```

Figure 91 - An algorithm to compute a^+

In practice, the ability to reason about functional dependencies is critical for ensuring the accuracy and efficiency of databases. Whether working with small or large sets of dependencies, understanding the principles of **Functional-Dependency Theory** and **Armstrong's axioms** is essential for anyone seeking to master this field.

Ensuring the satisfaction of **functional dependencies (FDs)** in a given relation schema is paramount. Such satisfaction guarantees the integrity and consistency of the data and is a fundamental requirement of any reliable database system. By checking the satisfaction of **FDs** on every update can be computationally expensive, especially for large datasets.

The canonical cover is a simplified set of **FDs** that has the same closure as the original set. It is obtained by eliminating extraneous attributes from each **FD** in the set until no extraneous attributes remain. An extraneous attribute is one that can be removed from a given **FD** without altering its closure. In other words, it is a redundant attribute that does not contribute to the satisfaction of the **FD**.

The process of computing the canonical cover involves iteratively removing extraneous attributes from each **FD** in the set until a fixed point is reached. This can be done efficiently by computing the closures of the set of **FDs** with and without each extraneous attribute. If the closure of the **FD** set with the extraneous attribute is the same as the closure of the **FD** set without the attribute, then the attribute is extraneous and can be safely removed.

Once we have obtained the canonical cover of a set of **FDs**, we can use it to check the satisfaction of **FDs** on updates. This is because any relation that satisfies the canonical cover also satisfies the original set of **FDs** and vice versa. Furthermore, the canonical cover has unique left sides and no extraneous attributes, making it an ideal representation of the minimal set of **FDs** required for the satisfaction of the original set.

A canonical cover is a powerful tool for reducing the complexity of **FD** satisfaction checks in relational databases. It allows us to obtain a simplified set of **FDs** that is equivalent to the original set while removing any **redundancy** and **ensuring minimalism**. With the canonical cover, we can ensure the integrity and consistency of our data, without sacrificing performance or scalability.

Algorithms for Decomposition

In the world of database design, lossless decomposition is a concept of utmost importance. This principle states that a database schema, defined by a relation schema $r(R)$ and a set of functional dependencies F , can be decomposed into two relation schemas, $r1(R1)$ and $r2(R2)$, without losing any information, as long as the original relation r contains the same set of tuples as the result of a natural join of the projection of r onto $R1$ and $R2$. This lossless join decomposition is crucial in ensuring that the database remains consistent and complete even after the decomposition process.

However, not all decompositions are lossless. A lossy decomposition occurs when information is lost during the decomposition process, resulting in a joint result that is a superset of the original relation but lacks certain key information. To avoid this, it is essential to test whether decomposition is lossless or not. This can be done using functional dependencies, where a decomposition is lossless if $R1 \cap R2$ forms a superkey of either $R1$ or $R2$.

Another important concept in database design is dependency preservation. This principle states that a decomposition of a schema R into multiple schemas should preserve all the functional dependencies of the original schema. To test this, we can restrict the set of functional dependencies to each individual relation schema, and verify that the union of these restrictions implies all the original functional dependencies. If this is the case, then the decomposition is said to be dependency-preserving.

```
compute  $F^+$ ;
for each schema  $R_i$  in  $D$  do
    begin
         $F_i$  := the restriction of  $F^+$  to  $R_i$ ;
    end
 $F' := \emptyset$ 
for each restriction  $F_i$  do
    begin
         $F' = F' \cup F_i$ 
    end
compute  $F'^+$ ;
if ( $F'^+ = F^+$ ) then return (true)
else return (false);
```

Figure 92 - Testing for dependency preservation

While these concepts may seem complex and technical, they are essential in ensuring that databases remain consistent and complete, and that data is not lost during the

decomposition process. By understanding lossless decomposition and dependency preservation, we can design databases that are robust, efficient, and accurate.

As we delve into the intricacies of database design, we are faced with the challenge of developing designs that adhere to the rules of appropriate normal form. It is for this reason that we require algorithms to assist in the generation of such designs, and in this section, we shall explore algorithms for **BCNF** and **3NF**.

To begin with, we must test whether a relation schema is in **BCNF**, and while the definition of **BCNF** can be used for this purpose, the computation of F^+ can prove to be a tedious task. However, simplified tests can be performed to verify whether a nontrivial dependency causes a violation of **BCNF**. Additionally, it is important to note that the testing procedure cannot be employed when a relation is decomposed, as it may not be sufficient to use F to test a decomposed relation for a **BCNF** violation. Therefore, we may require a dependency that is in F^+ , but not in F , to demonstrate that a decomposed relation is not in **BCNF**.

```
result := {R};
done := false;
compute F+;
while (not done) do
    if (there is a schema Ri in result that is not in BCNF)
        then begin
            let α → β be a nontrivial functional dependency that holds
            on Ri such that α → Ri is not in F+, and α ∩ β = ∅;
            result := (result - Ri) ∪ (Ri - β) ∪ (α, β);
        end
    else done := true;
```

Figure 93 - BCNF decomposition algorithm

An alternative **BCNF** test can be applied to check if a relation in a decomposition of **R** satisfies **BCNF**. The algorithm uses dependencies that demonstrate the violation of **BCNF** to perform the decomposition, generating not only **BCNF** schemas but also a lossless decomposition.

However, it is important to note that the **BCNF** decomposition algorithm takes exponential time, and while there are algorithms that can compute a **BCNF** decomposition in polynomial time, they may “*over-normalize*” by decomposing a relation unnecessarily.

```
let  $F_c$  be a canonical cover for  $F$ ;
i := 0;
for each functional dependency  $\alpha \rightarrow \beta$  in  $F_c$ 
    i := i + 1;
     $R_i := \alpha\beta$ ;
if none of the schemas  $R_j$ ,  $j = 1, 2, \dots, i$  contains a candidate key for  $R$ 
then
    i := i + 1;
     $R_i :=$  any candidate key for  $R$ ;
/* Optionally, remove redundant relations */
repeat
    if any schema  $R_j$  is contained in another schema  $R_k$ 
    then
        /* Delete  $R_j$  */
         $R_j := R_i$ ;
        i := i - 1;
until no more  $R_j$ s can be deleted
return ( $R_1, R_2, \dots, R_i$ )
```

Figure 94 - Dependency-preserving, lossless decomposition into 3NF

As an example of the practical application of the **BCNF** decomposition algorithm, let us consider a database design using the class schema. We must ensure that the set of functional dependencies that we require to hold on class are satisfied, and we can employ the **BCNF** decomposition algorithm to achieve this end. Ultimately, we must remember that the development of a robust database design requires a thorough understanding of the underlying principles and a careful application of algorithms and testing procedures to ensure adherence to the rules of appropriate normal form.

In the realm of relational database schemas, the **3NF** algorithm stands as a stalwart, a beacon of hope in a sea of uncertainty. Building a schema for each dependency in a canonical cover preserves dependencies and ensures a lossless decomposition. Though not uniquely defined and subject to the order in which dependencies are considered, the algorithm guarantees 3NF even for relations already in the form.

BCNF and **3NF** both offer advantages, but the latter allows for the preservation of dependencies without sacrificing losslessness. However, it comes with the caveat of potential repetition of information and null values to represent relationships. When designing databases with functional dependencies, our goals are to achieve **BCNF**, *losslessness*, and *dependency preservation*, but it's not always possible to satisfy all three. In such cases, we must choose between **BCNF** and dependency preservation with **3NF**.

It's worth noting that SQL lacks a way to specify functional dependencies, except for the declaration of superkeys with primary or unique constraints. While assertions can enforce functional dependencies, no database system currently supports such complex assertions. Even with a *dependency-preserving decomposition*, standard SQL can only efficiently test dependencies with a left-hand side that is key. However, materialized views can reduce the cost of testing functional dependencies, provided the database system supports **primary key constraints on materialized views**.

Decomposition Using Multivalued Dependencies

In the realm of database normalization, even the most sophisticated normalization technique, such as **Boyce-Codd normal form (BCNF)**, can sometimes fail to eliminate the problem of redundant information in certain relation schemas. Take for instance the scenario where a university instructor is associated with multiple departments. Although **BCNF** can be applied to this schema, it still suffers from redundant information due to the repetition of an instructor's address information for each department they are affiliated with.

To address this issue, a new form of constraint, called **multivalued dependencies (MVDs)**, must be introduced. While functional dependencies determine the presence or absence of certain tuples in a relation, **MVDs** require certain other tuples to be present. Multivalued dependencies are thus referred to as tuple-generating dependencies.

In a relation schema $r(R)$, a multivalued dependency $A \twoheadrightarrow B$ holds if, in any legal instance of **relation** $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[A] = t_2[A]$, there exist tuples t_3 and t_4 in r that satisfy certain conditions. If the multivalued dependency $A \twoheadrightarrow B$ is satisfied by all relations on schema R , it is considered a trivial multivalued dependency on schema R .

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figure 95 - Tabular representation

To better illustrate the distinction between functional and multivalued dependencies, consider the scenario of a university instructor with multiple addresses and affiliations with multiple departments. By applying the multivalued dependency $\text{ID} \rightarrow\!\!\! \rightarrow \text{street, city}$, we can eliminate the redundancy of repeating an instructor's address information for each department they are associated with.

<i>ID</i>	<i>dept.name</i>	<i>street</i>	<i>city</i>
22222	Physics	North	Rye
22222	Physics	Main	Manchester
12121	Finance	Lake	Horseneck

Figure 96 - An example of redundancy in a relation on a BCNF schema

<i>ID</i>	<i>dept.name</i>	<i>street</i>	<i>city</i>
22222	Physics	North	Rye
22222	Math	Main	Manchester

Figure 97 - An illegal r2 relation

While **BCNF** is a powerful tool for normalization, it may not always suffice in eliminating redundancy in relation schemas. Multivalued dependencies offer an effective means of addressing this issue by specifying conditions that must be met to ensure the presence of certain tuples in a relation.

Specifically, we explore the intricacies of the **Fourth Normal Form (4NF)**, which is a level of schema normalization that addresses the issue of multivalued dependencies.

To determine if each relation schema in a decomposition satisfies **4NF**, we must first identify the multivalued dependencies that exist within each schema. To do so, we define the restriction **F_i** of a set of functional dependencies **F** to **R_i**, which includes all functional dependencies in **F₊** that involve only attributes of **R_i**. We also consider a set **D** of both functional and multivalued dependencies, and the restriction of **D** to **R_i**, which is the set **D_i** consisting of functional dependencies and multivalued dependencies that only involve attributes of **R_i**.

The algorithm for decomposing a schema into **4NF** closely mirrors that of **BCNF**, but incorporates multivalued dependencies and uses the restriction of **D₊** to **R_i**. If we apply this algorithm to a schema such as $(ID, \text{dept name}, \text{street}, \text{city})$, we may discover nontrivial multivalued dependencies such as $\text{ID} \rightarrow\!\!\! \rightarrow \text{dept name}$, indicating the need for

further decomposition. By replacing the problematic attribute with two schemas that satisfy **4NF**, we can eliminate redundancy and improve schema design.

To ensure that our decompositions are lossless and preserve dependencies, we turn to the concept of multivalued dependencies and their relationship to losslessness. Specifically, we must verify that at least one of two multivalued dependencies holds true for a given decomposition to be considered lossless. The issue of preserving dependencies in the presence of multivalued dependencies is a complex one and requires further exploration.

More Normal Forms

The **fourth normal form (4NF)** is a powerful tool in database normalization, but it is **not** the **end-all-be-all** of normal forms. In fact, there are several more normal forms that can help us eliminate data redundancy and ensure data integrity. Join dependencies can lead to a higher level of normalization, known as the **project-join normal form (PJNF)**, which is sometimes referred to as the fifth normal form.

But **PJNF** is not the final frontier of normalization. There is a class of even more general constraints that lead to another normal form called **domain-key normal form (DKNF)**. However, while these generalized constraints can help eliminate data redundancy, they are often difficult to reason about and there is no set of sound and complete inference rules to reason about them. As a result, **PJNF** and **DKNF** are rarely used in practice.

Interestingly, the **second normal form (2NF)** is not discussed at length in this discussion of normal forms. While it has historical significance, its utility has been overshadowed by the more powerful and widely used **third normal form (3NF)** and beyond. However, it is still a valuable tool to experiment with and gain a deeper understanding of database normalization.

Database-Design Process

In delving deeper into the intricate details of the database-design process, we have uncovered significant insights that shed light on the complex interplay between the normalization of relation schemas and the broader context of database design. Our exploration has revealed that there are several paths by which a relation schema can be derived, including generating a schema from an **E-R diagram**, starting with a single relation schema and breaking it down through normalization, or designing relations **ad-hoc** and **verifying their conformance** to a **desired normal form**.

One key issue that emerges when generating relation schemas from **E-R diagrams** is the potential for functional dependencies between entity attributes to arise. For example, an instructor entity set may have attributes such as department name and department address, with a functional dependency between the former and the latter. In such cases, the normalization process becomes necessary to ensure that the **generated relation schema** meets the desired normal form. This underscores the importance of designing **E-R diagrams** with the utmost care, to avoid such issues and ensure that the normalization process can be carried out with minimal adjustments.

Moreover, we have seen that functional dependencies are useful for detecting poor **E-R design** and that the normalization process can be carried out formally as part of data modeling or left to the designer's intuition during **E-R modeling**. The **universal-relation approach** to **relational database design** assumes a single schema containing all attributes of interest, which defines how users and applications interact with the database.

In addition, we have explored the importance of **unique attribute naming**, which ensures that each attribute has a unique meaning in the database, preventing confusion between the **same attribute used in different schemas**. On the other hand, **attributes of different relations** that have the same meaning **can be named identically** to reduce user errors. The order of attribute names in a schema may not matter technically, but **primary-key attributes** are conventionally listed first to make reading default output easier.

In conclusion, our comprehensive analysis of the database-design process has illuminated the various nuances and subtleties involved, highlighting the importance of careful **E-R diagram design**, **functional dependencies**, **normalization**, and **unique attribute naming**. These insights will undoubtedly prove invaluable for database designers and developers seeking to optimize the design and functionality of their databases.

Modeling Temporal Data

In a world where temporal data reigns supreme, modeling such data can prove to be a challenging task. Consider, for instance, an *instructor* entity set and its *associated time-varying addresses*. To imbue these addresses with temporal information, one must create a multivalued attribute consisting of composite values containing an **address** and a **time interval**. But this is just the beginning of the complexity that arises when dealing with temporal data.

Entities themselves may possess valid time intervals, such as the student entity, which has a valid time from the date of entry to the date of graduation or leaving the university. Relationships too can have valid times, as in the case of the prereq relationship, which records when a course becomes a prerequisite for another course.

Adding such temporal details to an **E-R diagram** is no mean feat, for it can make the diagram difficult to create and comprehend. While there have been several proposals to extend the **E-R notation** to **account for temporal data**, there are no universally accepted standards.

To compound matters further, **functional dependencies** that we assumed to hold for **regular data** may no longer hold when tracking data values over time. Instead, we must turn to **temporal functional dependencies**, which hold on a relation schema if all snapshots of that schema satisfy the **functional dependency** $X \rightarrow Y$. While we could extend the theory of relational database design to account for temporal functional dependencies, few designers are prepared to tackle this level of complexity.

As a result, many database designers opt for simpler approaches, such as designing the database ignoring temporal changes, and then adding valid time information to each relation that requires it. This involves adding start and end time attributes, as in the case of the course relation, which may have multiple records with **different titles** and corresponding **valid time ranges**. If the relation is updated, a new tuple is added containing the new title and start time, and the end time of the previous tuple is updated accordingly.

However, when a relation has a **foreign key** referencing a temporal relation, the **database designer** must decide whether the reference is to the current version of the data or to the data as of a specific point in time. For example, while a student's transcript should refer to the course title at the time the student took the course, a reference from the instructor or student relation may only care about the current version of the data.

3.3 Application Design and Development

The ubiquitous use of databases in today's world occurs primarily through application programs. As a result, user interaction with databases is largely indirect, taking place through the medium of these programs. In response to this, **database systems** have incorporated tools such as form and **GUI builders**, which streamline the development of applications that interface with users. However, with the emergence of the **Web as the primary mode of user interaction**, the focus has shifted toward the development of **Web-based applications** that rely on databases to store data.

In this chapter on application design and development, we explore the various tools and technologies utilized in the creation of interactive applications that employ databases to manage data. After a brief introduction to application programs and user interfaces, we delve into the world of **Web-based applications**, providing an overview of relevant **Web technologies**. Into the widely utilized **Java Servlets technology** for building **Web applications**, followed by a succinct overview of Web application architectures.

To streamline the process of application development, we examine the tools available for rapid application development, while in the following section, we turn our attention to the crucial issue of performance when building **large-scale Web applications**. Afterward, we consider the paramount issue of application security, followed by a comprehensive discussion on encryption and its implementation in applications.

In conclusion, this chapter offers a comprehensive exploration of the tools and technologies employed in the development of interactive applications that rely on databases to manage data. It highlights the significance of **Web-based applications** in today's world and offers insights into key issues such as performance and security, ensuring that the reader is well-equipped to tackle the challenges of building successful applications in this arena.

Application Programs and User Interfaces

In the realm of database systems, the majority of users are not familiar with the use of query languages to directly interact with the system. Instead, application programs serve as a crucial intermediary, offering a user interface at the front end while communicating with the database at the back end. These applications accept user input through forms-based interfaces, allowing for data entry or information retrieval from the database based on the user's input. This output is then presented to the user.

One example of an application program is a university registration system. This type of system typically requires users to first authenticate themselves with a *username* and *password*, after which the system retrieves information about the user from the database, such as registered courses and name and presents it to them. The application offers users the ability to register for courses or query information about courses and instructors. Such applications are used in a variety of industries to **automate tasks ranging from sales and accounting to human resources management and inventory management**.

Some application programs are used even without the user's awareness, such as when a news site offers **users a personalized page** based on their browsing history. In this case, an application program generates a customized page for each user, taking into account their history of articles viewed. Application programs typically have a **front-end component for the user interface**, a **back-end component for database communication**, and a **middle layer containing "business logic"** that handles specific requests for information or updates.

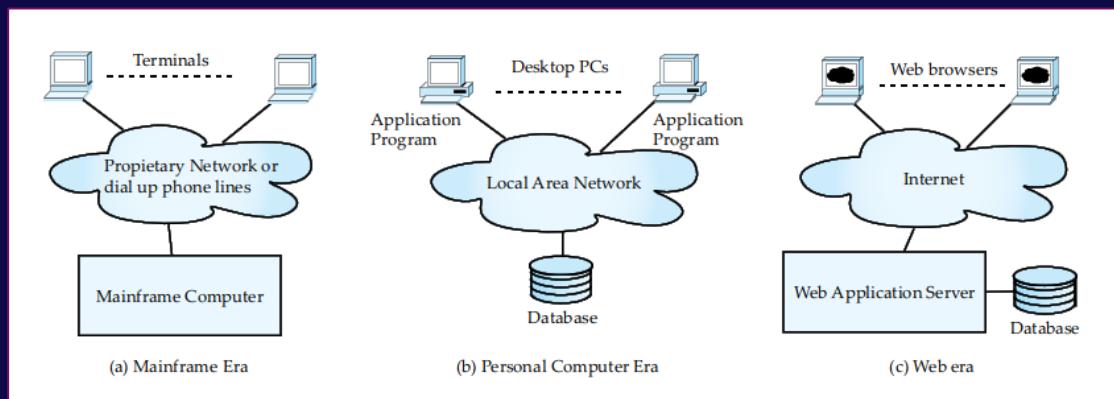


Figure 98 - Application architectures in different eras

The evolution of application architectures over time is depicted in the Figure, with airline reservations serving as an example of an application that has been around since the 1960s. Early computer applications were run on mainframe computers, with users interacting through terminals that often supported forms. The rise of personal computers brought with it a **client-server architecture**, which allowed for **powerful graphical user interfaces**. However, installing software on each user's computer made upgrades more difficult. In recent years, web browsers have become the go-to front end for database applications, using the standardized syntax of **HyperText Markup Language (HTML)** to create **forms-based interfaces**.

Unlike client-server architectures, **web-based applications** do not require any application-specific software to be installed on client machines. *JavaScript* programs are used to create sophisticated user interfaces and can be run in a safe mode that ensures security. These programs are downloaded to the browser transparently and do not require explicit installation on the user's computer. Application programs are executed on a web server in response to requests from a browser, with a variety of technologies available for creating such programs.

In the following chapter, we delve into the technologies and tools used to build web interfaces and application programs, as well as application architectures, performance, and security concerns.

Web Fundamentals

As we delve into the world of the World Wide Web, we must first acquaint ourselves with its fundamental technology. For those unfamiliar with the underlying infrastructure, let us begin with a discussion on **Uniform Resource Locators (URLs)**.

A URL is a globally unique name for each document that can be accessed on the Web. It consists of various parts that identify the document's location and how it is to be accessed. For instance, the "http" at the beginning of a URL indicates that the document is to be accessed by the **HyperText Transfer Protocol (HTTP)**, which is a protocol for transferring **HTML documents**. The second part of the URL gives the name of a machine that has a **Web server**, while the rest of the URL is the path name of the *file* on the machine or other unique identifier of a document within the machine.

```
<html>
<body>
<table border>
<tr> <th>ID</th>      <th>Name</th>      <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td> <td>Comp. Sci.</td> </tr>
<tr> <td>12345</td> <td>Shankar</td> <td>Comp. Sci.</td> </tr>
<tr> <td>19991</td> <td>Brandt</td> <td>History</td> </tr>
</table>
</body>
</html>
```

Figure 99 - Tabular data in HTML format

Furthermore, a URL can contain the identifier of a program located on the **Web server machine**, as well as arguments to be given to the program.

When a request is received for such a URL, the **Web server executes** the program, using the given arguments, and returns an **HTML document** to the **Web server**, which sends it back to the front end.

ID	Name	Department
00128	Zhang	Comp. Sci.
12345	Shankar	Comp. Sci.
19991	Brandt	History

Figure 100 - Display of HTML source

```
<html>
<body>
<form action="PersonQuery" method=get>
Search for:
<select name="persontype">
    <option value="student" selected>Student </option>
    <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 name="name">
<input type=submit value="submit">
</form>
</body>
</html>
```

Figure 101 - An HTML form

Moving on to **HyperText Markup Language (HTML)** is the language used to create web pages, allowing for the structuring of text, images, and other media on a webpage. **HTML pages** are enclosed in an **HTML tag**, while the body of the page is enclosed in a body tag. A table is specified by a table tag, which contains rows specified by a tr tag, and so on.

The use of **HTML tags** allows for the creation of forms, menus, and tables, making it easier for web designers to create visually appealing websites.

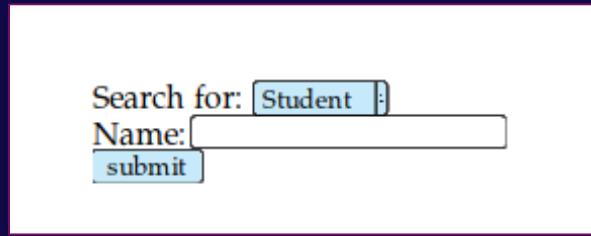


Figure 101 - Display of HTML source

But the power of the Web does not stop there, as Web servers provide an array of features beyond the mere transfer of documents. By executing programs with user-supplied arguments, a **Web server** can act as an intermediary to provide access to a variety of information services. The **Common Gateway Interface (CGI)** standard defines how the **Web server communicates** with application programs, allowing for the creation of new services by installing an application program that provides the service.

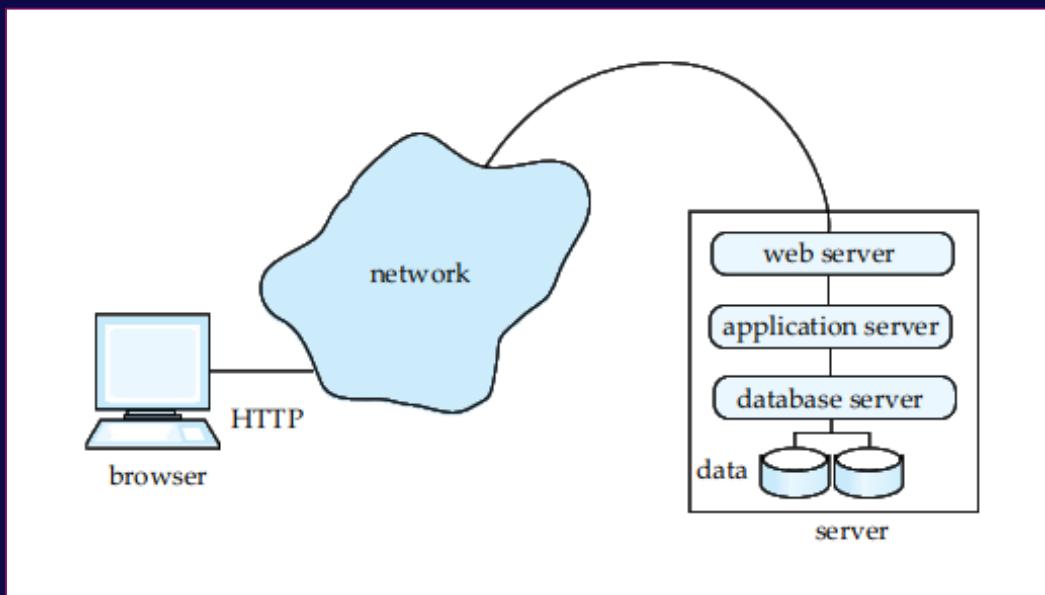


Figure 102 - Three-layer Web application architecture

As we marvel at the sheer magnitude of the **World Wide Web**, we must also acknowledge the intricate technology that makes it all possible. With the use of **URLs**, **HTML**, and **Web servers**, we are able to create and access a seemingly infinite world of information at our fingertips.

Servlets and JSP

In a two-layer web architecture, *Java* programs can be loaded into the web server itself to implement an application programming interface for communication between the **server** and the **application program**. This is known as the *Java* servlet specification. The *HttpServlet* class in *Java* is responsible for implementing the **servlet API specification**, and **subclasses** of this class are used to define specific functions. A *servlet* is essentially a *Java* program that implements the servlet interface, with the task of processing **HTTP requests**, **accessing databases**, and **dynamically generating HTML** pages for client browsers.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PersonQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD> <TITLE> Query Result </TITLE> </HEAD> ");
        out.println("<BODY> ");

        String persontype = request.getParameter("persontype");
        String number = request.getParameter("name");
        if(persontype.equals("student")) {
            ... code to find students with the specified name ...
            ... using JDBC to communicate with the database ..
            out.println(" <table BORDER COLS=3> ");
            out.println(" <tr> <td>ID </td> <td>Name: </td> " +
                       " <td>Department</td> </tr> ");
            for(... each result ...){
                ... retrieve ID, name and dept.name
                ... into variables ID, name and deptname
                out.println(" <tr> <td> " + ID + " </td> " +
                           " <td> " + name + " </td> " +
                           " <td> " + deptname + " </td> </tr> ");
            };
            out.println(" </table> ");
        }
        else {
            ... as above, but for instructors ...
        }
        out.println(" </BODY> ");
        out.close();
    }
}
```

Figure 103 - Example of servlet code

To **generate dynamic responses** to **HTTP requests**, servlets are commonly used to access inputs provided through **HTML forms**, **apply business logic**, and **generate HTML output** to be sent back to the browser. A servlet example, *PersonQueryServlet*, demonstrates the form in the figure, with the *servlet code responsible* for extracting values of the parameter type and number using *request.getParameter()*, running a query against a database, and returning the results of the query to the requester by outputting them to the *HttpServletResponse object response*.

In dealing with the stateless interaction between a browser and **a web server**, the *servlet API* provides a **method of tracking sessions** and **storing session information**. Using *getSession(false)* retrieves the *HttpSession object* corresponding to the browser that sent the request, allowing the server to ask the client to return a cookie with a specified name. If the client returns a value that does not match any ongoing session, then the request is not part of an ongoing session, with *getSession()* returning a null value, and the servlet directing the user to a login page.

The servlet life cycle begins with loading the servlet class, initializing the servlet by invoking the *init() method* once, and then handling the service requests using the *service() method*. Multiple requests can be handled in parallel, with each request resulting in a new thread within which the call is executed. The *destroy() method* is then invoked to take the servlet out of service, destroying the servlet instance once it is no longer needed.

Thus, *Java servlets* and *JSPs* provide a powerful tool for building web applications, allowing for dynamic processing of *HTTP requests*, session tracking, and a robust life cycle.

The emergence of client-side scripting has transformed the static, **text-and-graphic-based Web pages** of yesteryear into dynamic and interactive platforms for communication and commerce. By **embedding program code** within documents, web developers have opened up a world of possibilities for engaging with users beyond the limited confines of HTML and forms. This allows Web pages to perform activities such as animation and validation checks in real-time, all while executing programs on the client side, thereby speeding up interaction significantly. However, with great power comes great responsibility, and if the design of the system is done carelessly, program code embedded in a Web page can perform malicious actions on the user's computer.

Java, with its platform-independent “*byte-code*”, became a popular solution for executing programs on users’ computers, thanks to its safe mode that restricts programs from performing destructive actions. Similarly, simpler scripting languages, such as **JavaScript**, have emerged as the go-to solution for enriching user interaction while providing protection similar to **Java**. **JavaScript**, in particular, has become ubiquitous in the current generation of Web interfaces, allowing web developers to construct sophisticated user interfaces that perform error checks on user input, dynamically modify HTML code, and communicate asynchronously with the Web server.

```
<html>
<head>
<script type="text/javascript">
    function validate() {
        var credits=document.getElementById("credits").value;
        if (isNaN(credits)|| credits<=0 || credits>=16) {
            alert("Credits must be a number greater than 0 and less than 16");
            return false
        }
    }
</script>
</head>

<body>
<form action="createCourse" onsubmit="return validate()">
    Title: <input type="text" id="title" size="20"><br />
    Credits: <input type="text" id="credits" size="2"><br />
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

Figure 104 - Example of JavaScript used to validate form input

Although the **JavaScript language** has been standardized, differences between browsers, especially in the details of the **Document Object Model (DOM)** model, can create problems for developers. To mitigate this, web developers often use **JavaScript libraries**, such as **Yahoo’s YUI library**, that provide browser-independent functions that are optimized for different platforms.

The emergence of **client-side scripting** has revolutionized the way we interact with the Web, allowing us to create dynamic and interactive platforms for communication and commerce. While the power of this technology is undeniable, it is crucial to use it responsibly and with care, to avoid malicious attacks and *protect users’ privacy and security*.

Application Architectures

In the intricate world of software development, the architecture of large applications is often broken down into several layers to manage their complexity. The **presentation** or **user interface layer**, responsible for user interaction, is often conceptually divided further into distinct layers, based on the **model-view-controller architecture**. This approach allows for the creation of different views of an application, depending on the software or device used to access it.

- The **business-logic layer**, which provides a high-level view of data and actions on data, is where entities and actions are abstracted. For example, in an application designed for managing a university, the business-logic layer may provide abstractions for entities such as students, instructors, courses, and sections. In addition, workflows are included in the business logic, describing how a particular task is handled when it involves multiple participants. Error situations are also dealt with in this layer.
- The **data access layer provides** the interface between the business-logic layer and the underlying database. This layer is particularly important when using an object-oriented language to code the business-logic layer, as the data-access layer is responsible for mapping the object-oriented data model used by the business logic to the relational model supported by the database. Object-relational mapping, which automates this mapping process, is used to create in-memory objects on demand, as well as to store updated objects back as relations in the database.

Various systems have been developed to **implement object-relational mapping**, including **Hibernate**, which is widely used for mapping from Java objects to relations. A mapping file specifies how each Java class is mapped to one or more relations, with information about the database specified in a properties file.

This automated approach to mapping data models not only reduces the potential for errors but also increases efficiency, making the task less cumbersome.

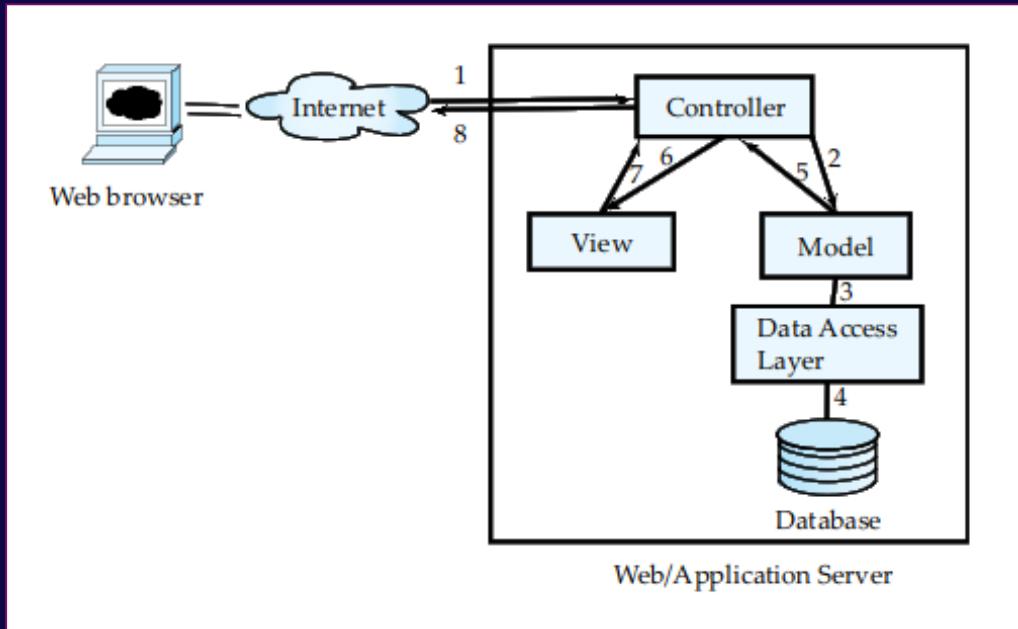


Figure 105 - Web application architecture

The architecture of web applications is critical, as it impacts not only the development process but also the overall performance of the application. By breaking down the application into various layers, such as the **presentation layer**, the **business logic layer**, and the **data access layer**, developers can manage complexity more effectively, ensuring that business rules are satisfied and workflows are followed correctly.

Rapid Application Development

In the realm of web development, the creation of user interfaces can be a significant challenge, requiring considerable programming effort to construct elements such as **menus**, **forms**, and **result displays**. However, several techniques have emerged to reduce the burden of building these interfaces.

One such approach is the provision of **pre-built libraries** of functions to generate user-interface elements with minimal programming, thereby allowing developers to focus more on business logic and database access. Additionally, **drag-and-drop** features in integrated development environments enable developers to drag user-interface elements from a menu into a design view of a page, with the **IDE generating code** that invokes library functions to create the element.

Furthermore, **automatic code** generation from a declarative specification can simplify user interface development by reducing the amount of manual coding required. These approaches have been part of **Rapid Application Development (RAD)** tools for many years and are now widely used for creating web applications as well.

Tools such as **Oracle Forms**, **Sybase PowerBuilder**, and **Oracle Application Express (APEX)** have been designed for the rapid development of interfaces for database applications, while others such as **Visual Studio** and **Netbeans VisualWeb** support several features designed for the rapid development of web interfaces for database-backed applications.

To minimize the amount of code required to build user interfaces, many **HTML constructs** are generated using appropriately defined functions rather than being written as part of the code of each web page. For instance, menus can be generated from data in the database using a function that executes a database query and populates the menu using the query result. Similarly, forms that require validation can be generated using functions that output JavaScript code to perform validation at the browser.

Displaying a set of results from a query is a common task for many database applications, and a generic function can be built to take an SQL query as an argument and display the tuples in the query result in a tabular form.

Pagination of results can also be implemented to handle situations where the query result is very large.

Acme Supply Company, Inc. Quarterly Sales Report			
Period: Jan. 1 to March 31, 2009			
Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	1,500,000
	Computer Software	500,000	
	All categories		
South	Computer Hardware	200,000	600,000
	Computer Software	400,000	
	All categories		
		Total Sales	2,100,000

Figure 106 - A formatted report

While there is no widely used standard **Java API** for carrying out user-interface tasks, tools such as **JavaServer Faces (JSF)** and **Microsoft's Active Server Pages (ASP)** provide a variety of controls that simplify the construction of web interfaces. These controls, such as drop-down menus and list boxes, can be associated with a **DataSet object**, which is similar to a **JDBC ResultSet object** and is typically created by executing a query on the database.

Moreover, validator controls can be added to form input fields to specify validity constraints, and error messages to be displayed on invalid input can be associated with each validator control.

Overall, these techniques have revolutionized the development of user interfaces for web applications, greatly reducing the effort required and making web development more accessible to developers of all skill levels.

Application Performance

Web developers face a **Herculean task** when it comes to ensuring fast response times for millions of users accessing their websites from all corners of the world. The challenge is compounded by the need to handle thousands of requests per second, if not more, for the most popular sites. To achieve this, developers employ a range of techniques to speed up processing and reduce overhead, such as caching and parallel processing.

- One technique that has proved effective in **reducing overhead is connection pooling**. With connection pooling, the application server creates a pool of open **JDBC/ODBC connections** and makes them available to the code servicing user requests. This eliminates the need to open a new connection to the database for each request, which can take several milliseconds and quickly become a **bottleneck**. The connection pool manager ensures that the maximum number of concurrent connections supported by the database is not exceeded, and closes any open connections that have not been used for a while.
- Web developers must also be vigilant about closing open connections to avoid overwhelming the database. Failure to do so can lead to the database reaching its limit for concurrent open connections, which only becomes apparent under heavy usage.
- Another effective technique for reducing costs is **caching query results** and **web pages**. By caching query results, developers can avoid repeated communication with the database for identical queries, greatly reducing overhead. Similarly, caching the final web page sent in response to a request can also eliminate the need to recompute the page when a new request with the same parameters comes in. Caching can be done at the fragment level or for complete web pages.
- **Parallel processing** is another popular technique used to handle heavy loads. This involves using multiple application servers to handle user requests in parallel, with a **web server** or **network router** used to route each request to a different application server. All requests from a particular client session must go to the same application server, to maintain the session state. However, the database is shared by all application servers, which can become a bottleneck if not properly managed. To avoid this, developers must minimize the number of requests to the database and employ parallel database systems where necessary.

Application Security

In today's technology landscape, ensuring application security has become a matter of utmost importance. With the increasing threat of hackers and malicious attacks, organizations must take proactive measures to safeguard their applications from potential vulnerabilities.

Beyond the conventional SQL authorization protocols, **application security** must tackle several security issues that arise due to poorly written application code. The initial point where security must be enforced is within the application itself. To achieve this, applications must authenticate users and restrict users to only authorized tasks.

Despite the **database system's robust security measures**, an application's security can still be compromised due to several security loopholes. In this regard, the section first sheds light on the loopholes that hackers can exploit to bypass authentication and authorization checks. Subsequently, the section describes techniques for **secure authentication, fine-grained authorization, and audit trails** that aid in recovering from unauthorized access and erroneous updates. Finally, the section concludes by highlighting issues concerning data privacy.

Among the most significant security threats that applications face are **SQL injection attacks**, whereby hackers manipulate an application into executing an SQL query crafted by the attacker. To avoid such attacks, the best practice is to employ prepared statements that automatically add escape characters, making it impossible for user-supplied quotes to terminate strings. Additionally, applications that create queries dynamically must ensure that the input string is one of the allowed values to avoid this kind of SQL injection.

Another significant security concern is the **potential for cross-site scripting (XSS) attacks**, which allow malicious users to enter client-side scripting languages instead of valid names or comments, potentially executing scripts that send private cookie information back to the attacker or execute an action on a different web server. To prevent such attacks, organizations must ensure that their applications incorporate measures such as proper input validation and output encoding to eliminate potential vulnerabilities.

Beyond the conventional **SQL authorization protocols**, application security must tackle several security issues that arise due to poorly written application code. The initial point where security must be enforced is within the application itself. To achieve this, applications must authenticate users and restrict users to only authorized tasks.

Encryption and Its Applications

In a world that is increasingly reliant on technology, encryption has become a critical component for securing sensitive data. From credit-card numbers to fingerprints and social security numbers, businesses and organizations must safeguard this information from cybercriminals who seek to exploit it for personal gain.

Encryption, the process of transforming data into an unreadable form that can only be reversed through decryption, has a long history dating back to the transmission of messages between senders and recipients who shared a secret key. However, as technology has advanced, encryption has become more sophisticated and is now widely used in a range of applications, such as data transfer on the Internet and cellular phone networks.

But not all encryption techniques are created equal. Simple encryption methods, such as substituting each character with the next letter in the alphabet, may be vulnerable to hacking by unauthorized users. A strong encryption technique, on the other hand, must be relatively simple for authorized users to encrypt and decrypt data, and it must rely on the encryption key - rather than the algorithm itself - to keep data secure.

One of the most widely-used encryption techniques today is the **Advanced Encryption Standard (AES)**, a symmetric-key algorithm that operates on a *128-bit block* of data at a time, with encryption keys that can be *128, 192, or 256 bits* in length. AES was adopted as an encryption standard by the U.S. government in *2000* and is known for its robustness in safeguarding sensitive data.

Despite the effectiveness of encryption, however, it is only as secure as the mechanism used to transmit the encryption key to authorized users. This vulnerability has prompted the development of alternative encryption schemes, such as public-key encryption, which uses two keys - a **public key** and a **private key** - to encrypt and decrypt data. Each user has a public key that is published and a private key that is known only to them, ensuring that unauthorized users are unable to access sensitive data.

Encryption is a critical tool for safeguarding sensitive information. As businesses and organizations continue to store more and more data, the need for effective encryption techniques has never been greater.

Digital signatures and digital certificates play a crucial role in authentication and non-repudiation in the world of cryptography. **Public-key encryption** is used to create digital signatures that act as electronic signatures for documents. The **private key** is used to sign the data, which can then be made public. Anyone can verify the signature using the **public key**, but only the person with the private key could have created the

signature. This provides a way to authenticate data and verify that it was created by the person who claims to have created it.

Digital certificates are a **two-way authentication process** used to ensure that a client is interacting with an authentic website. The public key of the site is signed by a certification agency, whose public key is widely known. The **root certification authorities' public** keys are stored in standard web browsers. The system uses a **multi-level approach** to reduce the burden of creating certificates on the root certification authorities. The digital certificate contains the name of the party to whom it was issued, the **party's public key**, and the **public key** of the certifying authority.

To verify a certificate, the encrypted text is decrypted using the public key to retrieve the name of the party and authenticate the public key for the site. Digital certificates are widely used in **HTTPS to authenticate websites** to users and prevent malicious sites from masquerading as other websites. The site provides its digital certificate to the browser, which uses the provided **public key** to **encrypt data**. The browser creates a **one-time symmetric key** after authentication, which reduces encryption costs. Digital certificates can also be used for authenticating users by submitting a digital certificate containing the **user's public key** to a site. The certificate is verified by a trusted authority, and the **user's public key** can be used to authenticate the user using a **challenge-response system**.

Digital signatures and digital certificates are crucial for **authentication** and **non-repudiation** in the world of cryptography. These techniques **use public-key encryption** and a **multi-level approach** to ensure authenticity and prevent malicious activities.

4 DATA STORAGE AND QUERYING

In the age of big data, the intricacies of database systems can sometimes be overlooked by those who only view them through a **high-level lens**. However, at their core, these systems rely on the physical storage of data as bits on various devices, such as magnetic disks or flash storage. The speed at which data can be accessed is largely determined by the characteristics of these devices, with disk access taking significantly longer than memory access.

To navigate the challenges of physical storage, the chapter of this text provides an in-depth overview of storage media, including measures to prevent data loss due to device failure. The chapter also delves into the mapping of records to files and ultimately to bits on a disk.

As many queries only reference a small percentage of records within a file, indices play a crucial role in facilitating speedy data retrieval without the need to scan all records. The chapter explores various types of indices utilized in database systems, with an example provided in the text for human use.

Breaking down complex user queries into smaller, more manageable operations. The text outlines algorithms for implementing individual operations and explains how these operations are executed in unison to process a query.

Given the range of options available for processing a given query, the chapter introduces the concept of query optimization. This process seeks to identify the most **cost-effective method** for evaluating a query, with varying approaches and costs to consider. In sum, these chapters provide a comprehensive understanding of the essential elements underpinning database systems, from physical storage to query optimization.

4.1 Storage and File Structure

In the world of databases, the focus is often on the **high-level models** that allow users to access data with ease. Indeed, a **database system's ultimate goal** is to simplify the process of accessing and managing data, and users should not be unduly burdened with the physical details of the system's implementation.

However, in this chapter and the following ones, we dive beneath the surface to examine the methods used to implement these models and languages. We begin by exploring the fundamental properties of the underlying storage media, including disk and tape systems, before delving into the various data structures that facilitate rapid data access.

By examining different structures and their suitability for specific types of data access, we can make informed decisions about the optimal data structure to utilize based on the expected usage and the physical constraints of the particular machine in question. In short, this chapter serves as a crucial foundation for understanding the inner workings of a **database system**, and the **subsequent chapters** will continue to build on this knowledge.

Overview of Physical Storage Media

Data is the lifeblood of any enterprise, and its storage is paramount. From the fleet-footed cache to the lumbering tape, the range of storage media available to computer systems is varied and vast. Each storage type is categorized by its *speed, cost, and reliability*, and each has its unique strengths and weaknesses.

At the peak of this hierarchy sits the cache, a fleeting but costly storage medium. The **cache's minuscule size** is managed by computer system hardware, with its effects considered when designing query processing data structures and algorithms.

Next in line is the main memory, the storage medium that houses data available for processing by the **general-purpose machine instructions**. Although it can accommodate several gigabytes of data in a personal computer, it is typically inadequate for storing an entire database. Additionally, the contents of the main memory are often lost in the event of a system crash or power failure.

Enter the flash memory, a storage medium that differs from the main memory in its **non-volatile nature**. Stored data persists even when power is lost, thanks to two **types of flash memory, NAND and NOR**. **NAND** flash is more cost-effective and accommodates larger storage capacities for the price. It is widely utilized in portable devices like *cameras, music players, and cell phones*, and increasingly in laptop computers.

Flash memory is increasingly replacing magnetic disks, serving as **solid-state drives** that store moderate amounts of data. Its cost per byte is lower than that of main memory, while its larger storage capacity is a boon to system performance. Flash memory is also used in server systems to cache frequently used data, enabling faster access than disks and more storage capacity than main memory.

Magnetic disk storage remains **the go-to medium for long-term online storage** of data. As the primary medium for storing entire databases, magnetic disks must transfer data to the main memory for processing. Magnetic disks are available in sizes ranging from **80 gigabytes** to **1.5 terabytes**, with prices beginning at **\$100** for a **1 terabyte disk**. Disk storage survives system crashes and power failures, although disk storage devices themselves may sometimes fail and destroy data.

Optical storage, on the other hand, is exemplified by the compact disk (CD) and the digital video disk (DVD), with optical data storage that is read by a laser. CDs can hold about **700** megabytes of data and have an **80**-minute playtime, while DVDs hold between **4.7** and **8.5** gigabytes of data per disk side. Blu-ray DVDs, with **27** gigabytes of capacity per layer, have a double-layer disk capacity of **54** gigabytes.

Tape storage, although slower than disks, is cheaper and well-suited to backup and archival data storage. Tapes can store between **40** and **300** gigabytes of data, making them ideal for exceptionally large data collections such as satellite data, which can easily top hundreds of terabytes or even multiple petabytes.

Data storage media is a complex and dynamic field that evolves rapidly. With each medium having its unique benefits, choosing the appropriate storage medium for a specific need is critical.

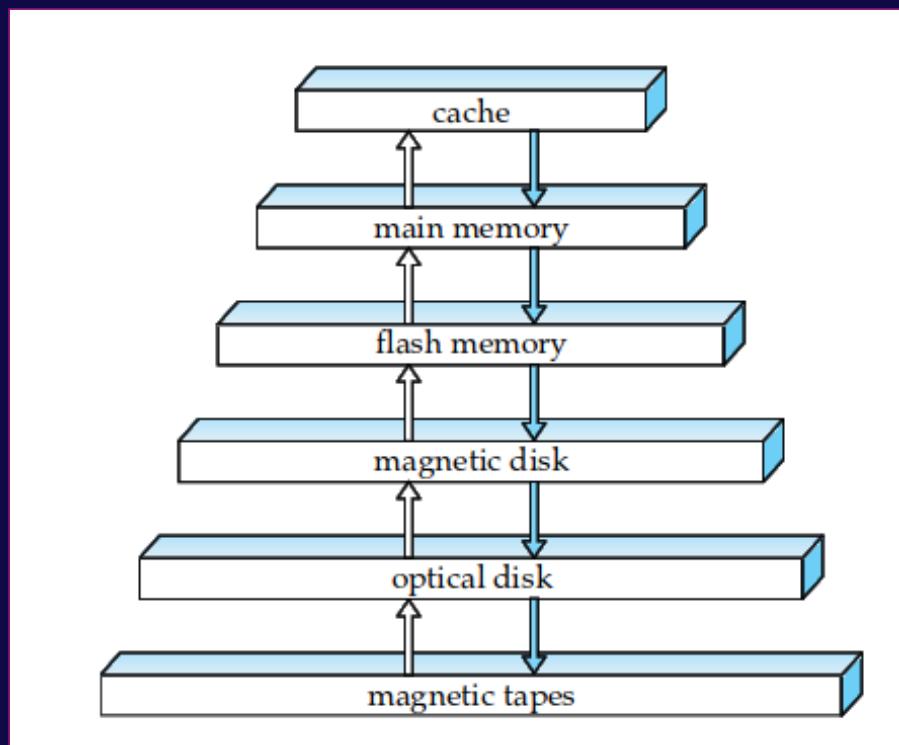


Figure 107 - Storage device hierarchy

Magnetic Disk and Flash Storage

Magnetic disks have long been the primary medium for secondary storage. However, as the storage requirements of large applications continue to outpace the growth rate of disk capacities, there has been a growing need for alternative solutions. Enter flash-memory storage, whose sizes have expanded exponentially in recent years, positioning it as a potential competitor to magnetic disk storage in select applications.

In order to better understand the underlying physical characteristics of disks, one must take a closer look at their construction. Disk platters, each having a flat, circular shape, are composed of rigid metal or glass, and possess two surfaces covered with a magnetic material, on which data is recorded. When in use, the platters are spun by a drive motor at a constant high speed, typically ranging from **60** to **120** revolutions per second. To read or write data, a read-write head is positioned just above the platter surface, and the disk surface is divided into tracks and sectors. In today's disks, sector sizes are typically **512 bytes**, with around **50,000** to **100,000** tracks per platter, and **1** to **5** platters per disk.

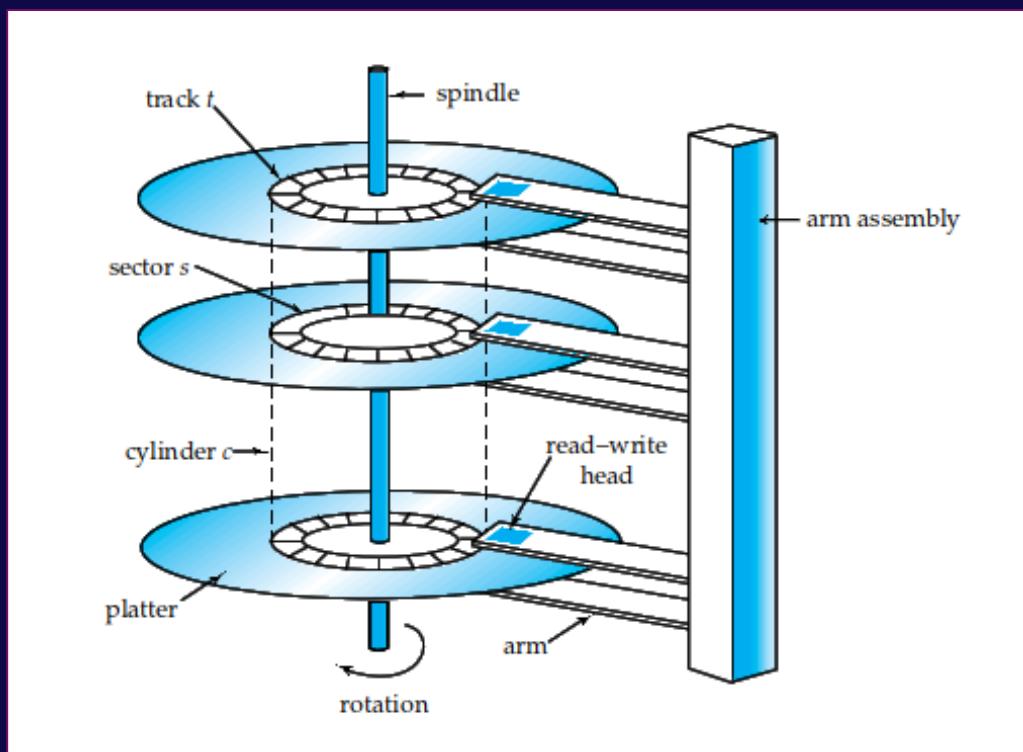


Figure 108 - Moving head disk mechanism

The read-write head, which magnetically stores information on a sector by reversing the direction of magnetization of the magnetic material, moves across the platter to access different tracks. All of the **read-write heads** on a disk, which may contain many platters, are mounted on a single assembly called a disk arm, and move together, creating the **so-called head-disk assembly**. The read-write heads being synchronized in this manner means that the i th tracks of all the platters together are referred to as the i th cylinder. In order to increase recording density, the heads are kept as close as possible to the disk surface, typically floating only microns above it.

However, head crashes remain a serious problem. If the head comes into contact with the disk surface, it can scrape off the recording medium, effectively destroying any data present. In older-generation disks, this often led to the medium becoming airborne and coming between the other heads and their platters, causing additional crashes that could result in the failure of the entire disk. Current-generation disk drives utilize a thin film of magnetic metal as the recording medium, making them much less susceptible to failure from head crashes than older oxide-coated disks.

To interface between the computer system and the disk drive unit, a disk controller is employed. This crucial component accepts high-level commands to read or write data and initiates actions such as moving the disk arm to the right track and performing the read or write operation. Disk controllers attach checksums to each sector that is written, which are computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum. If an error is detected, the controller will retry the read several times before signaling a failure. Additionally, if a sector is found to be damaged, the controller will remap the sector to a different physical location.

Disk are connected to a computer system via a **high-speed interconnection**, and there are several interfaces for connecting disks to computers, including **SATA** and **SCSI**. However, as the field of data storage technology continues to rapidly evolve, one can be certain that these interfaces will continue to evolve as well.

A clear distinction arises between **sequential** and **random access patterns**. In the former, successive requests for data are made in an orderly fashion, with contiguous blocks on the same track or on adjacent tracks being accessed. The first request in such a pattern may require a disk seek, but subsequent requests would either not require a seek or necessitate a seek to an adjacent track, which is much faster than a desire to a farther track.

On the other hand, in a random access pattern, successive requests are for blocks that are randomly located on the disk. Each such request would require a disk seek, and the number of random block accesses that can be processed by a single disk in a second depends on the seek time, which is typically about **100** to **200** accesses per second. Furthermore, since only one block of data is read per seek, the transfer rate is considerably lower with a random access pattern than with a sequential access pattern.

Several techniques have been developed to enhance the speed of access to blocks, including buffering, read-ahead, scheduling, file organization, and nonvolatile write buffers.

Buffering involves temporarily storing the blocks read from the disk in an in-memory buffer, which can be used to satisfy future requests. Read-ahead entails reading consecutive blocks from the same track into an **in-memory buffer**, even if there is no pending request for those blocks. Such **read-ahead** is useful for sequential access, where it ensures that many blocks are already in memory when they are requested, thereby minimizing the time wasted in disk seeks and rotational latency per block read.

Scheduling algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed, while file organization involves organizing blocks on disk in a way that corresponds closely to the way data is expected to be accessed.

Finally, nonvolatile write buffers utilize **nonvolatile random-access memory (NVRAM)** to speed up disk writes dramatically. **Battery-backed-up RAM** or flash memory can be used to implement **NVRAM**, and the contents of **NVRAM** are not lost in a power failure.

In update-intensive database applications such as **transaction-processing systems**, the speed of disk writes heavily influences performance, making nonvolatile write buffers an indispensable tool in boosting performance.

RAID

As the world's data storage needs continue to skyrocket, with particular emphasis on **web**, **multimedia**, and **database applications**, there has been a pressing demand for faster and more reliable storage solutions. **Redundant arrays of independent disks (RAID)** are an innovative and powerful technology that has emerged to address these needs.

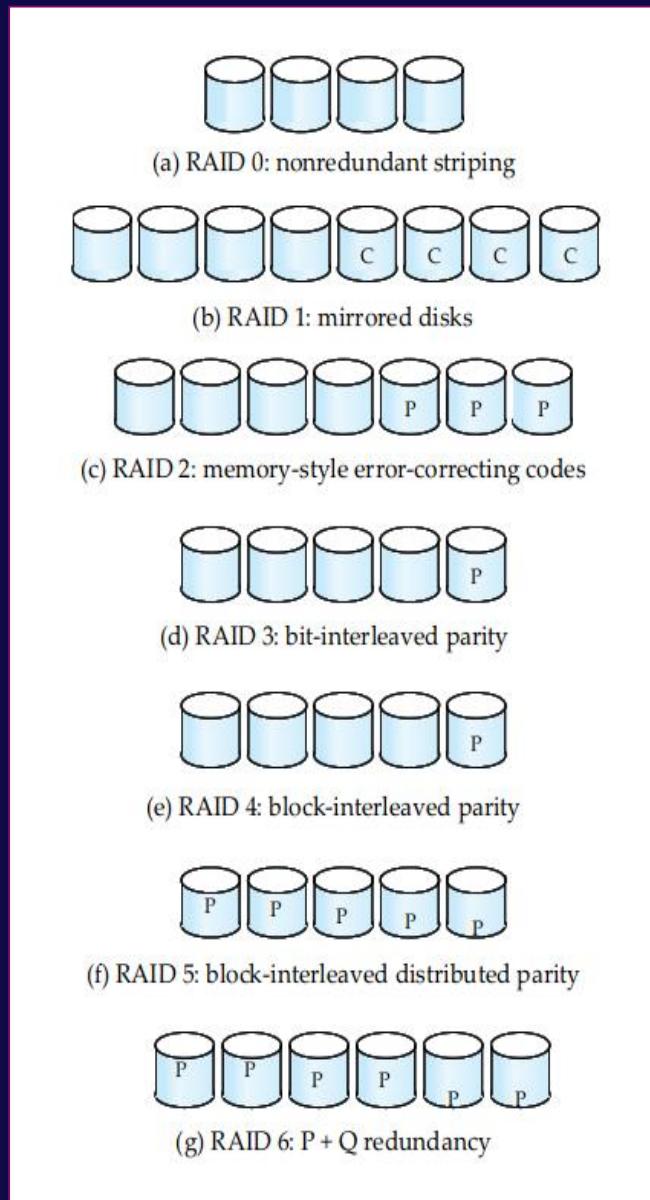


Figure 109 - RAID levels

By allowing for a large number of disks to be operated in parallel, **RAID systems** offer unparalleled read/write speeds that are essential for handling data-intensive applications. Performing independent reads and writes in parallel further enhances the system's overall performance.

However, their remarkable reliability may be the most impressive feature of **RAID systems**. Disk failure is an ever-present danger that can result in the loss of large amounts of valuable data. But it introduces redundancy, storing extra information that is not ordinarily needed but can be used to rebuild lost information in the event of a disk failure. Mirroring, the most basic **RAID technique**, duplicates every disk and carries out every write on both disks, so if one disk fails, the data can still be retrieved from the other disk. This redundancy ensures that data are never lost and the mean time to failure is greatly extended.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Figure 110 - Pattern

RAID systems offer an innovative and powerful storage solution that is essential for handling the ever-increasing data storage demands of modern society. By allowing for high-performance parallel reads and writes and introducing redundancy to increase reliability, **RAID technology** ensures that data are always safe and accessible.

The choice of **RAID** level is a critical decision that can significantly impact the performance and reliability of a system. With the monetary cost of extra disk storage requirements and the number of **I/O operations** needed for optimal performance being key factors, the decision of whether to opt for the more popular **RAID level 1** or the **storage-efficient RAID level 5** is a tough one that requires careful consideration of the unique needs of each application.

As **disk-storage capacities** continue to grow at an **unprecedented rate**, the **cost per byte** has fallen dramatically, making mirroring a more viable option for many existing database applications with moderate storage requirements. However, access speeds have not kept pace with this growth, and the **number of I/O operations** required per second has skyrocketed, particularly for web application servers. This puts a premium on choosing a **RAID level** that can deliver the necessary write performance without sacrificing reliability or storage efficiency.

To this end, **RAID system designers** must make several other crucial decisions, including the number of disks in an array and the number of bits protected by each parity bit. While a higher number of disks can boost data-transfer rates, it also raises the system's cost, while increasing the number of bits protected by a parity bit lowers the space overhead due to parity bits but heightens the risk of data loss if a second disk fails before the first one is repaired.

Hardware is another vital consideration in choosing a **RAID implementation**, with **hardware RAID systems** offering several benefits over **software RAID**.

For instance, special-purpose hardware can use **nonvolatile RAM** to record writes before they are performed, ensuring that incomplete writes are completed in case of power failure. Additionally, good **RAID controllers** can perform scrubbing to minimize the chance of data loss caused by latent failures or bit rot.

All in all, choosing the right **RAID level** is a critical decision that requires careful weighing of several factors, including monetary cost, performance requirements, and data safety. With the right combination of **RAID level** and hardware *implementation, data storage and retrieval systems* can deliver the high performance and reliability that modern applications demand.

Tertiary Storage

A peculiar tertiary storage emerges - one that is capable of hosting voluminous amounts of data. Indeed, such data often necessitates the deployment of secondary storage, but when even the latter proves insufficient for the task, tertiary storage provides a solution.

Two prime examples of tertiary storage media come to mind - the magnetic tape and the optical disk. Both serve unique purposes in the storage hierarchy, but in this exposition, we shall focus on the latter.

Compact disks (CDs) have become a popular means of distributing software and multimedia data, including audio and images, and other electronically published content. These disks have a storage capacity ranging from **640** to **700** megabytes, and they can be mass-produced at a low cost. However, digital video disks (DVDs) have replaced **CDs** in applications requiring higher data volumes. **DVD-5** format disks can store up to **4.7** gigabytes of data on a single recording layer, while **DVD-9** format disks can store up to **8.5** gigabytes of data on two recording layers. Double-sided disks, such as the **DVD-10** and **DVD-18 formats**, can store even more data - up to **9.4** gigabytes and **17** gigabytes, respectively.

Blu-ray DVDs have set a new benchmark, boasting a capacity of **27** to **54** gigabytes per disk. However, optical disks' seek times are much longer than those of magnetic disks due to their relatively heavier head assembly. They also have lower rotational speeds, although faster CD and DVD drives rotate at speeds comparable to lower-end magnetic disk drives. Optical disks store more data in outside tracks and fewer data in inner tracks, like magnetic disk drives. Their data transfer rates are lower than those of magnetic disks, but they have a transfer rate characteristic of $n\times$, indicating that they can transfer data at n times the standard rate.

Optical disks come in two forms - the record-once version (**CD-R, DVD-R, and DVD+R**) and the multiple-write **version (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)**. The former is popular for distributing data and archival storage since it cannot be overwritten and is ideal for storing unalterable data, such as audit trails. The latter is also useful for archival purposes.

To achieve even more massive storage capacity, jukeboxes come in handy. These devices store hundreds of optical disks that can be loaded automatically onto one of several drives. The system's aggregate storage capacity can be in the terabytes, with disk load/unload time being much longer than disk access times.

File Organization

In the realm of database management, the organization of data is crucial for ensuring efficient and effective data retrieval. To this end, files are the basic construct of operating systems and are essential for mapping a database into a series of files that reside permanently on disks. Each file is organized logically as a sequence of records, which are then mapped onto disk blocks.

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

Figure 111 - Example

In turn, each file is partitioned into fixed-length storage units known as blocks. These blocks serve as the units of storage allocation and data transfer, and most databases use block sizes of **4** to **8** kilobytes by default, although larger block sizes can be specified for specific database applications.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 112 - File containing instructor records

It is important to note that each record must be entirely contained within a single block, and no record should be larger than a block. Although there are large data items such as images that can exceed the block size, these can be handled by storing them separately and including a pointer to the data item in the record. In addition, a relational database has tuples of different sizes. To account for this, we may use several files and store records of only one fixed length in each file. Alternatively, we can structure our files to accommodate multiple lengths for records. However, files of **fixed-length records** are easier to implement than files of **variable-length records**.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 113 - File with record 3 deleted and all records moved

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Figure 114 - File with record 3 deleted and final record moved

In dealing with fixed-length records, let us consider a file of instructor records for a university database as an example. We can allocate the maximum number of bytes that each attribute can hold, which results in an instructor record that is **53** bytes long. However, a simple approach of using the first **53** bytes for the first record, the next **53** bytes for the second record, and so on, would not work. This approach results in records crossing block boundaries, and it becomes difficult to delete records from this structure.

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 115 - File with free list after deletion of records 1, 4, and 6

To solve these problems, we allocate only as many records to a block as would fit entirely within the block, leaving any remaining bytes unused. When deleting records, we can move the final record of the file into the space occupied by the deleted record, rather than moving every record following the deleted record. This approach minimizes the number of records that need to be moved and avoids additional block accesses.

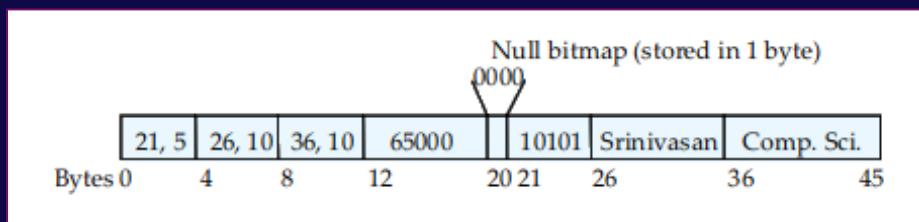


Figure 116 - Representation of variable-length record

To manage available space, we introduce an additional structure in the form of a file header that contains information about the file, including the address of the first record whose contents are deleted. By following this approach, we can organize **fixed-length records** in a way that maximizes efficiency and minimizes errors.

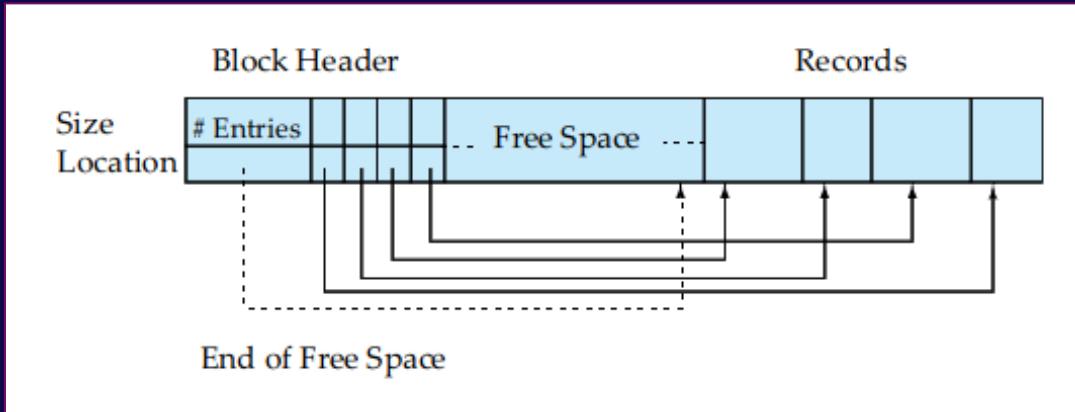


Figure 117 - Slotted-page structure

Organization of Records in Files

The organization of records in files is a critical aspect of efficient data management. Different techniques are used to store records in **files**, **including heap file organization, sequential file organization, and hashing file organization**. These methods enable fast and efficient processing of records, depending on the specific requirements of a given database.

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

Figure 118 - Sequential file for instructor records

Sequential file organization, for instance, is designed to facilitate the processing of sorted records based on some search key. To achieve this, records are stored physically in search-key order, and chains of pointers are used to link them together.

```
select dept.name, building, budget, ID, name, salary  
from department natural join instructor;
```

Figure 119 - Example

While this organization permits fast retrieval of records in search-key order, it can be challenging to maintain physical sequential order as records are inserted and deleted. To address this issue, pointer chains are utilized for record deletion, while specific rules guide the insertion of new records.

The diagram illustrates a sequential file organization. A main table contains 12 records, each with five fields: ID, name, department, salary, and a pointer field. The pointer field contains the ID of the next record in sequence. An inset shows a new record being inserted at the bottom of the file. Arrows indicate the movement of pointers from the original last record to the new record, and from the new record back to its predecessor.

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

32222	Verdi	Music	48000	
-------	-------	-------	-------	--

Figure 120 - Sequential file after an insertion

<i>dept.name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

Figure 121 - The department relation

<i>ID</i>	<i>name</i>	<i>dept.name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

Figure 122 - The instructor relation

Multitable clustering file organization is another technique used to store records of different relations in the same file. In this approach, related records of different relations are stored on the same block, allowing fast retrieval of related records from different relations in a single **I/O** operation. Although this simple file structure is well suited to small database systems such as those found in embedded systems or portable devices, more complex file structures may be required for larger databases to ensure optimal performance.

Understanding the various techniques for organizing records in files is vital for the effective management of database systems. These techniques enable fast and efficient processing of records, depending on the specific requirements of a given database.

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

Figure 123 - Multitable clustering file structure



Figure 124 - Multitable clustering file structure with pointer chains

Data-Dictionary Storage

The storage of metadata, also known as "*data about data*", is crucial for any relational database system. This includes information about the schema of *relations*, *views*, *integrity constraints*, and *authorized users*. In addition, statistical data such as the number of tuples and storage methods for each relation may also be stored. The data dictionary, or system catalog, is the structure that stores all of this important metadata.

To ensure efficient access to system data, it is recommended to store metadata as relations within the database itself. However, the exact representation of this metadata must be determined by the system designers. One possible representation is shown in the figure, which is not in first normal form but is likely to be more efficient for access.

When retrieving records from a relation, the database system must consult the related metadata to find the location and storage organization of the relation. It is important to note that the storage organization and location of the relation metadata itself must be recorded elsewhere for easy retrieval.

In essence, the metadata stored in the data dictionary constitutes a miniature database that serves as the backbone for the entire relational database system.

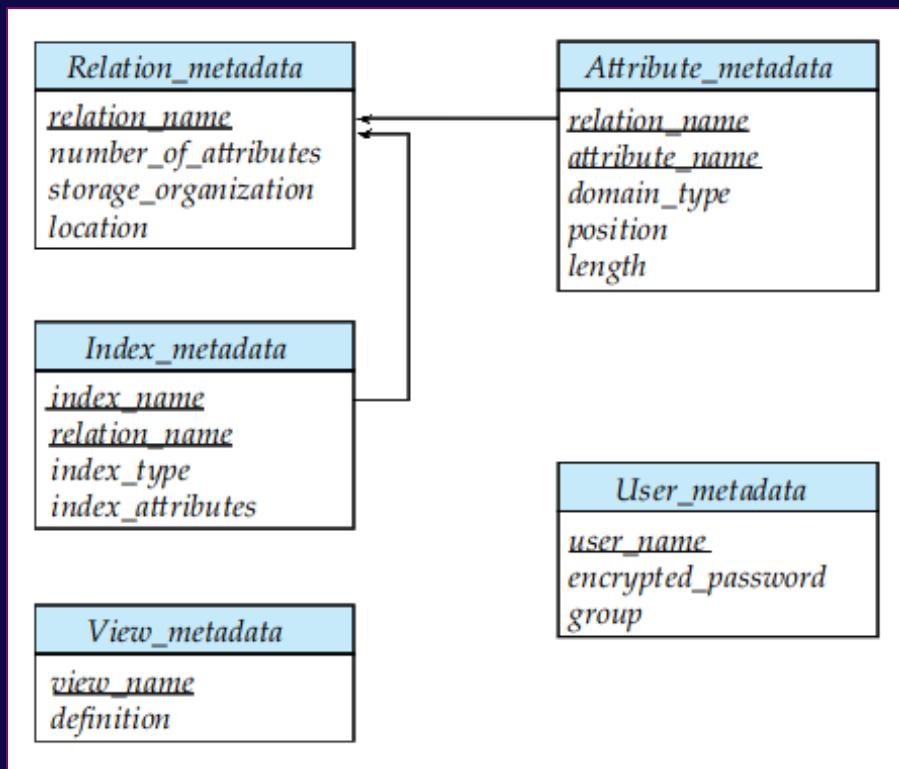


Figure 124 - Relational schema representing system metadata

Database Buffer

In the realm of database systems, the maximization of efficiency is the ultimate goal. One of the most crucial objectives in achieving this is to minimize the number of block transfers between the memory and disk. The method by which we can accomplish this is to retain as many blocks as possible in the main memory. The intention is to heighten the likelihood that when a block is accessed, it already exists in the main memory, and as a result, no disk access is necessary.

The space available in the main memory for the storage of blocks must be allocated wisely, as it is not feasible to keep all blocks in the main memory. The buffer, which is part of the main memory available for the storage of copies of disk blocks, is responsible for the allocation of buffer space, and the subsystem managing it is known as the buffer manager. Every block has a copy on disk, but the copy on disk may be an older version than the one in the buffer.

When a program in the database system requests a block from the disk, it makes a call to the buffer manager. If the block already exists in the buffer, the buffer manager delivers the address of the block in the main memory to the requester. However, if the block is not in the buffer, the buffer manager must first allocate space in the buffer for the block.

To make room for the new block, the buffer manager may have to discard another block. If the discarded block has been modified since it was last written to disk, it must be written back to the disk. The buffer manager then reads the requested block from the disk to the buffer and delivers the address of the block in the main memory to the requester.

The buffer manager's internal activities are imperceptible to the programs that request disk-block access.

To many, the buffer manager appears to be nothing more than a **virtual-memory manager**, much like those found in most operating systems. However, the size of the database could exceed the hardware address space of a machine. Furthermore, the buffer manager must employ more sophisticated methods than typical **virtual-memory management** schemes to serve the database system effectively. These include buffer replacement strategies, pinned blocks, and forced output of blocks.

```
for each tuple i of instructor do
    for each tuple d of department do
        if i[dept.name] = d[dept.name]
        then begin
            let x be a tuple defined as follows:
            x[ID] := i[ID]
            x[dept.name] := i[dept.name]
            x[name] := i[name]
            x[salary] := i[salary]
            x[building] := d[building]
            x[budget] := d[budget]
            include tuple x as part of result of instructor  $\bowtie$  department
        end
    end
end
```

Figure 125 - Procedure for computing join

The buffer replacement policy's primary goal is to reduce access to the disk by replacing blocks in the buffer. Since it is not feasible to forecast which blocks will be referenced by general-purpose programs, operating systems use **past block-reference patterns** as a predictor of future references. In contrast, database systems may have information concerning at least the **short-term future** because a user request to the database system involves multiple steps.

The **toss-immediate strategy** is an example of a buffer management strategy. It instructs the buffer manager to free the space taken up by an instructor block as soon as the final tuple has been processed, even though the block has been used recently. Another example is the **read-ahead strategy**, where the buffer manager reads blocks into the buffer in anticipation of future block accesses. The buffer manager then removes the blocks from the buffer once they are no longer required. These approaches, which utilize information about future block access, allow us to improve the **LRU strategy**.

4.2 Indexing and Hashing

In the vast realm of databases, the efficiency of queries that access only a fraction of the records is crucial. Why waste precious time and resources sifting through irrelevant tuples when we can swiftly locate the desired ones? Queries such as "*Find all instructors in the Physics department*" or "*Find the total number of credits earned by the student with ID 22201*" only require a small subset of records to be accessed, rendering a traditional read of the entire relation impractical. Therefore, additional structures, such as **indexing** and **hashing**, are implemented to enhance the system's performance and enable direct access to the desired records.

Basic Concepts

In the world of database systems, indices play a pivotal role in efficiently accessing data. An index is much like the index in a textbook, allowing us to quickly locate the pages containing the desired information. Similarly, a database index allows us to quickly find the disk block containing the data record that matches our search criteria.

However, maintaining a sorted list of data values, such as a list of student IDs, can become unwieldy in large databases with thousands of entries. This is where sophisticated indexing techniques come into play, such as ordered indexing and hashing.

Each **indexing technique** has its own strengths and weaknesses and must be evaluated based on a **variety of factors**, including **access types**, **access time**, **insertion time**, **deletion time**, and **space overhead**. While no one technique is universally superior, each is best suited to specific database applications.

It is also common to have multiple indices for a **single file**, such as indices for **author**, **subject**, and **title searches**. These attributes used to look up records in a file are known as search keys.

Database indexing is a complex and nuanced one, with a variety of techniques available to suit different needs. Whether using ordered indices or hash indices, it is crucial to consider the various factors involved in evaluating the effectiveness of each technique and to carefully choose the search keys for each index to ensure efficient and accurate data access.

Ordered Indices

The efficient retrieval of records is a key concern. To facilitate this process, an **index structure** can be employed. Such a structure is associated with a **specific search key** and **stores the values** of the search keys in sorted order. The ordered index, similar to a book's index, enables fast random access to records, and each search key is associated with the records that contain it.

In some cases, the records themselves may be stored in some sorted order, and a **file** may have **multiple indices** on **different search keys**. If the file is sequentially ordered, a clustering index is used to define the sequential order of the file. Such an index is also called a primary index and can be built on any search key, not necessarily a primary key.

Indices **whose search key** specifies an order different from the sequential order of the file are called **non-clustering** indices or secondary indices. Dense indices and sparse indices are two types of ordered indices. A **dense index** contains an index entry for every **search-key value** in the file, while a sparse index contains entries for only some of the **search-key values**.

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

Figure 126 - Sequential file for instructor records

A search in a dense index is quicker as each index record contains a **search-key value** and a pointer to the first data record with that **search-key value**. Sparse indices, on the other hand, can only be used if the relation is stored in the sorted order of the **search key**. In such cases, each index entry contains a **search-key value** and a pointer to the first data record with that **search-key value**. To locate a record, we find the index entry with the largest **search-key value** that is less than or equal to the **search-key value** for which we are looking.

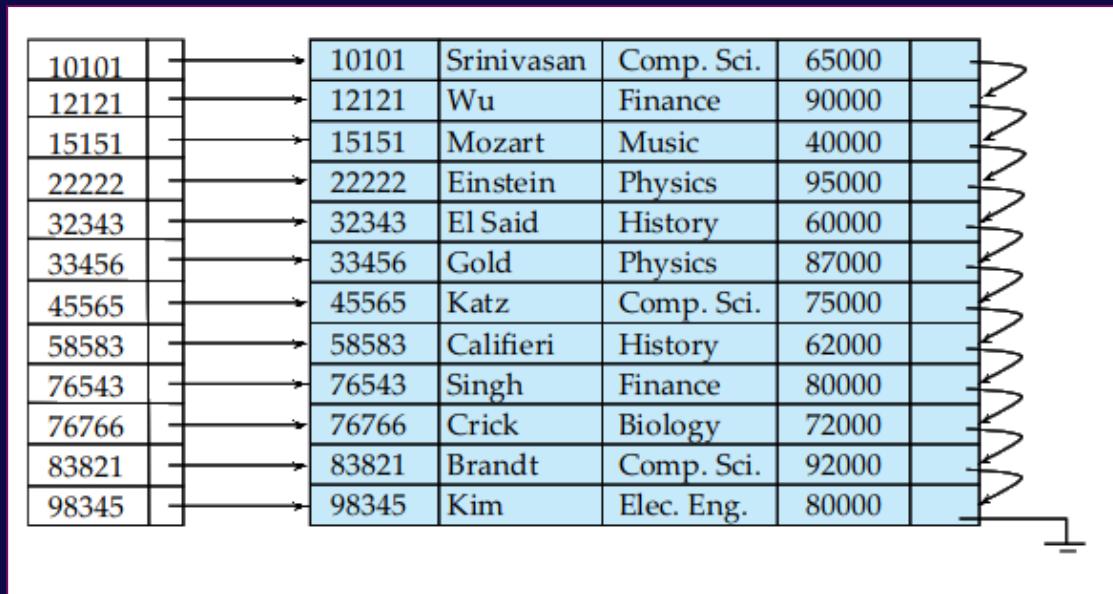


Figure 127 - Dense index

The use of ordered indices is crucial for applications that require both sequential processing of an entire file and random access to individual records. By utilizing such structures, we can expedite record retrieval and optimize database performance.

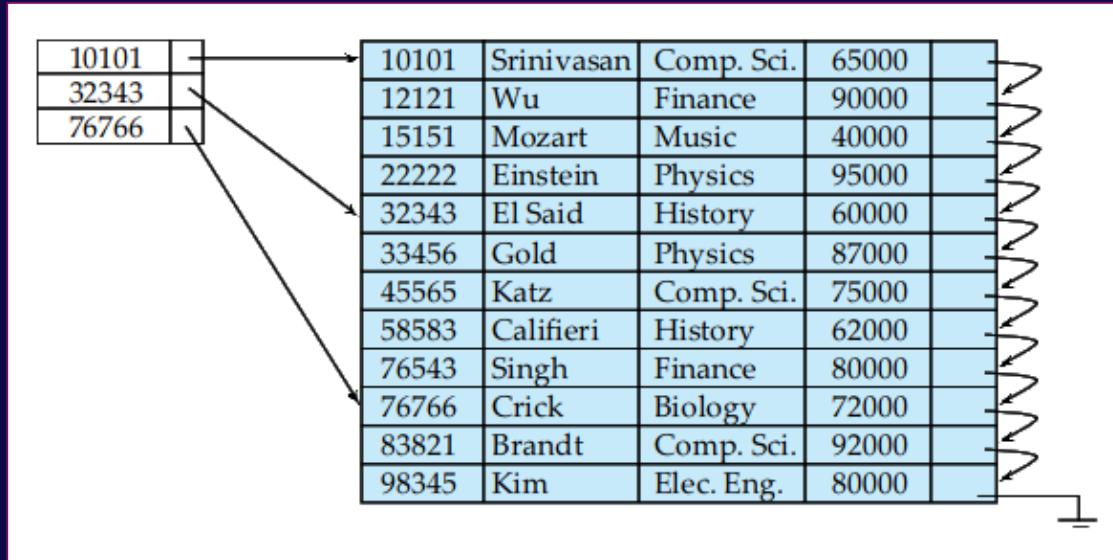


Figure 128 - Sparse index

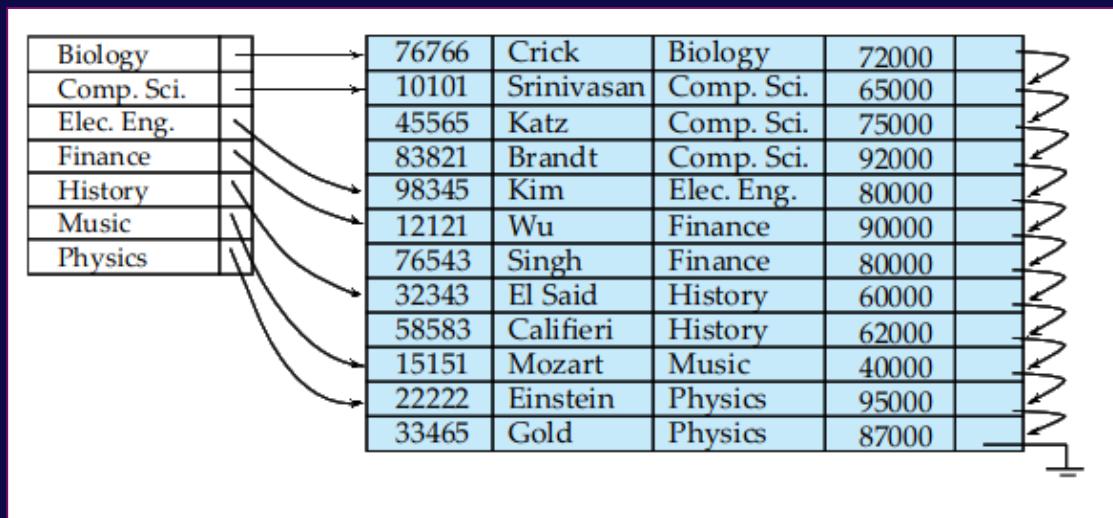


Figure 129 - Dense index with search key dept name

Even the most efficient search mechanisms can be brought to their knees by the sheer size of **an index file**. Such files, stored as **sequential files on disk**, can become unwieldy and **time-consuming** to search as they grow into the **gigabyte range**. This can lead to frustration for users who require quick access to data.

Enter the multilevel index - a tool that promises to make searching large index files a breeze. By constructing a sparse outer index on top of an inner index, a **multilevel index allows users** to perform a search using just one index block read, as opposed to the 14 required by binary search.

But what happens when even the outer index grows too large to fit into the main memory? No problem - simply create another level of the index. In fact, the process can be repeated as many times as necessary to accommodate even the largest index files.

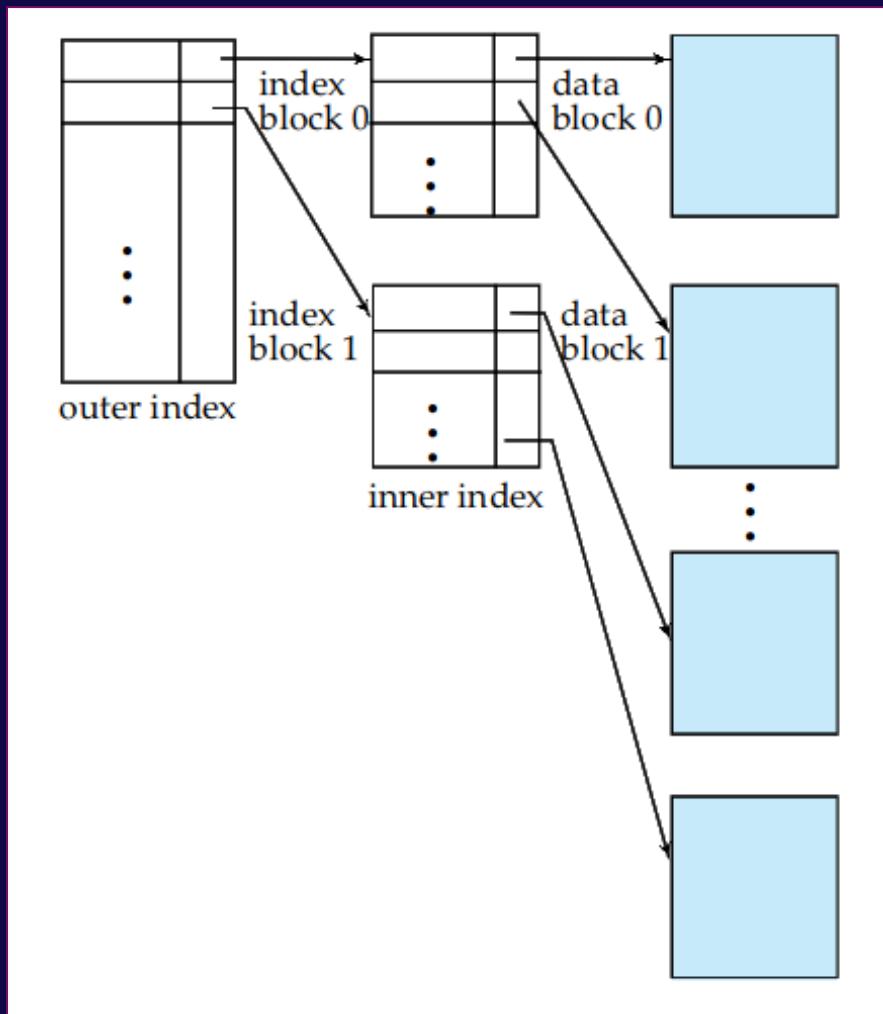


Figure 130 - Two-level sparse index

Of course, every index must be updated whenever a record is **inserted**, **deleted**, or updated in the file. This can be modeled as a deletion of the old record followed by an insertion of the new value, resulting in an index deletion followed by an index insertion. But with algorithms designed for **single-level indices**, the process of updating a **multilevel index** need not be a headache.

So whether you're managing a **small-scale database** or one that's millions of records strong, the **multilevel index** is a tool you'll want to have in your arsenal. Its power lies in its ability to perform fast and efficient searches, making it a must-have for any serious data manager.

The **concept of secondary indices** plays a critical role in the efficient retrieval of records based **on search-key values**. A **secondary index** is considered dense, meaning that every **search-key value** has a corresponding **index entry**, along with a pointer to each record in the file. This stands in contrast to a clustering index, which can afford to be sparse, as it is always possible to sequentially access a portion of the file to locate records with intermediate **search-key values**.

When a secondary index is created on a **candidate key**, it closely resembles a dense **clustering index**. However, if the search key of a secondary index is not a **candidate key**, then it must contain pointers to all records, as the records are not ordered by the search key of the **secondary index**. In this scenario, an extra level of indirection is used to implement the **secondary index**, with pointers directed to a bucket containing pointers to the file.

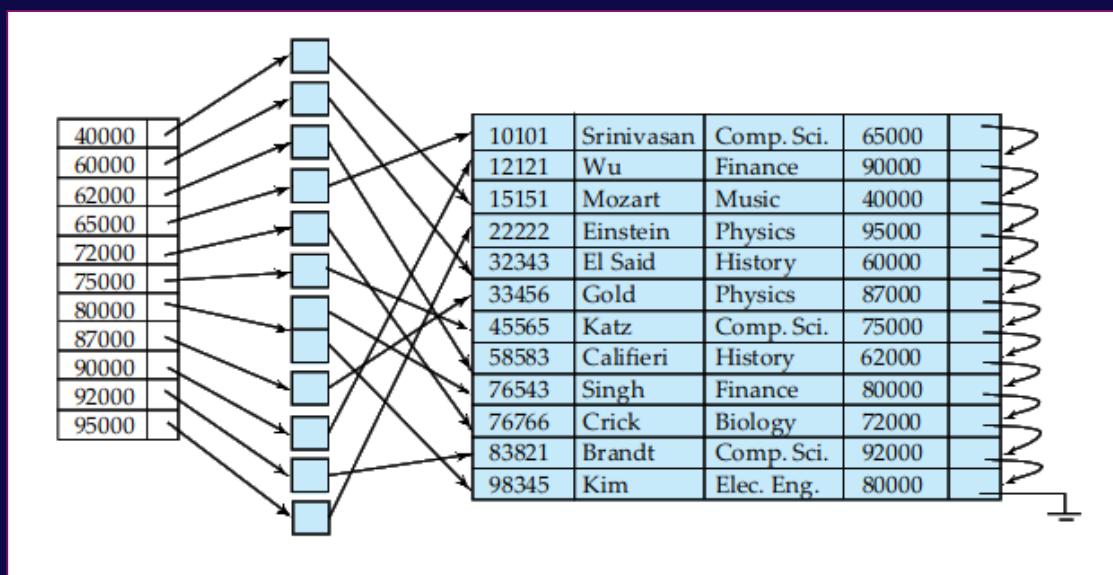


Figure 131 - Secondary index on instructor file, on noncandidate key salary

Indices on **multiple keys**, known as composite **search keys**, offer further potential for database optimization. An ordered index on a composite key allows for efficient retrieval of records based on multiple attributes, using lexicographic ordering to establish the order of search-key values.

While secondary indices can significantly improve query performance, they also impose a substantial overhead on database modification. Therefore, the decision to implement secondary indices is made on the basis of an estimate of the relative frequency of queries and modifications.

B+-Tree Index Files

As we delve into the world of database indexing, we encounter the well-known limitation of the **index-sequential file organization**: its performance degrades as the file size grows. Frequent file reorganization can alleviate this, but it's an undesirable solution.

Enter the **B+-tree index structure**, the most widely used indexing structure that maintains efficiency despite data insertion and deletion. It takes the form of a balanced tree where every path from the root to a leaf is of equal length. Each non-leaf node has between $n/2$ and n children, where n is a fixed value for a particular tree.

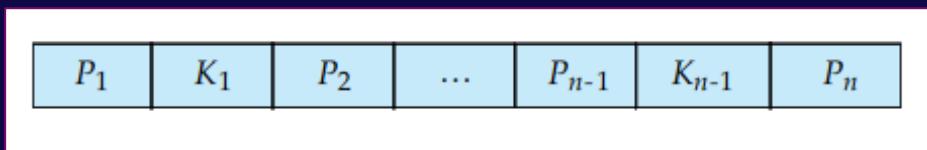


Figure 132 - Typical node of a B+-tree

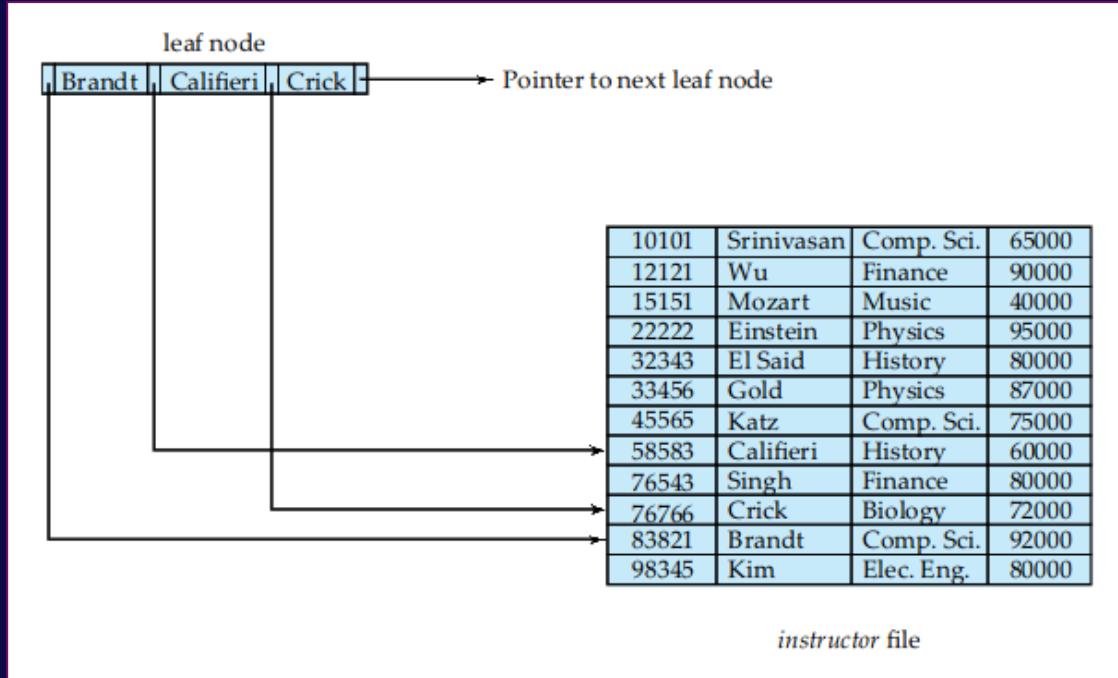


Figure 133 - A leaf node for instructor B+-tree index ($n = 4$)

Despite imposing performance and space overhead on insertion and deletion, the **B+-tree** structure provides acceptable results even for frequently modified files as the cost of file reorganization is avoided. And with nodes capable of holding up to half-empty children, wasted space is a small price to pay for the performance benefits of the **B+-tree** structure.

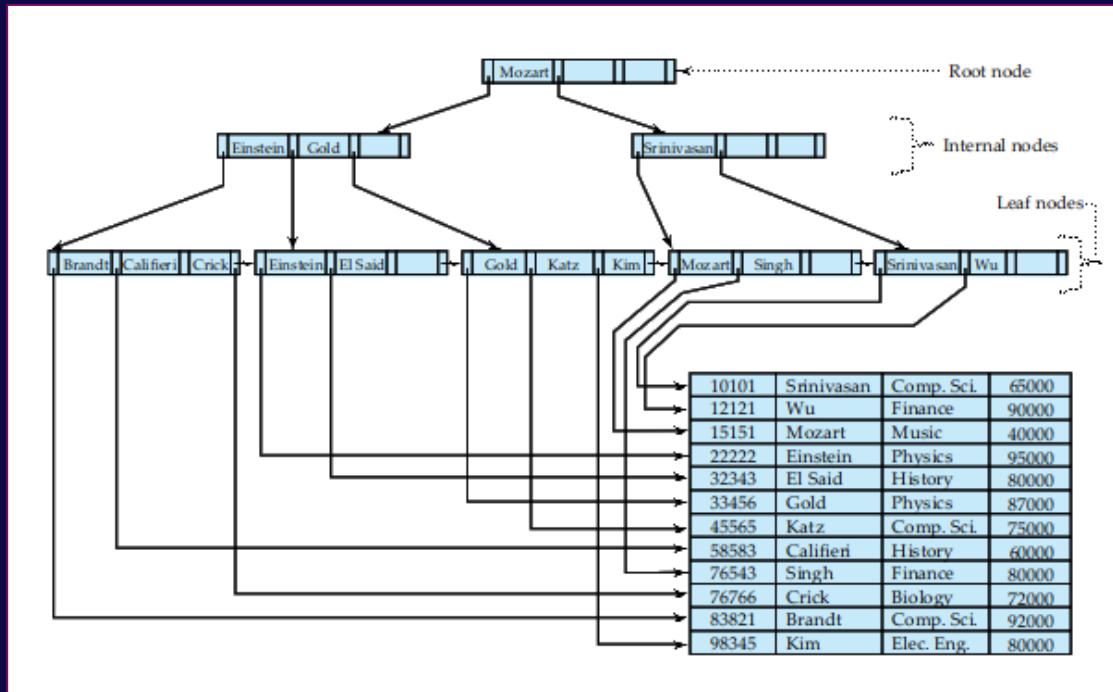


Figure 134 - B+-tree for instructor file ($n = 4$)

A **B+-tree** index is a multilevel index, unlike the multilevel index-sequential file, and the structure of its nodes differs as well. Each node can contain up to **n-1 search-key values** and **n pointers**, which are kept in sorted order. The ranges of values in each leaf do not overlap, except for duplicate **search-key values**, and **every search-key value must appear in some leaf node**.

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

Figure 135 - B+-tree for instructor file with $n = 6$

```

function find(value V)
/* Returns leaf node C and index i such that  $C.P_i$  points to first record
* with search key value V */
    Set C = root node
    while (C is not a leaf node) begin
        Let  $i$  = smallest number such that  $V \leq C.K_i$ 
        if there is no such number  $i$  then begin
            Let  $P_m$  = last non-null pointer in the node
            Set C =  $C.P_m$ 
        end
        else if ( $V = C.K_i$ )
            then Set C =  $C.P_{i+1}$ 
        else C =  $C.P_i$  /*  $V < C.K_i$  */
    end
    /* C is a leaf node */
    Let  $i$  be the least value such that  $K_i = V$ 
    if there is such a value  $i$ 
        then return (C, i)
    else return null ; /* No record with key value V exists */

procedure printAll(value V)
/* prints all records with search key value V */
    Set done = false;
    Set (L, i) = find(V);
    if ((L, i) is null) return
    repeat
        repeat
            Print record pointed to by  $L.P_i$ 
            Set  $i = i + 1$ 
        until ( $i >$  number of keys in L or  $L.K_i > V$ )
        if ( $i >$  number of keys in L)
            then L =  $L.P_n$ 
        else Set done = true;
    until (done or L is null)

```

Figure 136 - Querying a B+-tree

With a linear order of leaves based on their search-key values, the pointer **Pn** is used to chain the leaf nodes together in **search-key order**, allowing for efficient sequential processing of the file. Non leaf nodes of the **B+-tree** form a multilevel index on the leaf nodes, with a similar structure as the leaf nodes, but with all pointers pointing to tree nodes instead.

The **B+-tree index structure** provides a dynamic indexing solution that maintains its efficiency despite data insertion and deletion. Its multilevel structure allows for efficient sequential processing of the file, with acceptable performance and space overhead for frequently modified files. As we look toward the future of database indexing, the **B+-tree structure** remains a reliable and widely used option.

The **B+-tree structure** stands apart from its in-memory tree brethren. While binary trees consist of small nodes, each having no more than two pointers, the nodes in **B+ trees** are comparably large - often an entire disk block - and can contain a considerable number of pointers. As such, **B, trees** adopt a shape that is the inverse of the tall, skinny binary tree, instead opting for a short, wide structure.

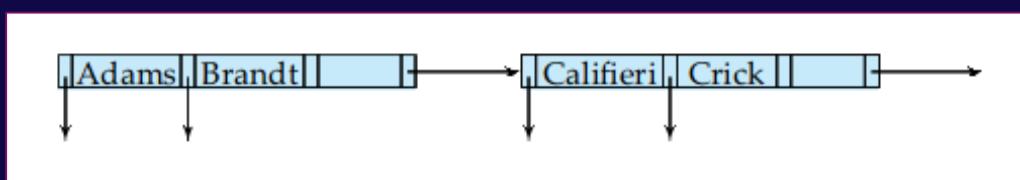


Figure 137 - Split of leaf node on insertion of "Adam"

However, despite their differences in size, both **B+ trees** and **binary trees** are useful for indexing records in a file, and the effectiveness of their lookup operations is contingent on the number of records contained therein. In a **balanced binary tree**, for instance, the length of the path for a lookup would be proportional to the logarithm of the number of records in the **indexed file**. For a file with one million records, this would necessitate around 20 node accesses.

If each node were located on a different disk block, **20 block** reads would be required to process a lookup. This is in stark contrast to the mere four block reads needed for the **B+-tree**, where nodes are more capacious and can hold more pointers. The difference in required block reads may seem trivial, but the time cost of each read, coupled with the need for disk arm seeks, compounds the disparity. Such a read with a disk arm seeks can take upwards of **10 milliseconds** on a typical disk, which is no small delay in the realm of computing.

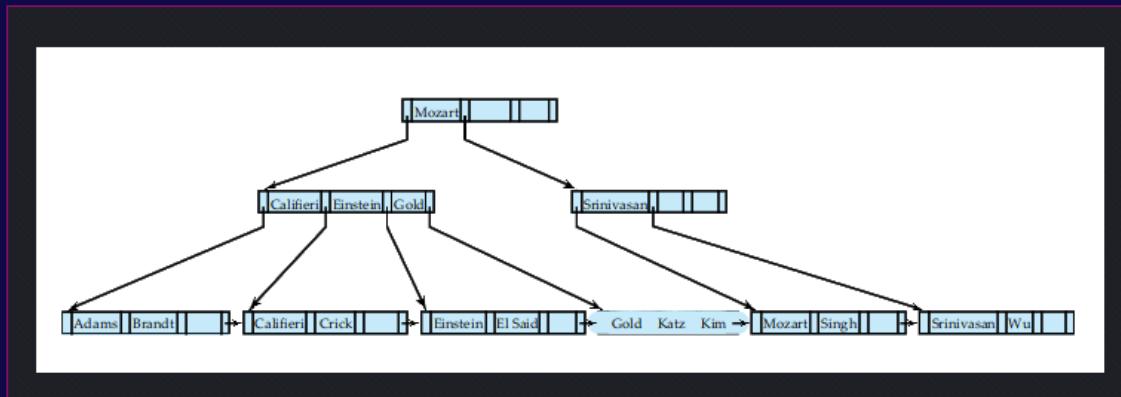


Figure 138 - Insertion of "Adams" into the B+-tree

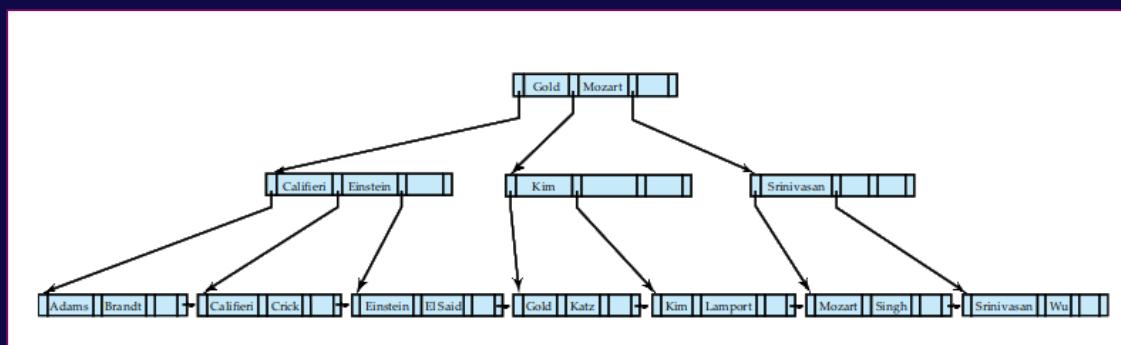


Figure 139 - Insertion of "Lamport" into the B+-tree

Updating records in a relation can be challenging, especially when indices are involved. A record update can be modeled as a deletion of the old record followed by an insertion of the updated record. Insertions and deletions can be more involved than lookups since they may necessitate **node splitting**, where a **node** is divided in **two**, or **node coalescing**, where two nodes are combined. In addition, when a node is split or combined, the balance of the tree must be maintained.

```

procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than n - 1 key values)
        then insert.in.leaf (L, K, P)
        else begin /* L has n - 1 key values already, split it */
            Create node L'
            Copy L.P1 . . . L.Kn-1 to a block of memory T that can
                hold n (pointer, key-value) pairs
            insert.in.leaf (T, K, P)
            Set L'.Pn = L.Pn; Set L.Pn = L'
            Erase L.P1 through L.Kn-1 from L
            Copy T.P1 through T.Klceil n/2 from T into L starting at L.P1
            Copy T.Plceil n/2+1 through T.Kn from T into L' starting at L'.P1
            Let K' be the smallest key-value in L'
            insert.in.parent(L, K', L')
        end

procedure insert.in.leaf (node L, value K, pointer P)
    if (K < L.K1)
        then insert P, K into L just before L.P1
        else begin
            Let Ki be the highest value in L that is less than K
            Insert P, K into L just after T.Ki
        end

procedure insert.in.parent(node N, value K', node N')
    if (N is the root of the tree)
        then begin
            Create a new node R containing N, K', N' /* N and N' are pointers */
            Make R the root of the tree
            return
        end
    Let P = parent (N)
    if (P has less than n pointers)
        then insert (K', N') in P just after N
        else begin /* Split P */
            Copy P to a block of memory T that can hold P and (K', N')
            Insert (K', N') into T just after N
            Erase all entries from P; Create node P'
            Copy T.P1 . . . T.Plceil n/2 into P
            Let K'' = T.Klceil n/2
            Copy T.Plceil n/2+1 . . . T.Pn+1 into P'
            insert.in.parent(P, K'', P')
        end
    end

```

Figure 140 - Insertion of entry in a B+-tree

When inserting a record into a **B+-tree**, one must first find the leaf node that should contain the new record. After doing so, an entry - consisting of the **search-key value and record pointer pair** - can be added to the node in such a way that the **search keys** remain sorted. Conversely, when deleting a record, the node containing the entry to be deleted must first be found, and the entry can then be removed. The remaining entries in the node to the right of the deleted entry are then shifted left by one position so that there are no gaps in the entries.

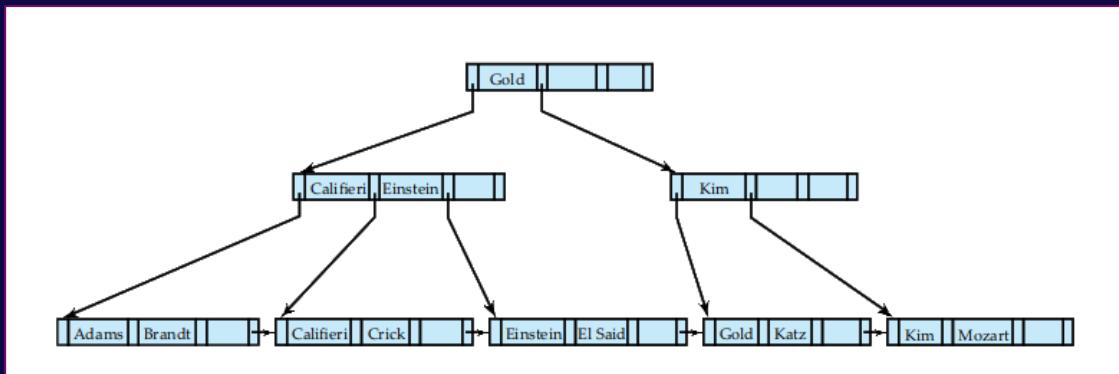


Figure 141 - Deletion of "Singh" and "Wu" from the B+-tree

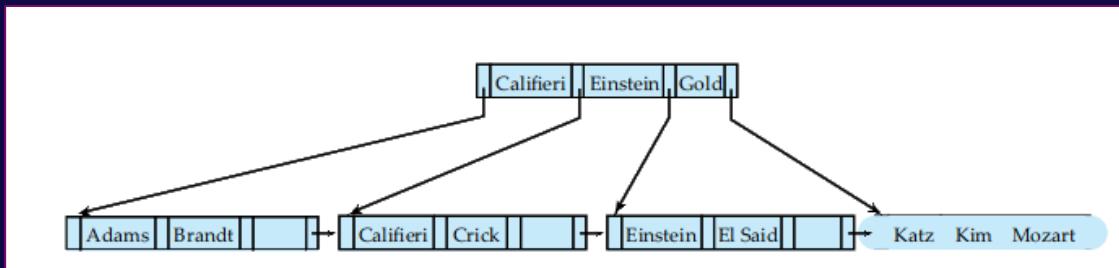


Figure 142 - Deletion of "Gold" from the B+-tree

Node splitting can occur when a node becomes too large to hold its contents, and the example given in the text illustrates this. If a record with the name "**Adams**" is inserted into a **B+-tree**, and the leaf node containing "**Brandt**," "**Califieri**," and "**Crick**" does not have enough space to accommodate it, the node must be split into two separate nodes. The **first $n/2$ search-key** values are kept in the existing node, while the remaining search-key values are added to the newly created node. The new leaf node is then inserted into the **B+-tree structure** by adding an entry - consisting of the smallest **search-key value** and a pointer to the new node - into the parent of the split node.

The updating of records in a **B+-tree structure** is complex, as it involves a delicate **balance of node splitting, node coalescing, and parent-node insertion**, all with an eye toward maintaining the overall balance of the tree.

The importance of efficient and speedy search operations cannot be overstated. Enter the **B+-tree**, a **ubiquitous index structure** used to optimize search operations on large datasets. However, as with any system, there are limitations to be addressed, such as the handling of nonunified **search keys**.

In situations where a relation can contain multiple records with the same search key value or a **nonunified search key**, the process of deleting a specific record can prove arduous. With a large number of entries to sift through, even spanning **multiple leaf nodes**, finding the corresponding entry for deletion can be a **time-consuming process**. To combat this, most database systems utilize a composite search **key** that contains the original search **key** and another attribute, unique to each record, **creating a unique search key** for each record. This extra attribute, known as the unifier attribute, enables the database to locate and delete the record efficiently with a single traversal from root to leaf.

While nonunified **search keys** pose challenges, an alternative to the traditional storage method exists, where each **key value** is stored only once in the tree, with a bucket or list of record pointers used to handle nonunified **search keys**.

This approach is more space-efficient, but implementation comes with **extra complexities**, such as **variable-sized buckets**, fetching records across **multiple blocks**, and **inefficient deletion operations**.

```

procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete.entry(L, K, P)

procedure delete.entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
        then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N') then swap.variables(N, N')
                if (N is not a leaf)
                    then append K' and all pointers and values in N to N'
                    else append all (Ki, Pi) pairs in N to N'; set N'.Pn = N.Pn
                    delete.entry(parent(N), K', N); delete node N
                end
            else begin /* Redistribution: borrow an entry from N' */
                if (N' is a predecessor of N) then begin
                    if (N is a nonleaf node) then begin
                        let m be such that N'.Pm is the last pointer in N'
                        remove (N'.Km-1, N'.Pm) from N'
                        insert (N'.Pm, K') as the first pointer and value in N,
                            by shifting other pointers and values right
                        replace K' in parent(N) by N'.Km-1
                    end
                else begin
                    let m be such that (N'.Pm, N'.Km) is the last pointer/value
                    pair in N'
                    remove (N'.Pm, N'.Km) from N'
                    insert (N'.Pm, N'.Km) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K' in parent(N) by N'.Km
                end
            end
            else ... symmetric to the then case ...
        end
    end

```

Figure 143 - Procedure for computing join

Despite the complexities, **B+-trees** remain a popular choice for index structures in database management systems, with their low **I/O** operation costs for insertion and deletion. The worst-case complexity of these operations is proportional to the height of the **B+-tree**, and with a fanout of **100**, operations result in fewer **I/O** operations than the worst-case bounds. In addition, with large memory sizes being commonplace, most **non-leaf nodes** are already in the database buffer when they are accessed, and only one or two **I/O** operations are required for a lookup.

Furthermore, with random insertions, **B+-tree** nodes can be expected to be more than two-thirds full on average, and even in the case of sorted insertions, they are only half full. The **B+-tree's** capabilities and limitations must be carefully considered when implementing a database management system, but their efficiency and speed make them a reliable and often utilized tool in **large-scale data management**.

B+-Tree Extensions

The use of the B+-tree index structure has become a staple for efficient and organized file storage. While index-sequential file organization can be riddled with performance issues as the file grows, the B+-tree structure provides a solution to the degradation of index lookups and actual record storage. By utilizing the leaf level of the B+-tree as an organizer for records in a file, rather than just as an index, we can avoid the issue of storing pointers to records and instead store the records themselves.

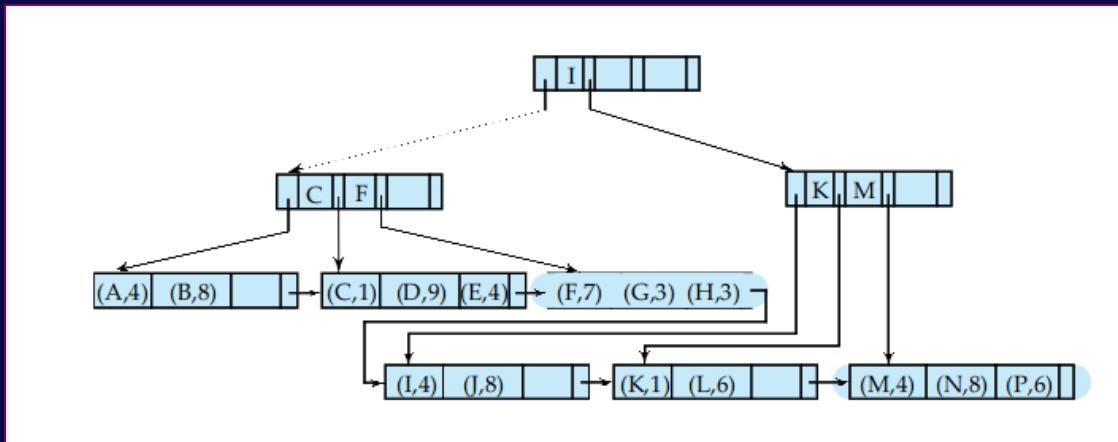


Figure 144 - B+-tree file organization

Inserting and deleting records in a **B+-tree file organization** follows the same principles as in a **B+-tree index**. To insert a record with a given key value, the system locates the block that should contain the record by searching the **B+-tree** for the largest key that is less than or equal to the given value. If the block has enough free space, the record is stored in the block.

Otherwise, the block is split and the records are redistributed to create space for the new record. Deleting a record works similarly, as the record is removed from its block and the non-leaf nodes of the **B+-tree** are updated.

- To improve space utilization in a **B+-tree**, we can involve more sibling nodes in redistribution during splits and merges. By redistributing entries between adjacent nodes during insertion or deletion, we can ensure that each node contains at least a certain number of entries. However, as more sibling nodes are involved in redistribution, the cost of updating becomes higher.
- It is important to note that in a **B+-tree index** or **file organization, leaf nodes** that are adjacent to each other in the tree may not be located adjacent to each other on disk. As such, sequential access to the file may become increasingly lost as insertions and deletions occur. In such cases, an index rebuild may be required to restore sequentiality.
- Finally, **B+-tree** file organizations can be used to store large objects, such as SQL clob and blobs, which may be larger than a disk block. By splitting these objects into smaller records that are organized in a B+-tree file organization, we can efficiently store and access these large objects.

As we delve into the world of indexing, we come across the B-Tree and the B+-Tree - two distinct data structures that serve a common purpose. The B-Tree stands out from its sibling in that it avoids the redundant storage of search-key values, a trait that could potentially reduce the number of nodes in an index.

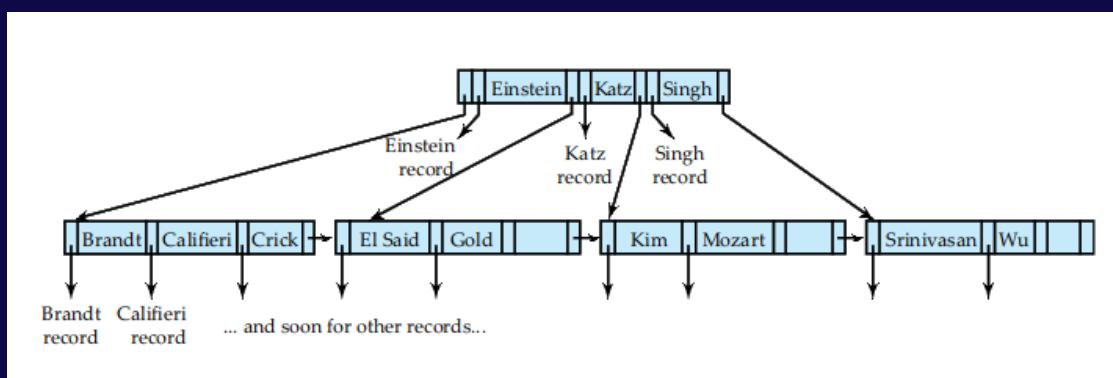


Figure 145 - Procedure for computing join

As we scrutinize the Figure above, we notice that the **B-Tree**, unlike the **B+-Tree**, limits the number of times a **search-key value appears**. This constraint brings about the need for additional pointer fields in nonleaf nodes that point to file records or buckets associated with a search key.

Despite the growing trend in using **B-Tree** and **B+-Tree** interchangeably, this book adheres to the conventional definition of the two data structures to prevent any confusion. A generalized **B-Tree** is presented in Figure below, with leaf nodes identical to that of the B+-Tree.

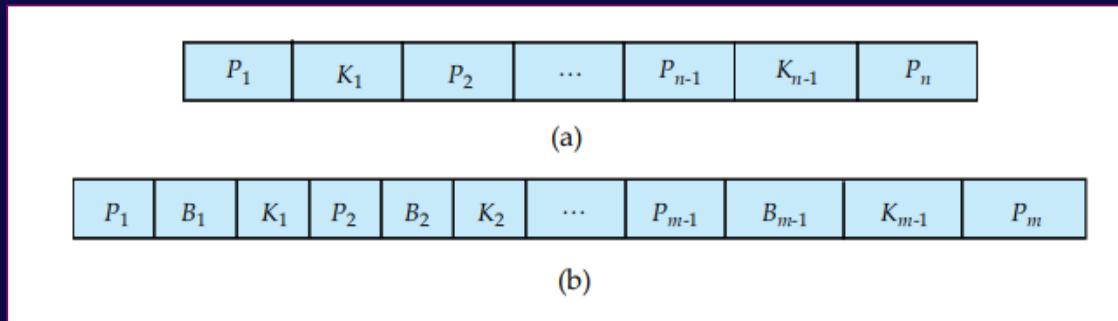


Figure 146 - Typical nodes of a B-tree. (a) Leaf node. (b) Nonleaf node

The number of nodes accessed during a lookup in a **B-Tree** varies based on the location of the search key. In contrast to the **B+-Tree**, a **B-Tree** could present the desired value before reaching a leaf node. Nevertheless, the advantages of finding certain values early on are insignificant, given that **B-Tree's nonleaf nodes** contain fewer **search keys**, leading to a smaller fanout and a greater depth than that of its cousin.

Deleting entries in a **B-Tree** is more complicated than in a **B+-Tree**, given that the entry could appear in nonleaf nodes. On the other hand, inserting values in a **B-Tree** is a relatively simple process. However, when it comes to large indices, the space advantages of **B-Trees** are trivial, making **B+-Tree** the preferred choice for most **database-system implementations**.

As we tread further into the world of indexing, it's worth noting that the rise of flash memory has revolutionized data storage, and its impact on the index structure is undeniable. With the **B+-Tree** index structure compatible with flash memory storage, faster access speeds are guaranteed, and lookup times are reduced to a mere microsecond, giving magnetic disk storage a run for its money.

Multiple-Key Access

The use of multiple indices is a strategy that can greatly improve query processing efficiency for certain types of queries. In particular, when queries involve multiple attributes, an index built on a composite search key can provide a significant advantage over the use of **single-key indices**.

However, the effectiveness of this strategy can depend heavily on the **specific query** and **data characteristics**. In some cases, the use of multiple indices can result in a large number of pointers being scanned to produce a **small result**, leading to poor query performance. To address this issue, specialized index structures such as bitmap indices and **R-trees** have been developed to speed up the intersection and comparison operations involved in query processing.

Another optimization technique is the use of covering indices, which store the values of some attributes along with the pointers to the record. This technique can allow certain queries to be answered using just the **index**, without accessing the actual records. While this technique can improve query processing efficiency, it also increases the size of the search key, leading to increased storage requirements.

Overall, the use of multiple indices and specialized index structures **represents a powerful tool** in the arsenal of database system designers, but it must be applied judiciously and with careful consideration of the specific query and data characteristics involved.

Static Hashing

Hashing has emerged as a technique that provides a way of constructing indices and avoids accessing index structures or employing binary search, thereby reducing the number of **input/output (I/O)** operations. However, not all hash functions are created equal. To ensure an even distribution of records across buckets, an ideal hash function should distribute search keys uniformly across all buckets and **appear random**, regardless of any externally visible ordering on the **search-key values**.

bucket 0				

bucket 1				
15151	Mozart	Music	40000	

bucket 2				
32343	El Said	History	80000	
58583	Califieri	History	60000	

bucket 3				
22222	Einstein	Physics	95000	
33456	Gold	Physics	87000	
98345	Kim	Elec. Eng.	80000	

bucket 4				
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	

bucket 5				
76766	Crick	Biology	72000	

bucket 6				
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	

bucket 7				

Figure 147 - Hash organization of instructor file, with dept name as the key

To illustrate this point, consider the case of the instructor file, where the search key is the department name. If a simple hash function is employed, which maps names beginning with the i th letter of the alphabet to the i th bucket, the distribution will not be uniform due to the imbalance in the number of names beginning with certain letters.

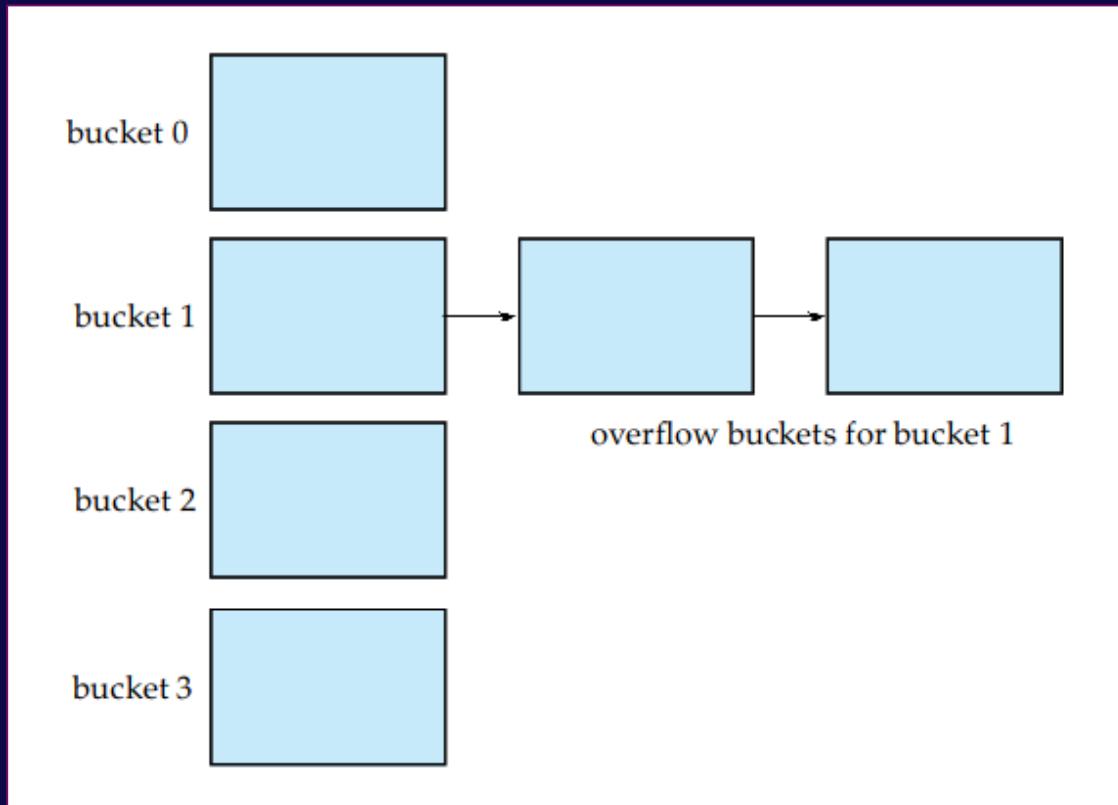


Figure 148 - Overflow chaining in a hash structure

On the other hand, a hash function based on the search key salary, which divides the values into **10** ranges, provides a uniform distribution but is not random, as records with certain salaries are more common than others.

Therefore, it is important to choose a hash function carefully, taking into account the specific characteristics of the search keys and the size of the file. Typical hash functions operate on the binary representation of characters in the **search key** and **employ mathematical operations** to ensure an even distribution of records across buckets. By selecting an appropriate hash function, file organization can be optimized, reducing the number of **I/O** operations required and improving overall system performance.

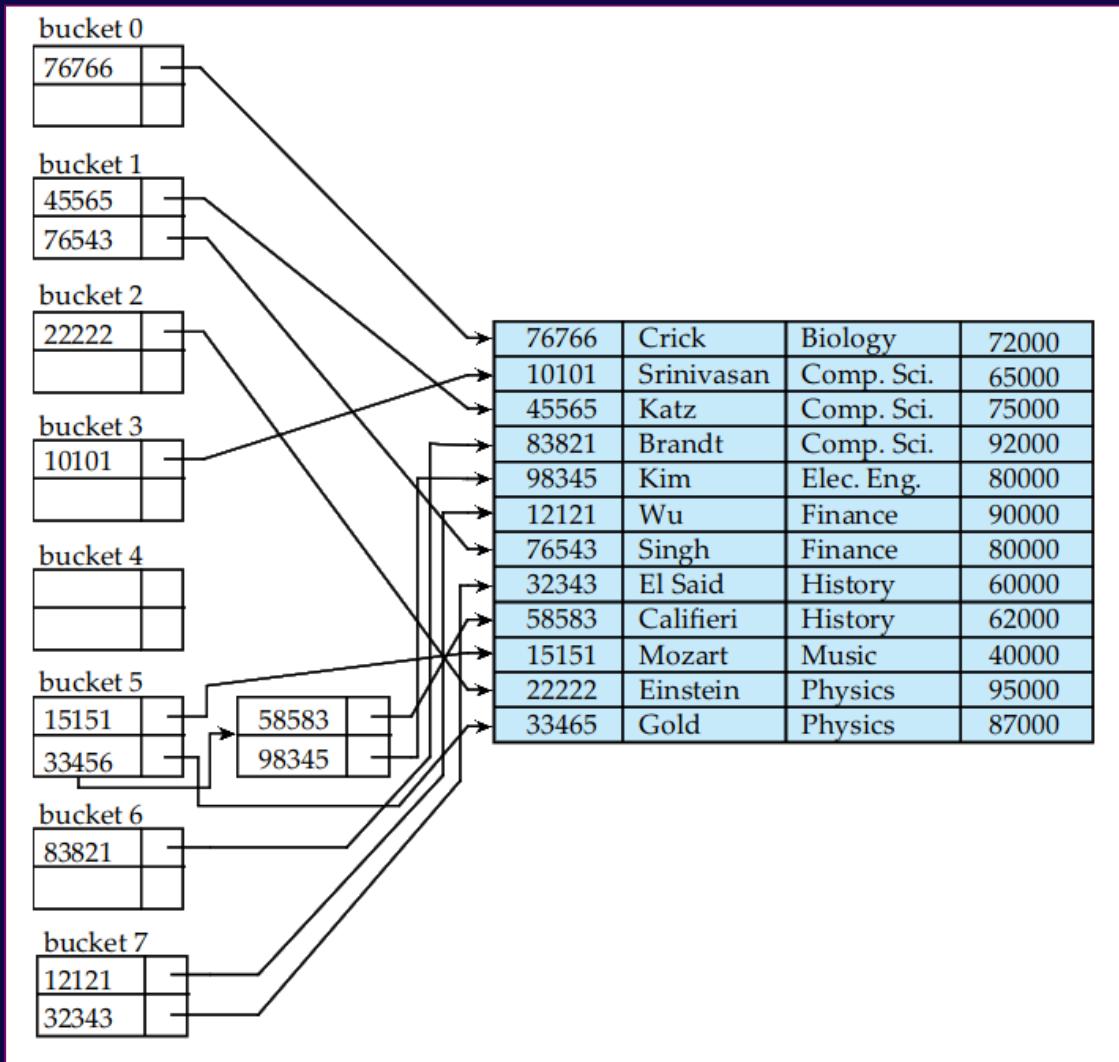


Figure 148 - Hash index on search key ID of instructor file

Dynamic Hashing

The issue of fixed bucket addresses presents a grave challenge when utilizing the static hashing technique. Databases, as we know, tend to grow in size over time. This, however, poses a problem for static hashing, and when dealing with such databases, we have three possible options.

- Firstly, we may choose to base our hash function on the current size of the file. However, this option results in a loss of performance as the database continues to expand. Secondly, we can opt for a hash function based on the anticipated size of the file at some point in the future. Although this approach avoids a decline in performance, it may result in a substantial initial wastage of space.
- Lastly, we may periodically reorganize the hash structure to accommodate the growth of the database. This, however, is a **massive** and **time-consuming operation**, requiring the choice of a new hash function, the recomputation of the hash function on every record in the file, and the generation of new bucket assignments. Moreover, it is necessary to prevent access to the file during this reorganization process.
- Fortunately, several dynamic hashing techniques, including extendable hashing, allow for the hash function to be modified dynamically to cope with the growth or shrinkage of the database. With extendable hashing, we can split and coalesce buckets as the database expands or contracts, thus retaining space efficiency. The reorganization process in this case is performed on only one bucket at a time, ensuring that the resulting performance overhead is acceptably low.

To achieve this, we select a hash function h that generates values over a relatively large range, typically b -bit binary integers, with b being **32** in most cases. Instead of creating a bucket for each hash value, we create buckets on demand, as records are inserted into the file. At any point, we use i bits, where $0 \leq i \leq b$, to locate the correct bucket for a record.

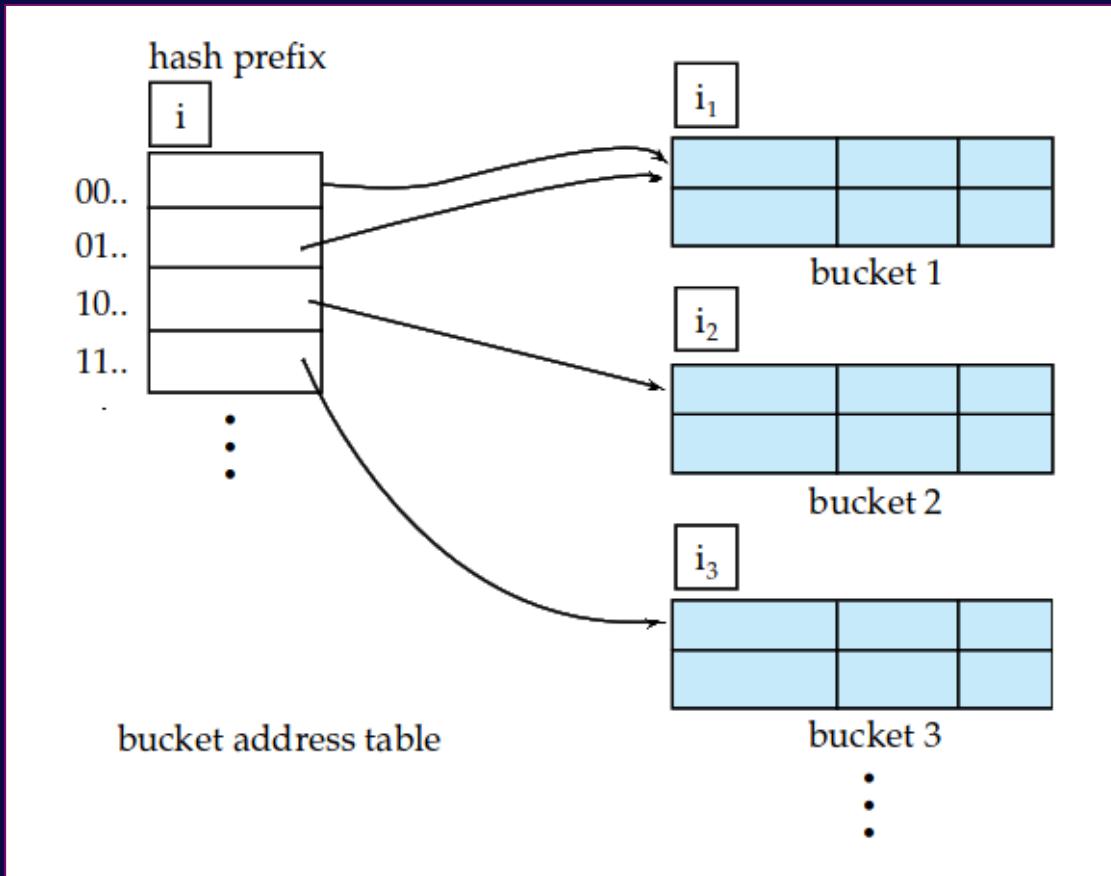


Figure 149 - General extendable hash structure

The general extendable hash structure, as illustrated in the figure above, shows the **i** bits required to determine the correct bucket for a given record. The integer associated with each bucket represents the length of the common hash prefix shared by all entries that point to that bucket.

<i>dept_name</i>	<i>h(dept_name)</i>
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Figure 150 - Hash function for dept name

To perform a **lookup**, **insertion**, or **deletion** on an extendable hash structure, the system takes the first **i** high-order bits of the hash value and follows the corresponding table entry for this bit string, thereby locating the bucket containing the **search-key** value. If the bucket is full, it must be split and the records redistributed. To determine whether it needs to increase the number of bits used, the system checks whether **i** is equal to **ij**.

If **i** is equal to **ij**, only one entry in the bucket address table points to the bucket. In this case, the system increases the size of the bucket address table by considering an additional bit of the hash value, doubling the size of the table, and allocating a new bucket. It then rehashes each record in the original bucket and, depending on the first **i** bits, either keep it in the original bucket or assigns it to the newly created bucket.

Extendable hashing offers an efficient and dynamic solution for databases that grow or shrink in size. In the pursuit of efficient data storage and retrieval, computer scientists have long been on the lookout for the most optimal hashing techniques.

Among the contenders for the top spot is extendable hashing, a dynamic hashing technique that promises optimal performance without the typical space overhead associated with other hashing techniques.

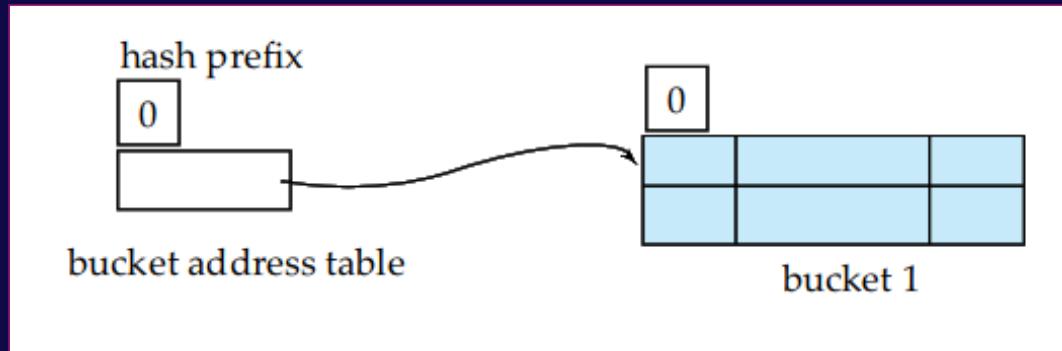


Figure 151 - Initial extendable hash structure

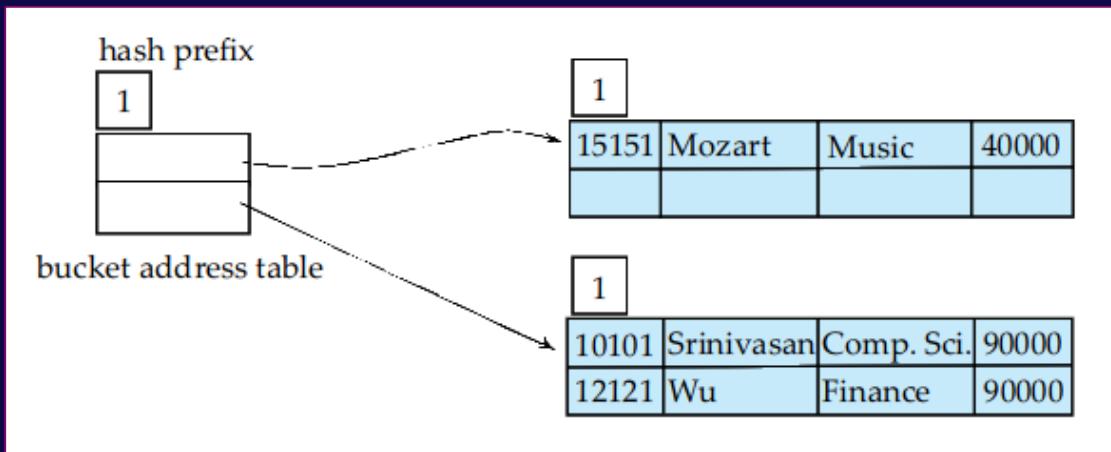


Figure 152 - Hash structure after three insertions

In a recent demonstration of the power of extendable hashing, the illustrious instructor file in the figure was used as a test case to illustrate the operation of insertion.

Utilizing the **32-bit hash values**, and assuming an empty file as in figure, records were inserted one by one, to showcase all the features of extendable hashing in a small structure.

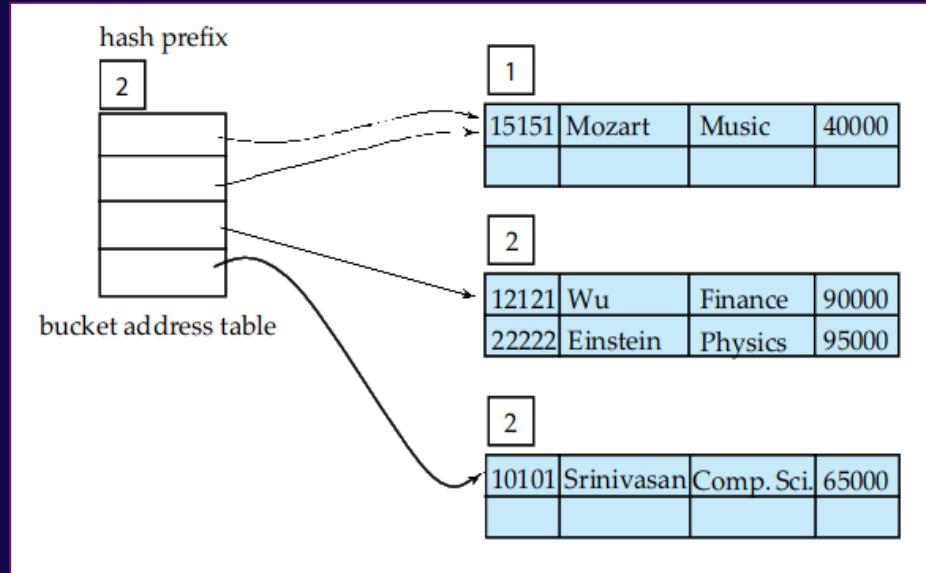


Figure 153 - Hash structure after four insertions

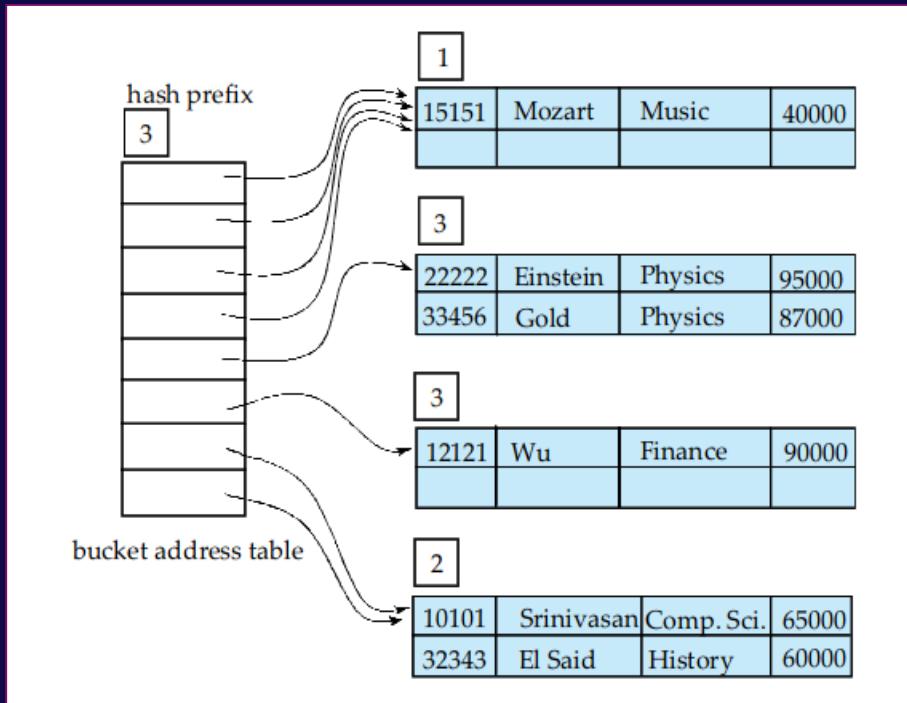


Figure 154 - Hash structure after six insertions

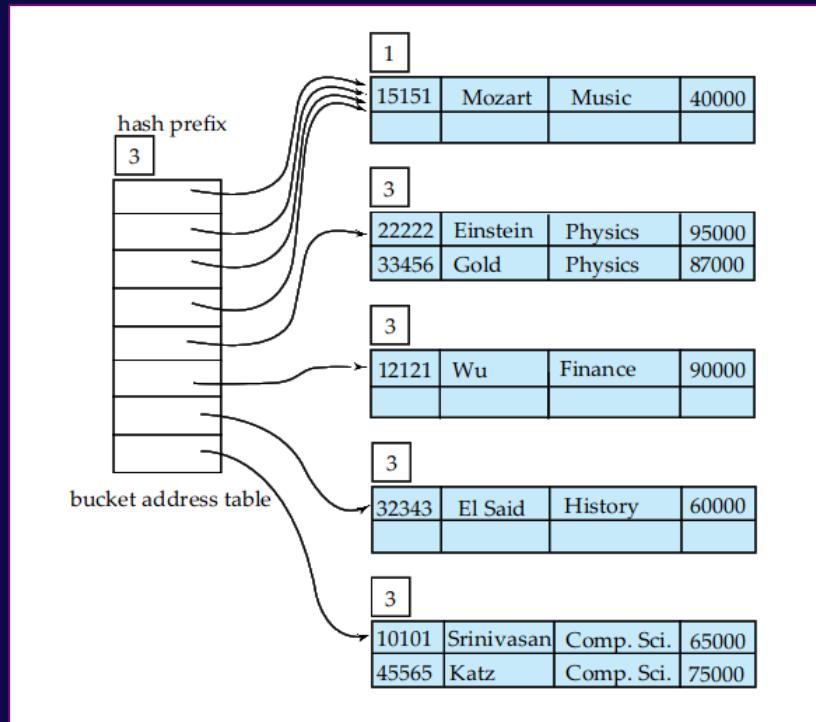


Figure 155 - Hash structure after seven insertions

In the interest of simplicity, it was assumed that each bucket could hold only two records. The first two records were inserted without incident, but when the third record was introduced, it became evident that the bucket was already full.

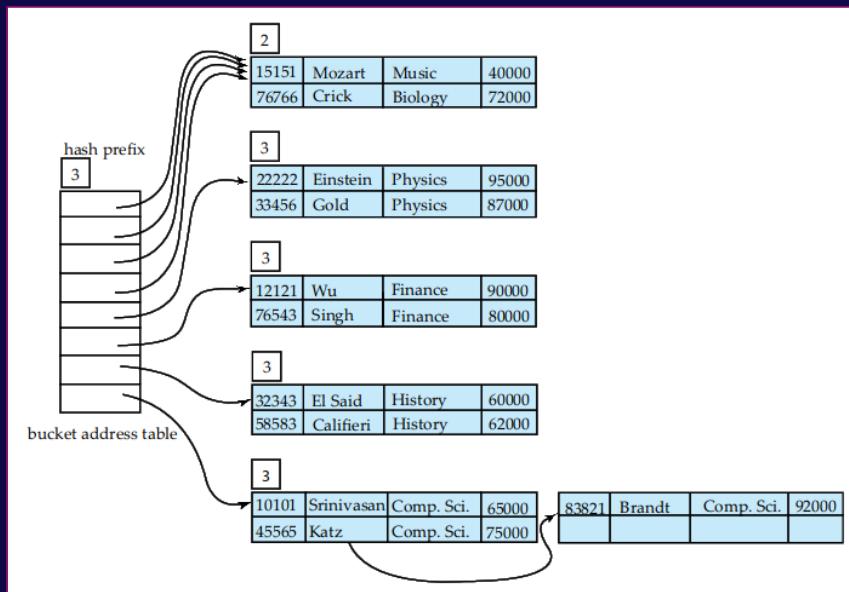


Figure 156 - Hash structure after eleven insertions

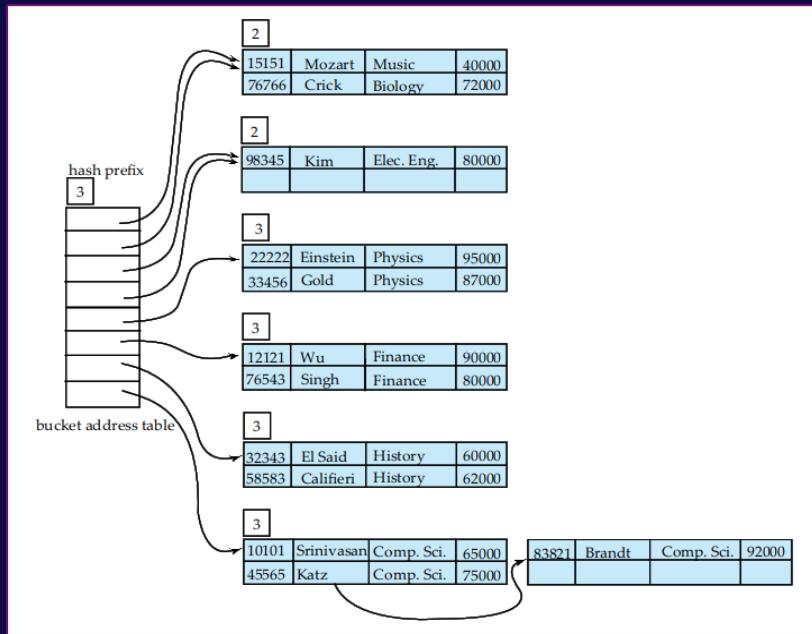


Figure 157 - Extendable hash structure for the instructor file

To accommodate the overflow, the number of bits used in the hash value was increased, and the bucket was split, with the new bucket holding records whose search key had a hash value beginning with **1**. As the file grew, buckets continued to be allocated dynamically, without the need for any buckets to be reserved for future growth.

While extendable hashing does involve an additional level of indirection in the lookup process, the advantages of its dynamic allocation and optimal performance make it a highly attractive technique for data storage and retrieval.

Though it requires a more complex implementation than other hashing techniques, the bibliographical notes provide more detailed descriptions of extendable hashing and **alternative dynamic hashing techniques**, such as linear hashing, that avoid the extra level of indirection.

Comparison of Ordered Indexing and Hashing

The choice of file organization and indexing technique can make all the difference. Should one use ordered indexing or hashing? The answer is not so straightforward, as each technique has its advantages and disadvantages depending on the situation at hand.

If a system's queries primarily involve searching for a specific value, a hashing scheme may be preferable. With hashing, the average lookup time is constant and independent of the size of the database. In contrast, an ordered-index lookup requires time proportional to the logarithm of the number of values in r for the given attribute. However, **the worst-case lookup time** for hashing is proportional to the number of values in r for the attribute, whereas the **worst-case lookup** time for ordered indexing is proportional to the logarithm of the number of values in r . Nevertheless, as worst-case scenarios are unlikely to occur with hashing, it remains the preferred method.

On the other hand, if the queries involve a range of values, **ordered-index techniques** are preferable. When executing a range query with an ordered index, the system performs a lookup on the first value in the range, then follows the pointer chain in the index to read the next bucket in order until it reaches the end of the range. With hashing, it is difficult to determine the next bucket that must be examined, as a good hash function assigns values randomly to buckets. Thus, the values in the specified range are likely to be scattered across many or all of the buckets, necessitating the reading of all the buckets to find the required search keys.

In general, the decision to use ordered indexing or hashing depends on several factors. The cost of periodic reorganization of the index or hash organization, the relative frequency of insertion and deletion, the desired optimization of average access time versus increasing **worst-case access time**, and the types of queries that users are likely to pose all come into play. Ultimately, the choice between these two techniques must be made based on the expected type of query. However, in cases where it is not known in advance whether range queries will be infrequent or not, it is usually best to choose ordered indexing, unless the database is temporary and no range queries will be performed.

Bitmap Indices

After delving into the depths of database indexing, we come across the specialized world of Bitmap Indices. These unique indices offer a streamlined approach to querying multiple keys within a given relation. While each bitmap index is built on a **single key**, the sequential numbering of records within the relation paves the way for efficient retrieval of information.

To illustrate the usefulness of bitmap indices, let us consider a relation with an attribute that can only take on a small number of values. A prime example of this would be the attribute "*gender*" in a relation containing instructor information. In such a scenario, creating a bitmap index for each possible value of the attribute allows for quick identification of the relevant records.

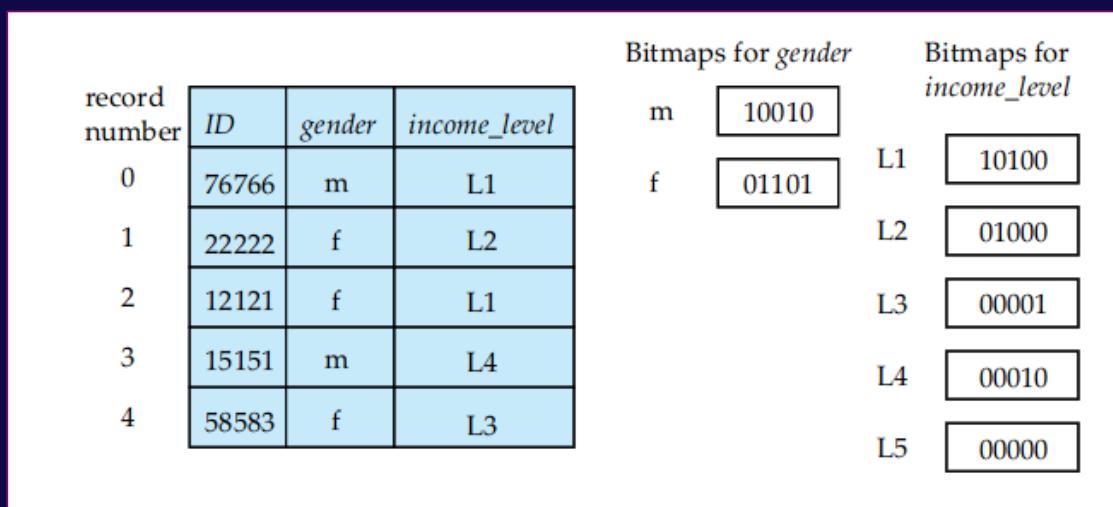


Figure 158 - Bitmap indices on relation *instructor info*

But the true power of bitmap indices shines through when there are selections on multiple keys. By creating bitmap indices on multiple attributes, we can perform queries that require satisfying multiple conditions. The intersection of the resulting bitmaps provides the necessary information for the query, enabling us to bypass access to the entire relation.

In addition to their query prowess, bitmap indices also offer a space-saving advantage. With each bitmap consisting of a mere array of bits, the size occupied by bitmap indices is typically less than **1%** of the relation size. And when an attribute can only take on a limited number of values, the bitmap index size shrinks even further, occupying a measly **1%** of the relation size.

The flexibility and efficiency of bitmap indices make them a **valuable tool** for **data analysis**. By leveraging their unique structure, we can retrieve information and count the number of tuples satisfying a given selection, all without even accessing the relation. So the next time you find yourself knee-deep in database indexing, consider the power of bitmap indices to streamline your queries and simplify your data analysis.

Index Definition in SQL

The SQL standard offers no provisions for the user or administrator to regulate the creation and maintenance of indices in the database system. While indices are not essential for data accuracy, they play a pivotal role in efficient transaction processing, including updates and queries. Moreover, indices facilitate the enforcement of integrity constraints.

In theory, a database system could automate the process of determining which indices to create. However, due to the spatial costs associated with indices and their impact on update processing, it is challenging to automatically make the right choices. As a result, most SQL implementations provide programmers with control over index creation and deletion via **data definition language (DDL)** commands.

The syntax of these commands is **illustrated next**. While widely used and supported by various database systems, it is not part of the SQL standard. The SQL standard restricts itself to the logical database schema and does not support the physical database schema's management.

Creating an index requires the "*create index*" command with the following form: "*create index on ()*". The attribute list consists of the attributes of the relations that constitute the search key for the index.

To create an index named "*dept index*" on the *instructor* relation with "*dept name*" as the search key, we would write "*create index dept index on the instructor (dept name)*". If we wish to declare that the search key is a candidate key, we add the attribute "*unique*" to the index definition. The command "*create unique index dept index on the instructor (dept name)*" would indicate that "*dept name*" is a candidate key for "*instructor*." However, it is unlikely that this is what we would want for our university database.

If "*dept name*" is not **a candidate key** at the time of the "*create unique index*" command, the system will display an error message, and the index creation attempt will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. It is worth noting that the "*unique*" feature is redundant if the database system supports the unique declaration of the SQL standard.

Some database systems allow the specification of the type of index to be used, such as **B+-tree** or **hashing**. Additionally, certain database systems permit one of the indices on a relation to be declared as "*clustered*." The system then stores the relation sorted by the **search-key** of the clustered index.

Finally, to drop an index, the "*drop index*" command takes the form "*drop index*." The index name we specified when creating the index is required for this command.

4.3 Query Processing

The intricate process of extracting data from a database, known as query processing, involves a multitude of activities. Among these activities are the translation of **high-level database language queries** into expressions that can be utilized at the physical level of the file system, a **myriad of query-optimizing transformations**, and ultimately, the evaluation of queries. The complex and dynamic nature of query processing necessitates a careful and deliberate approach, requiring a thorough understanding of the underlying database architecture and the nuances of query optimization.

Overview

The process of query processing is a complex and intricate task, involving several steps that culminate in the retrieval of data from a database. These steps are illustrated in the figure and include parsing and translation, optimization, and evaluation. Before the commencement of query processing, the system must first convert the query into a format that is suitable for internal use. While SQL is well-suited for human consumption, it is unsuitable as an internal representation of a query. Therefore, a more useful format is one based on extended relational algebra.

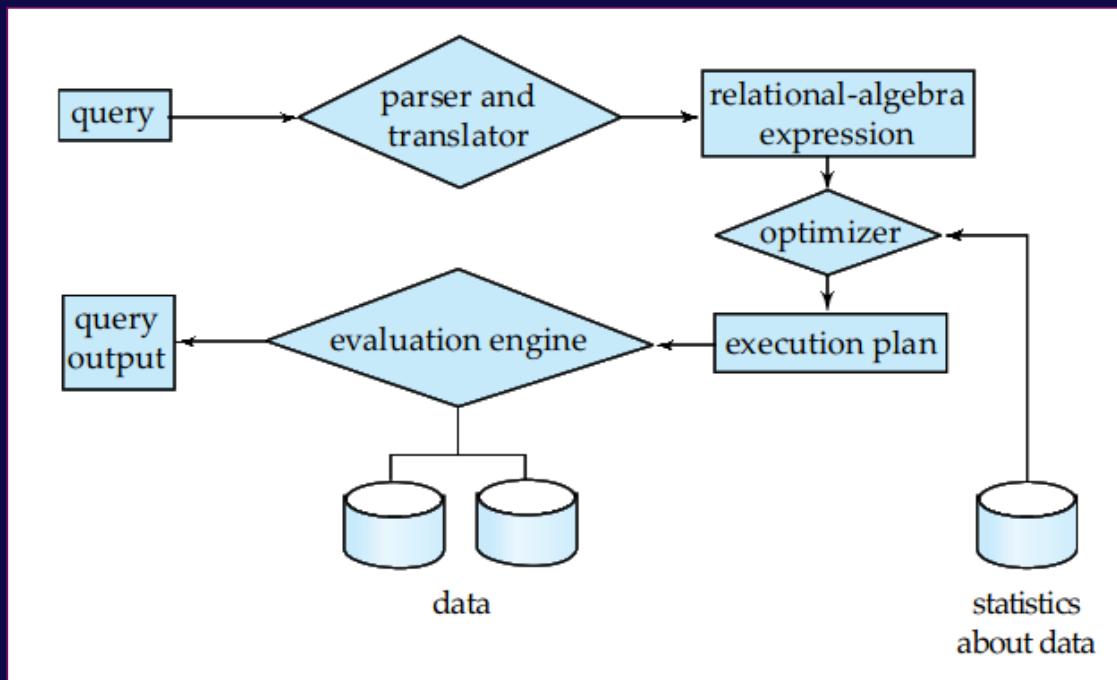


Figure 159 - Steps in query processing

The initial step in query processing is the translation of the query into its internal format, a process akin to that performed by a compiler's parser. During the translation process, the parser checks the syntax of the query, confirms that the relation names referenced in the query are indeed present in the database, and so on. The result of this process is a parse-tree representation of the query, which is subsequently converted into a relational-algebra expression. If the query is defined in terms of a view, the translation phase replaces all references to the view with the relational-algebra expression that defines the view.

- $\sigma_{\text{salary} < 75000} (\Pi_{\text{salary}} (\text{instructor}))$
- $\Pi_{\text{salary}} (\sigma_{\text{salary} < 75000} (\text{instructor}))$

Figure 160 - Example

While a query can be expressed in various ways, there are typically multiple methods for computing the answer. The relational-algebra representation of a query partially specifies how the query is evaluated, but several different algorithms can be used to evaluate the expression. The process of fully specifying how to evaluate a query involves not only providing the relational-algebra expression but also annotating it with instructions on how to evaluate each operation, a process known as query optimization. A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or query-evaluation plan.

Different evaluation plans for the same query can have different costs, and it is the responsibility of the system to construct a **query-evaluation plan** that minimizes the cost of query evaluation. This task, known as query optimization, is discussed in detail in the chapter. Once the query plan is chosen, the query is evaluated, and the result of the query is output.

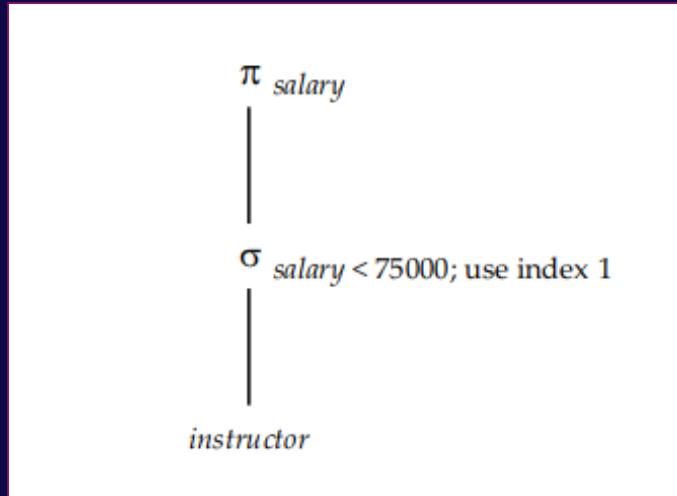


Figure 161 - A query-evaluation plan

Although the precise sequence of steps involved in query processing can vary among databases, the concepts described here form the foundation of query processing in databases. To optimize a query, the query optimizer must have knowledge of the cost of each operation. While it is difficult to compute the exact cost, it is possible to estimate the execution cost for each operation.

In this chapter, we delve into how individual operations are evaluated in a query plan and how their costs can be estimated. Additionally, we explore how pipelined operations can be used to coordinate the execution of multiple operations and avoid the writing of intermediate results to disk.

Measures of Query Cost

Optimizing query evaluation plans in a database system is a crucial task that involves comparing and choosing among several alternatives based on their estimated costs. To estimate the cost of an evaluation plan, it is necessary to first estimate the cost of individual operations and then combine them to get the total cost of the plan.

The cost of query evaluation is measured in terms of various resources such as **disk accesses, CPU time, and communication costs** in a distributed system. However, in large databases, the cost of accessing data from disk is usually the most significant as disk accesses are inherently slower than **in-memory operations**. Therefore, disk activity is expected to dominate the total query execution time, and the number of block transfers and disk seeks are used to estimate the cost of a query evaluation plan.

To further refine the cost estimates, it is necessary to distinguish between block reads and writes as the latter are typically twice as expensive as reads. Nevertheless, such details are often ignored for simplicity, and buffer size in main memory is used to estimate the cost of various algorithms. In the best case, all data can be read into buffers, and the disk is not accessed again. However, in the worst case, only a few blocks of data can be stored in the buffer, and the cost estimates are based on such pessimistic assumptions.

In addition, the response time for a query evaluation plan is a critical measure of its cost. Unfortunately, estimating response time is challenging as it depends on multiple factors such as the contents of the buffer when the query begins execution and the distribution of accesses among multiple disks. As a result, rather than minimizing response time, optimizers focus on minimizing the total resource consumption of the query plan.

Estimating the cost of query evaluation plans involves considering various factors such as **disk accesses, CPU time, and communication costs**. Although the response time is an essential measure of cost, estimating it is challenging, and optimizers instead focus on minimizing total resource consumption.

Selection Operation

The file scan reigns supreme as the most fundamental operator for accessing data. It functions as a search algorithm that relentlessly seeks out and retrieves records that meet the conditions specified in a selection. In the case of relational systems, the file scan allows for the entire relation to be read when it is stored in a single, dedicated file.

When it comes to implementing a selection operation on a relation consisting of tuples stored together in one file, there are various algorithms to consider. The linear search algorithm, for instance, entails scanning each file block and testing all records for compatibility with the selection condition. Although it may not be the speediest method of implementing a selection, the linear search algorithm can be applied to any file, regardless of the ordering of the file, the availability of indices, or the nature of the selection operation.

On the other hand, other selection algorithms that exist tend to be faster than linear search but are not universally applicable. Among these algorithms, some use index structures, which are referred to as access paths. Such structures provide a pathway through which data can be located and accessed, with the selection predicate guiding the choice of the index used in processing the query.

For instance, a primary index (*also known as a clustering index*) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. When an equality comparison on a key attribute with a primary index is made, the index can be used to retrieve a single record that satisfies the corresponding equality condition.

Cost estimates are shown in the Figure below:

Algorithm	Cost	Reason
A1 Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1 Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies condition, scan can be terminated as soon as the required record is found. In the worst case, b_r blocks transfers are still required.
A2 Primary B ⁺ -tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3 Primary B ⁺ -tree Index, Equality on Nonkey	$h_i * (t_T + t_S) + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a primary index) and don't require additional seeks.
A4 Secondary B ⁺ -tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to primary index.
A4 Secondary B ⁺ -tree Index, Equality on Nonkey	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5 Primary B ⁺ -tree Index, Comparison	$h_i * (t_T + t_S) + b * t_T$	Identical to the case of A3, equality on nonkey.
A6 Secondary B ⁺ -tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on nonkey.

Figure 162 - Cost estimates for selection algorithms

Additionally, there are cases where multiple records may need to be fetched, such as when a **primary index** is used to retrieve multiple records if the selection condition specifies an equality comparison on a **non-key attribute**. In such instances, the records must be stored consecutively in the file since the file is sorted on the **search key**.

Selections specifying an equality condition can also use a **secondary index**. If the equality condition is on a key, this strategy can retrieve a single record. However, if the indexing field is not a key, multiple records may be retrieved. In this second case, each record may be resident on a different block, which could result in one **I/O** operation per retrieved record, with each **I/O** operation requiring a seek and a block transfer. The worst-case time cost, in this case, is dependent on several factors, and could even exceed that of linear search if a large number of records are retrieved.

It is worth noting that when records are stored in a **B+-tree** file organization or other file organizations that may require relocation of records, secondary indices usually do not store pointers to the records. Instead, they store the values of the attributes used as the search key in a **B+-tree** file organization.

Accessing a record through such a secondary index is more expensive, as it requires searching the secondary index to find the primary index **search-key values**, and then looking up the records in the **primary index**. Cost formulae described for secondary indices must be modified accordingly in such cases.

Selection operations are a crucial aspect of query processing, and there are various algorithms and index structures available to streamline and optimize the selection process. From the straightforward linear search to the **more complex primary** and **secondary index searches**, each algorithm has its strengths and limitations, and their respective costs must be carefully considered in determining the most efficient approach.

Sorting

Sorting data in database systems is a crucial task with multiple benefits. First, it enables SQL queries to specify the output to be sorted. Second, sorting helps enhance the efficiency of relational operations such as joins when the input relations are sorted. Therefore, sorting is a fundamental step that precedes the discussion of join operations.

```
i = 0;  
repeat  
    read M blocks of the relation, or the rest of the relation,  
        whichever is smaller;  
    sort the in-memory part of the relation;  
    write the sorted data to run file Ri;  
    i = i + 1;  
until the end of the relation
```

Figure 163 - Example

While building an index on the sort key can sort a relation, it orders the relation logically through an index rather than physically. This process can lead to disk access for each record, making it very expensive. Hence, it may be preferable to sort records physically.

Sorting has been studied extensively for relations that are bigger than memory. The most commonly used technique for external sorting is the external sort-merge algorithm. This technique involves creating sorted runs with each run sorted but containing only some of the records of the relation. Then, the runs are merged, and the output is the sorted relation. The merge stage operates by reading one block of each file into a buffer block in memory and then choosing the first tuple (in sort order) among all buffer blocks.

The output of the merge stage is a generalization of the two-way merge used by the standard **in-memory sort-merge** algorithm, but it merges **N** runs, so it is called an **N-way** merge.

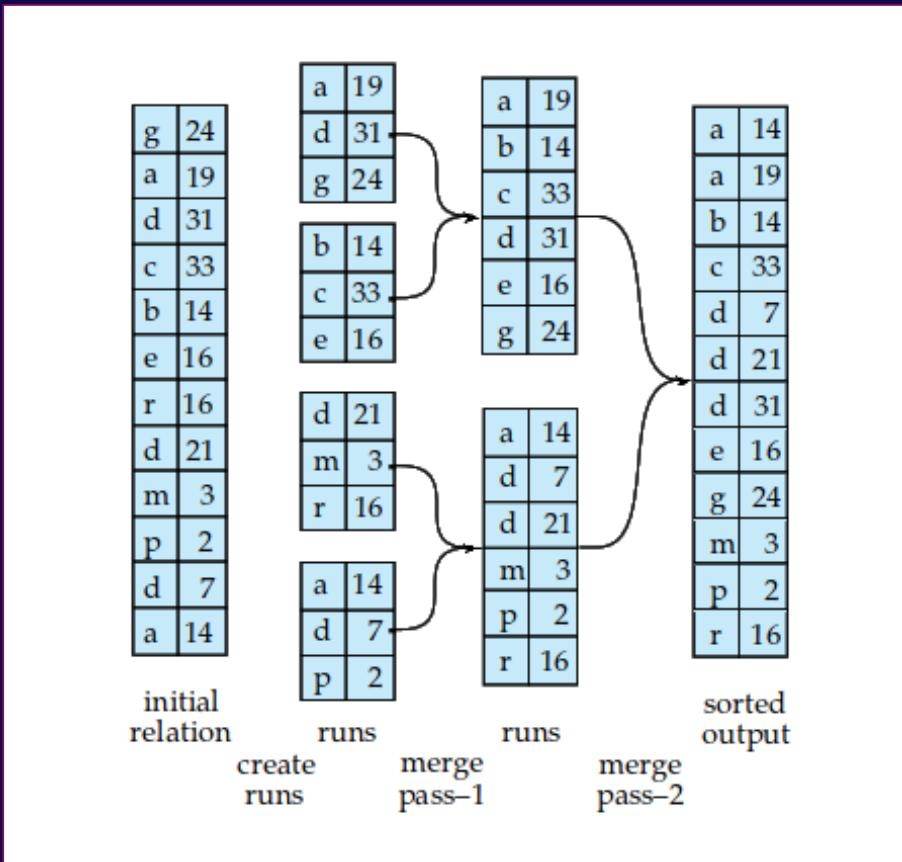


Figure 164 - External sorting using sort-merge

If the relation is much larger than memory, there may be **M** or more runs generated in the first stage, and it is not possible to allocate a block for each run during the merge stage. In this case, the merge operation proceeds in multiple passes. Each pass reduces the number of runs by a factor of **M** – 1, and the passes repeat as many times as required until the number of runs is less than **M**. Finally, a final pass generates the sorted output.

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

Figure 165 - Formula

The cost analysis of external sort-merge is calculated using the number of blocks containing records of relation r , with the total number of block transfers for external sorting of the relation being given by a specific formula. By understanding the sorting of data and the **external sort-merge algorithm**, one can enhance the efficiency of database systems and facilitate more efficient querying and sorting of data.

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$$

Figure 166 - Formula

Join Operation

Join operations are a fundamental task in database management systems, allowing us to combine information from different relations based on some common attributes. In this regard, several algorithms have been proposed to efficiently compute the join of relations.

One of the most basic algorithms is the **nested-loop join**, which essentially consists of a pair of **nested for loops**. However, this algorithm is not very efficient, as it examines every pair of tuples in the two relations, resulting in a high computational cost.

To address this issue, the **block nested-loop join** algorithm was proposed, which processes the relations on a **per-block basis** rather than on a **per-tuple basis**. This variant pairs every block of the inner relation with every block of the outer relation and generates all possible pairs of tuples, satisfying the join condition.

```
for each block  $B_r$  of  $r$  do begin
    for each block  $B_s$  of  $s$  do begin
        for each tuple  $t_r$  in  $B_r$  do begin
            for each tuple  $t_s$  in  $B_s$  do begin
                test pair  $(t_r, t_s)$  to see if they satisfy the join condition
                if they do, add  $t_r \cdot t_s$  to the result;
            end
        end
    end
end
```

Figure 167 - Block nested-loop join

The **block nested-loop join** algorithm provides a significant saving in block accesses, as each block in the inner relation is read only once for each block in the outer relation, instead of once for each tuple in the outer relation. Furthermore, the cost of this algorithm can be further optimized by using the smaller relation as the inner relation.

Overall, these algorithms enable us to perform efficient join operations on large-scale relational databases, providing a powerful tool for data analysis and decision-making.

The merge-join algorithm stands tall as a formidable tool for computing natural and equi-joins. Boasting a structure, not unlike the merge stage of the **merge-sort algorithm**, **merge-join offers a quick** and efficient means of joining two relations, **$r(R)$** and **$s(S)$** , sorted on their shared attributes **$R \cap S$** .

Central to the merge-join algorithm is the use of pointers that move through the relations as groups of tuples with identical values on the join attributes are read into sets **Ss**. If each set fits into the main memory, the algorithm proceeds apace; otherwise, the algorithm will perform a **block nested-loop join**, matching the larger sets with corresponding blocks of tuples in **r** with the same join attribute values.

```

 $pr :=$  address of first tuple of  $r$ ;
 $ps :=$  address of first tuple of  $s$ ;
while ( $ps \neq \text{null}$  and  $pr \neq \text{null}$ ) do
  begin
     $t_s :=$  tuple to which  $ps$  points;
     $S_s := \{t_s\}$ ;
    set  $ps$  to point to next tuple of  $s$ ;
     $done := false$ ;
    while (not  $done$  and  $ps \neq \text{null}$ ) do
      begin
         $t_s' :=$  tuple to which  $ps$  points;
        if ( $t_s'[\text{JoinAttrs}] = t_s[\text{JoinAttrs}]$ )
          then begin
             $S_s := S_s \cup \{t_s'\}$ ;
            set  $ps$  to point to next tuple of  $s$ ;
          end
        else  $done := true$ ;
      end
     $t_r :=$  tuple to which  $pr$  points;
    while ( $pr \neq \text{null}$  and  $t_r[\text{JoinAttrs}] < t_s[\text{JoinAttrs}]$ ) do
      begin
        set  $pr$  to point to next tuple of  $r$ ;
         $t_r :=$  tuple to which  $pr$  points;
      end
    while ( $pr \neq \text{null}$  and  $t_r[\text{JoinAttrs}] = t_s[\text{JoinAttrs}]$ ) do
      begin
        for each  $t_s$  in  $S_s$  do
          begin
            add  $t_s \bowtie t_r$  to result;
          end
        set  $pr$  to point to next tuple of  $r$ ;
         $t_r :=$  tuple to which  $pr$  points;
      end
    end.

```

Figure 168 - Merge join

Once the input relations have been sorted, tuples with identical join attribute values are in consecutive order, resulting in each tuple being read only once. Consequently, with just one pass through each file, the **merge-join algorithm** proves to be an economical choice. Assuming that multiple buffer blocks are allocated to each relation, the cost of disk seeks is kept relatively low compared to data transfer.

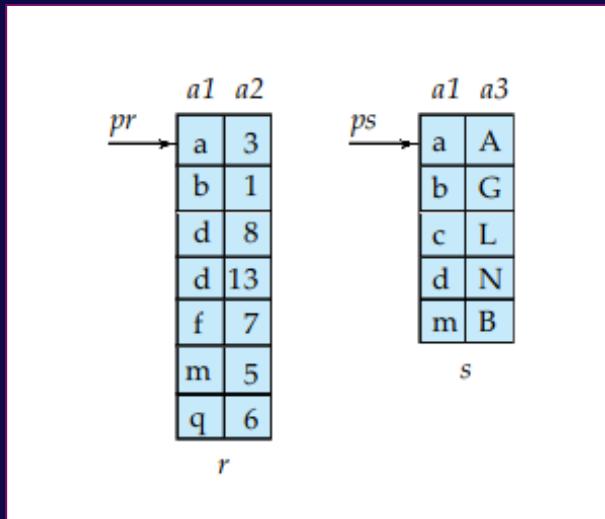


Figure 169 - Sorted relations for merge join

Despite its many advantages, **merge-join** does have its limitations. Should either of the input relations **r** and **s** not be sorted on the join attributes, sorting must be done before the algorithm can be used, adding to the overall cost of computation.

Additionally, if some sets of **Ss** are too large to fit into memory, the algorithm's efficiency may take a hit.

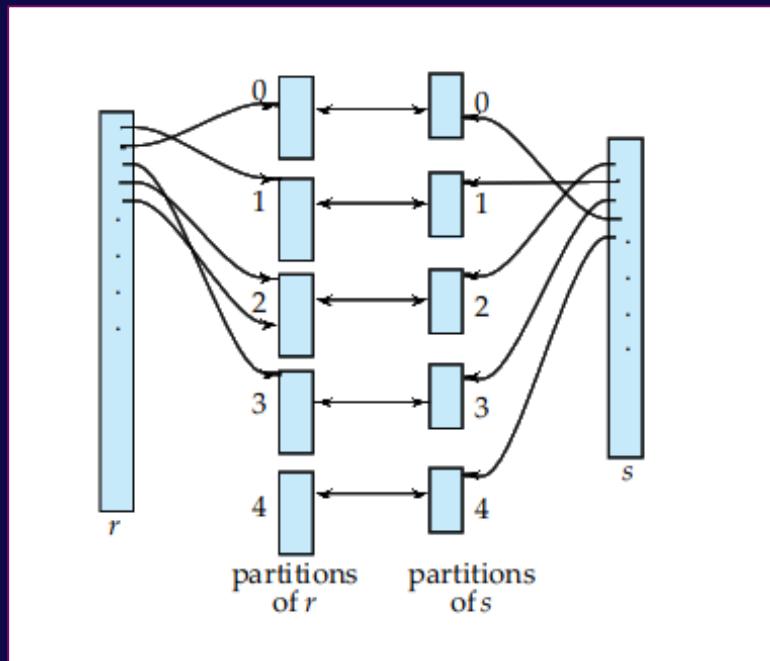


Figure 170 - Hash partitioning of relations

All in all, the merge-join algorithm remains a powerful tool in the database management arsenal, capable of handling natural and equi-joins with ease, provided the relations are sorted and the memory is allocated correctly.

The hash-join algorithm is a powerful technique that facilitates the computation of natural joins of relations r and s . It operates by utilizing a hash function to determine if an r tuple and an s tuple satisfy the join condition. If their values for the join attributes are the same, the r tuple can be found in the partition corresponding to the hash value of the join attribute, while the s tuple can be found in the same partition of the s relation.

This avoids the need to compare r tuples with s tuples in any other partition, thereby enhancing the efficiency of the join operation.

```
/* Partition s */
for each tuple  $t_s$  in  $s$  do begin
     $i := h(t_s[JoinAttrs]);$ 
     $H_{s_i} := H_{s_i} \cup \{t_s\};$ 
end
/* Partition r */
for each tuple  $t_r$  in  $r$  do begin
     $i := h(t_r[JoinAttrs]);$ 
     $H_{r_i} := H_{r_i} \cup \{t_r\};$ 
end
/* Perform join on each partition */
for  $i := 0$  to  $n_h$  do begin
    read  $H_{s_i}$  and build an in-memory hash index on it;
    for each tuple  $t_r$  in  $H_{r_i}$  do begin
        probe the hash index on  $H_{s_i}$  to locate all tuples  $t_s$ 
        such that  $t_s[JoinAttrs] = t_r[JoinAttrs];$ 
        for each matching tuple  $t_s$  in  $H_{s_i}$  do begin
            add  $t_r \bowtie t_s$  to the result;
        end
    end
end
```

Figure 171 - Hash join

The **hash-join algorithm** involves building a **hash index** on each partition of the **s** relation in memory, and then probing it with tuples from the corresponding partition of the **r** relation. This process requires only a single pass through both inputs, making it highly efficient. However, if the number of partitions is greater than the number of available buffer blocks, recursive partitioning is required to split the input into smaller partitions.

Furthermore, the **hash-table** overflow may occur due to the hash index being larger than the main memory. In such cases, the fudge factor is used to increase the number of partitions and reduce the size of each partition to handle the skewness. Overall, the hash-join algorithm is a powerful tool for computing natural joins efficiently and effectively.

Complex joins present a unique challenge. While nested-loop and block nested-loop joins can be applied regardless of join conditions, the more efficient techniques are limited to simple conditions such as natural and equi-joins. However, there is a way to implement joins with complex conditions such as conjunctions and disjunctions by utilizing the techniques developed for handling complex selections.

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

Figure 172 - Example

To illustrate this point, let us consider a join with a conjunctive condition. By applying one of the simpler joins on the individual conditions, we can generate an intermediate result consisting of tuples from both tables. The final join can then be obtained by testing the remaining conditions on this intermediate result.

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

Figure 173 - Example

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

Figure 174 - Example

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

Figure 175 - Example

A disjunctive join can be computed similarly, as the union of the records in individual joins. With this technique, complex joins need not be an insurmountable challenge, but rather a task that can be effectively managed with the proper tools and strategies.

Other Operations

Several fundamental operations play a pivotal role in querying data. Along with the classic selection and join operations, relational databases also offer a handful of other operations that enable users to manipulate their data in novel ways.

One such operation is the elimination of duplicate records. This operation can be efficiently performed through the use of sorting or hashing techniques. While sorting allows for duplicates to be detected and removed as **adjacent copies**, **hashing partitions** the relation and constructs an **in-memory hash index** that can be used to remove any existing duplicates. Projection is another powerful operation that allows for the selection of certain attributes from a relation. When applied, projection can produce a relation with duplicate tuples, which can be efficiently removed using the techniques discussed above.

Set operations such as union, intersection, and the set difference can be implemented through sorting or hashing as well. When sorting is used, both input relations are sorted and scanned only once to produce the desired result. When hashing is used, the two input relations are partitioned by a common hash function, and the respective partitions are then processed based on the desired operation.

Finally, outer join operations, which were previously introduced, can be implemented through two different strategies. One strategy is to compute the corresponding join and add further tuples to the result to get the outer join result. The other strategy is to use the techniques discussed earlier to compute the **left outer join of two input relations**, **r**, and **s**.

These operations are critical to the functionality of a relational database system, and their efficient implementation is key to the performance of such systems.

Evaluation of Expressions

In this piece, we delve into the intricacies of evaluating expressions in the field of database systems. After exploring individual relational operations, we turn our attention to expressions containing multiple operations and the most effective ways to evaluate them.

One approach to evaluating an expression is to execute one operation at a time, in a logical order, and store each intermediate result in a temporary relation for later use. However, this method is not without its drawbacks, particularly when dealing with large data sets that require the creation of sizable temporary relations that must be written to disk.

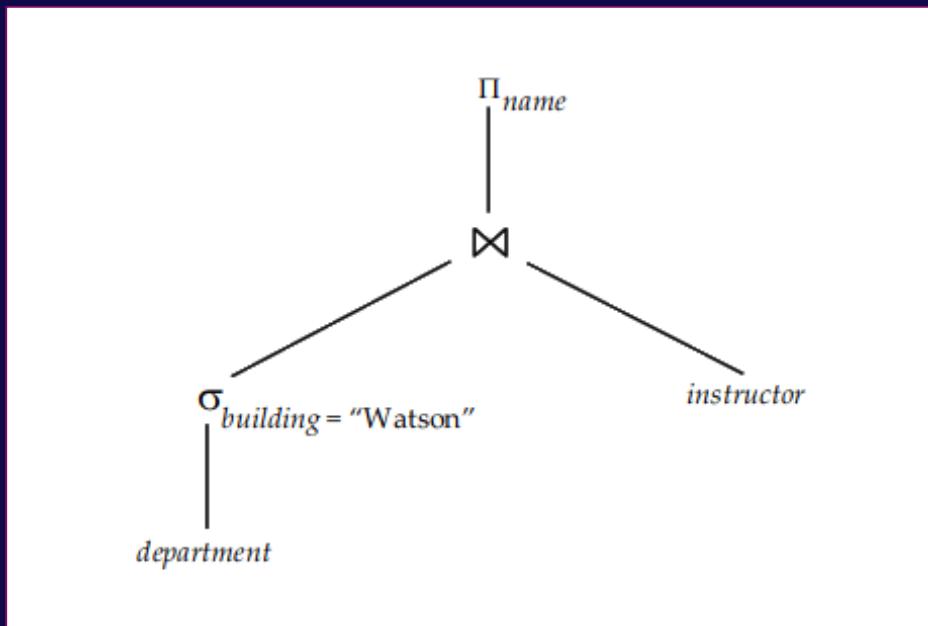


Figure 176 - Pictorial representation of an expression

A more efficient alternative is pipelining, whereby multiple operations are executed simultaneously and the results of each operation are passed directly to the next, without the need for storing intermediate results. This approach is particularly useful when dealing with queries that generate results that need to be displayed in real-time, allowing users to receive query results as they are generated.

However, while pipelining can be a more efficient evaluation method, it is not always feasible, and in some cases, materialization is the only viable option. In addition, it is important to consider the cost of each approach when evaluating expressions, which can differ depending on the specific operations involved.

```
doner := false;  
dones := false;  
r := ∅;  
s := ∅;  
result := ∅;  
while not doner or not dones do  
  begin  
    if queue is empty, then wait until queue is not empty;  
    t := top entry in queue;  
    if t = Endr then doner := true  
    else if t = Ends then dones := true  
    else if t is from input r  
      then  
        begin  
          r := r ∪ {t};  
          result := result ∪ ({t} ⋈ s);  
        end  
    else /* t is from input s */  
      begin  
        s := s ∪ {t};  
        result := result ∪ (r ⋈ {t});  
      end  
  end
```

Figure 178 - Double-pipelined join algorithm

To implement a pipeline, a single complex operation can be constructed that combines the operations involved in the pipeline. However, it is often more desirable to reuse the code for individual operations to create a pipeline.

The evaluation of expressions in database systems is a complex and multi-faceted process that requires careful consideration of the specific operations involved, as well as the most efficient evaluation methods available.

4.4 Query Optimization

Query optimization is a critical task in the realm of database management, and it involves selecting the most efficient approach for processing complex queries. It is not expected that users will write their queries in a manner that is optimized for processing efficiency. Rather, the system is tasked with the responsibility of devising a query evaluation plan that minimizes the cost of query evaluation. This is where the intricate art of query optimization comes into play.

Optimization occurs at various levels, including the **relational-algebra level**, where the system endeavors to identify an expression that is equivalent to the given expression but more efficient to execute. Another level of optimization involves selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation or determining the specific indices to use.

The difference in evaluation time between a good and a bad strategy can be staggering, often spanning several orders of magnitude. Therefore, it is worth investing a considerable amount of time in the selection of a suitable strategy for processing a query, even if it is executed only once. The importance of query optimization cannot be overstated, as it ultimately leads to enhanced performance and productivity in database management.

Overview

In the complex world of database systems, query optimization is a key component of efficient data retrieval. The process involves generating **alternative query-evaluation plans** and **selecting the least-costly** one to produce the same result as the original expression. The query optimizer achieves this by applying logical equivalence rules to generate equivalent expressions and then annotating these expressions in alternative ways to generate various evaluation plans.

To estimate the cost of each plan, the optimizer leverages statistical information about the relations, such as relation sizes and index depths, and uses this data to estimate the costs of individual operations. These costs are combined to determine the estimated cost of evaluating a given relational algebra expression.

Moreover, materialized views are utilized to speed up the processing of certain queries. Maintaining these views and performing query optimization with them is a separate challenge altogether.

While the best way to view the evaluation plan for a given query is through the GUI provided by the database system, command line interfaces offer variations of a command that display the execution plan chosen for a specified query. These commands vary by database, but all display the estimated costs along with the plan.

In short, query optimization is a vital aspect of database systems that requires a deep understanding of equivalence rules, statistics, and materialized views. Through the use of evaluation plans and estimated costs, the query optimizer seeks to reduce the time and resources required for data retrieval and ensure optimal performance of the system.

Transformation of Relational Expressions

As we delve deeper into the world of database querying, we find ourselves exploring the concept of relational expression transformation. This involves the consideration of alternative expressions that generate the same set of tuples as the original relational expression on every legal database instance, albeit with varying evaluation costs. Such alternative expressions are said to be equivalent.

Equivalence rules serve as a guide for the transformation of relational-algebra expressions into other logically equivalent expressions. These rules can be used to replace an expression of the first form with an expression of the second form or vice versa. As a result, the optimizer is better equipped to transform expressions into more optimized, efficient, and logically equivalent ones.

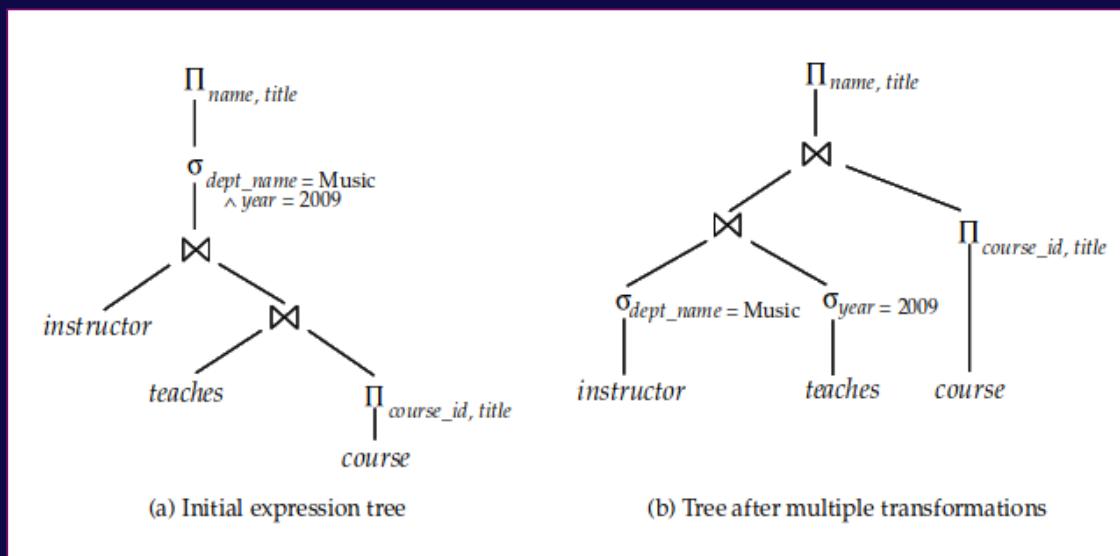


Figure 179 - Multiple transformations

Among the many general equivalence rules governing relational-algebra expressions, the conjunctive selection operation can be deconstructed into a sequence of individual selections, and selection operations are commutative. Furthermore, only the final operations in a sequence of projection operations are needed. Selections can be combined with **Cartesian products** and theta joins, which are commutative, associative, and important for join reordering in query optimization.

The selection operation distributes over the theta-join operation under certain conditions, and the projection operation distributes over the **theta-join operation under other conditions**. Set union and intersection are commutative and associative, while the set difference is not commutative. Additionally, the selection operation distributes over the union, intersection, and set-difference operations.

```
procedure genAllEquivalent(E)
EQ = {E}
repeat
    Match each expression  $E_i$  in EQ with each equivalence rule  $R_j$ 
    if any subexpression  $e_i$  of  $E_i$  matches one side of  $R_j$ 
        Create a new expression  $E'$  which is identical to  $E_i$ , except that
             $e_i$  is transformed to match the other side of  $R_j$ 
        Add  $E'$  to EQ if it is not already present in EQ
until no new expression can be added to EQ
```

Figure 180 - EProcedure to generate all equivalent expressionsxample

As we embark on this journey of database querying and the transformation of relational expressions, let us remember that the end goal is not only to generate the same set of tuples but to do so efficiently, logically, and with precision.

Estimating Statistics of Expression Results

In this section, we delve into the intricate details of estimating statistics for database relations. The aim is to provide a framework for estimating the cost of operations based on their input size and other pertinent information. Our focus lies in exploring the statistical information maintained in database catalogs, and how this information can be harnessed to provide cost estimates for a range of relational operations.

We begin by examining the statistical information stored in database catalogs, which includes the number of tuples and blocks in a relation, the size of a tuple, the blocking factor of a relation, and the number of distinct values that appear in a relation for a particular attribute. This last statistic is particularly useful, as it is the same as the size of the attribute's domain in the relation. We also note that statistics about indices, such as the height of **B+-tree indices** and the **number of leaf** pages in the indices, are also maintained in the catalog.

However, it is important to bear in mind that the estimates we derive from these statistics are not always precise, as they rely on certain assumptions that may not hold in reality. As a result, the lowest estimated execution cost may not necessarily equate to the lowest actual execution cost. Nevertheless, real-world experience has shown that the plans with the lowest estimated costs are often either the lowest actual execution costs or close to them.

To maintain accurate statistics, it is necessary to update them every time a relation is modified. However, updating the statistics incurs a significant amount of overhead, which is why most systems only update statistics during periods of light system load. Consequently, the statistics used to choose a query-processing strategy may not be completely accurate, although they are often sufficient to provide good estimates of the relative costs of different plans.

Furthermore, in addition to the **simplified statistical information** outlined above, real-world optimizers often maintain further statistical information, such as histograms of attribute distributions, to improve the accuracy of cost estimates for evaluation plans. These histograms divide the values for an attribute into a number of ranges, and for each range, associate the number of tuples whose attribute value falls within that range, as well as the number of distinct attribute values in that range.

This section highlights the importance of maintaining accurate statistical information in database catalogs to enable cost estimates for relational operations. The estimation process is not without its challenges, particularly due to the assumptions involved. Nevertheless, the use of statistical information is essential for achieving efficient query-processing strategies in real-world database systems.

We delve into the intricate details of estimating statistics for database relations. The aim is to provide a framework for estimating the cost of operations based on their input size and other pertinent information. Our focus lies in exploring the statistical information maintained in database catalogs, and how this information can be harnessed to provide cost estimates for a range of relational operations.

We begin by examining the statistical information stored in database catalogs, which includes the number of tuples and blocks in a relation, the size of a tuple, the blocking factor of a relation, and the number of distinct values that appear in a relation for a particular attribute. This last statistic is particularly useful, as it is the same as the size of the attribute's domain in the relation. We also note that statistics about indices, such as the height of **B+-tree indices** and the number of **leaf pages** in the indices, are also maintained in the catalog.

However, it is important to bear in mind that the estimates we derive from these statistics are not always precise, as they rely on certain assumptions that may not hold in reality. As a result, the lowest estimated execution cost may not necessarily equate to the lowest actual execution cost. Nevertheless, real-world experience has shown that the plans with the lowest estimated costs are often either the lowest actual execution costs or close to them.

To maintain accurate statistics, it is necessary to update them every time a relation is modified. However, updating the statistics incurs a significant amount of overhead, which is why most systems only update statistics during periods of light system load. Consequently, the statistics used to choose a **query-processing strategy** may not be completely accurate, although they are often sufficient to provide good estimates of the relative costs of different plans.

Furthermore, in addition to the simplified statistical information outlined above, real-world optimizers often maintain further statistical information, such as histograms of attribute distributions, to improve the accuracy of cost estimates for evaluation plans. These histograms divide the values for an attribute into a number of ranges, and for each range, associate the number of tuples whose attribute value falls within that range, as well as the number of distinct attribute values in that range.

The estimation process is not without its challenges, particularly due to the assumptions involved. Nevertheless, the use of statistical information is essential for achieving efficient query-processing strategies in real-world database systems.

Choice of Evaluation Plans

In the world of database optimization, the choice of evaluation plans is crucial in determining the best algorithm for each operation and coordinating the execution of those operations. With statistics estimated by various techniques, cost-based optimizers explore the space of query-evaluation plans equivalent to a given query and choose the one with the least estimated cost. However, when dealing with complex queries, generating all equivalent plans may be too expensive, leading to the use of heuristics that reduce optimization costs, albeit at the risk of not finding the optimal plan.

In SQL queries involving **joins** of a few relations, the optimal join order selection problem arises. For instance, given a join query, the number of different query plans that are equivalent to the query can be quite large, with the number of **join** orders rising quickly as the number of relations increases. Yet, by using dynamic-programming algorithms, we can store the results of computations and reuse them, reducing execution time significantly.

```
procedure FindBestPlan(S)
  if (bestplan[S].cost ≠ ∞) /* bestplan[S] already computed */
    return bestplan[S]
  if (S contains only 1 relation)
    set bestplan[S].plan and bestplan[S].cost based on best way of accessing S
  else for each non-empty subset S1 of S such that S1 ≠ S
    P1 = FindBestPlan(S1)
    P2 = FindBestPlan(S – S1)
    A = best algorithm for joining results of P1 and P2
    cost = P1.cost + P2.cost + cost of A
    if cost < bestplan[S].cost
      bestplan[S].cost = cost
      bestplan[S].plan = “execute P1.plan; execute P2.plan;
                           join results of P1 and P2 using A”
  return bestplan[S]
```

Figure 181 - Dynamic-programming algorithm for join order optimization

Such a dynamic-programming algorithm for finding optimal join orders is recursive, applying selections on individual relations at the earliest possible point, and taking into account all join conditions that relate attributes from the two subexpressions. The algorithm works by checking if the best plan for computing the join of a given set of relations has already been computed, then recording the best way of accessing a relation (taking selections into account) or trying every way of dividing a set into two disjoint subsets, recursively finding the best plans for each of the two subsets and computing the cost of the overall plan by combining the two. The use of associative

arrays to store the evaluation plans computed by the algorithm makes the entire process more efficient and less time-consuming.

The optimization of query evaluation plans can be a daunting and computationally intensive task. The cost-based optimization approach, while effective, can suffer from the drawback of excessive computational costs. To mitigate this issue, heuristic rules can be employed to reduce the cost of optimization. However, the use of such heuristics can lead to suboptimal query evaluation plans, which may be more costly than optimal plans.

To address this issue, a general-purpose cost-based optimizer based on equivalence rules has been proposed. This optimizer leverages the power of physical equivalence rules to generate all possible evaluation plans while utilizing cost estimation techniques to select the optimal plan. However, the efficiency of this approach depends on several factors, including space-efficient expression representation, dynamic programming with memoization, and techniques that avoid generating all possible equivalent plans.

Despite the strengths of this approach, it remains complex and computationally intensive. To address this issue, heuristic rules are commonly employed in optimization. These rules can help to reduce the cost of query evaluation plans by performing selection and projection operations early in the process. While effective in many cases, the use of heuristic rules can lead to suboptimal query evaluation plans, and further heuristics are often employed to optimize the optimization process.

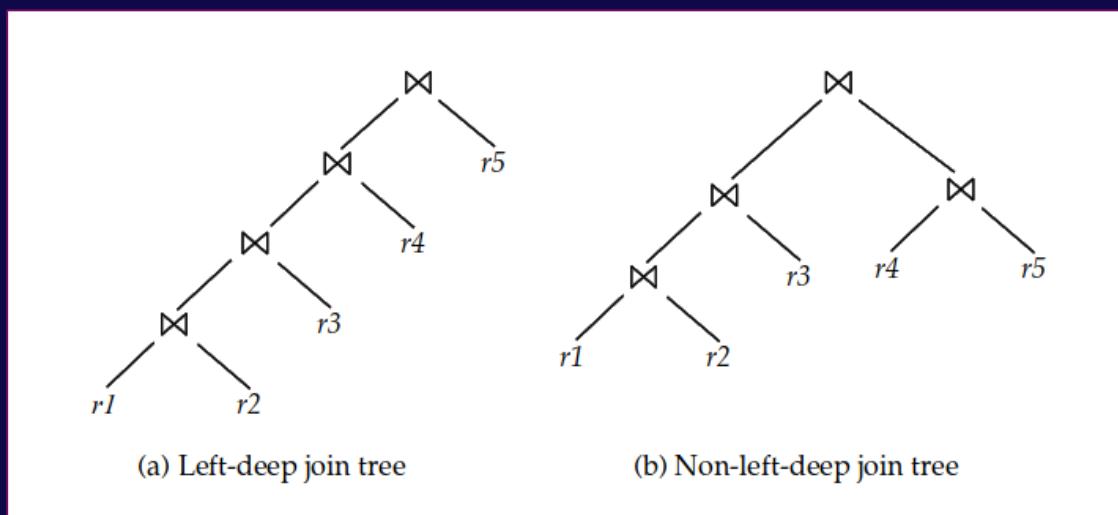


Figure 182 - Left-deep join trees

While cost-based optimization and heuristic rules have proved effective in many cases, the pursuit of optimal query evaluation plans continues to be a fascinating and elusive challenge for researchers and practitioners alike.

Optimizing nested subqueries in SQL can be a tricky task. SQL treats nested subqueries as functions that take parameters and return either a single value or a set of values. These parameters are the variables from an outer-level query that are used in the nested subquery, and they are known as correlation variables. The goal is to find an efficient way to evaluate a query with nested subqueries, since correlated evaluation, which evaluates the subquery for each tuple in the outer-level query, can result in a large number of random disk **I/O** operations.

SQL optimizers try to transform nested subqueries into joins where possible. Join algorithms help avoid expensive random **I/O**, which makes them a more efficient option. However, if the transformation isn't possible, the optimizer keeps the subqueries as separate expressions, optimizes them separately, and then evaluates them through correlated evaluation.

Transforming a nested subquery into a join isn't always straightforward. In some cases, creating a temporary relation that contains the results of the nested query and joining it with the outer-level query is necessary. The process of replacing a nested query with a **join** (*possibly with a temporary relation*) is called decorrelation.

Optimizing complex nested subqueries is a difficult task. While many optimizers do a limited amount of decorrelation, it is best to avoid using complex nested subqueries where possible. We cannot be sure that the query optimizer will succeed in converting them to a form that can be evaluated efficiently.

Materialized Views

Materialized views are a boon to databases, allowing for improved performance in certain applications. Whereas traditional views only store the query defining the view, materialized views go a step further, computing and storing the contents of the view. While this constitutes redundant data, the cost savings from reading a materialized view as opposed to computing the view using its query definition can be significant.

For example, a view providing the total salary for each department may require reading every instructor tuple pertaining to a department and summing the salary amounts, which can be a time-consuming process. However, if the view definition of the total salary amount was materialized, the total salary amount could be found by looking up a single tuple in the materialized view.

However, maintaining materialized views presents a challenge. Whenever the data used in a view definition changes, the materialized view must be kept up-to-date, a task known as view maintenance. While manual code updates can be used to maintain materialized views, this approach is error-prone and can lead to inconsistencies.

A better option for maintaining materialized views is to define triggers on insert, deleting, and update of each relation in the view definition. These triggers can modify the contents of the materialized view to reflect changes in the underlying data. Incremental view maintenance, whereby only affected portions of a materialized view are modified, is another option that modern database systems support.

The changes (*inserts and deletes*) to a relation or expression that can cause a materialized view to becoming out-of-date are referred to as its differential. The incremental maintenance of materialized views can be achieved through simple operations. For instance, to update a materialized view based on a join operation, we simply need to add the tuples that have been inserted into the old contents of the view.

Materialized views can significantly improve database performance, but maintaining their consistency is a challenge. With the support of modern database systems, however, it is possible to perform incremental maintenance of materialized views through simple operations. The incremental maintenance of materialized views is a topic of great importance. Similar to projection operations, aggregation operations in SQL utilize count, sum, avg, min, and max to compute information from materialized views. However, the process of updating these aggregates on insertion or deletion of tuples is not always straightforward.

In the case of the count, a **materialized view $v = \text{AGcount}(B)(r)$** is created to compute the count of attribute **B** after grouping **r** by attribute **A**. On insertion, the system looks for the group **t.A** in the materialized view; if it is not present, a new entry **(t.A, 1)** is added. If the group already exists, the count for the group is incremented. Conversely, on deletion, the count for the group is decremented, and if it becomes zero, the tuple for group **t.A** is removed from the materialized view.

Handling sum aggregates requires an extra count value to distinguish between a group sum of zero and the case where the last tuple in a group is deleted. Similarly, computing the average requires the maintenance of sum and count aggregates, which are used to compute the average as the sum divided by the count.

Materialized views also enable the optimization of query operations, either by rewriting queries to utilize the view or by replacing the view with its definition. The implementation of these techniques can provide significant performance gains for database systems.

Advanced Topics in Query Optimization

In the realm of database management systems, there exist a myriad of opportunities for optimizing queries beyond the rudimentary techniques we have thus far explored. Let us delve into a few of these advanced optimization strategies.

First, let us examine the optimization of **top-K queries**. Often, queries retrieve results sorted according to certain attributes and require only the **top K results**. In some instances, the bound **K** is explicitly specified, while in others, the query may not mention the limit, but the optimizer can be signaled to retrieve only the **top K results**. When **K** is small, a query plan that generates the entire set of results before sorting and selecting the **top K** is highly inefficient, discarding most of the intermediate results that it computes. Consequently, novel approaches to optimizing **top-K** queries have emerged. For instance, one strategy involves using pipelined plans that can generate the results in sorted order. Another technique involves estimating the highest value on the sorted attributes that will appear in the **top-K** output and introducing selection predicates to eliminate larger values. If too few or too many tuples are generated, the selection condition is modified, and the query re-executed.

Next, we consider the optimization of **join minimization**, which arises when queries are generated through views. Occasionally, more relations are joined than needed for the computation of the query, leading to unnecessary computational overheads. In such cases, dropping a relation from a join is a typical example of **join minimization**. This technique can be performed in various situations, resulting in improved query performance.

Another optimization strategy is updating queries, which are notorious for involving subqueries in the set and where clauses, which must be taken into account in optimizing the update. Where an updated tuple may be reinserted in the index ahead of the scan and seen again by the scan, causing erroneous updates.

Lastly, we delve into **multi-query optimization** and shared scans, where a query optimizer can potentially exploit common subexpressions between different queries, evaluating them once and reusing them where required. Complex queries may have subexpressions repeated in different parts of the query, which can be similarly exploited, reducing query evaluation costs. Such optimization is known as **multi-query optimization** and can even outperform common subexpression elimination in certain cases. A judiciously chosen set of query evaluation plans for a batch of queries may provide greater sharing and a lower cost than that offered by choosing the lowest cost evaluation plan for each query.

5 TRANSACTION MANAGEMENT

In the realm of database management, the term "*transaction*" holds paramount importance. It is a collection of operations that form a cohesive unit of work, aimed at achieving a specific goal. Consider, for instance, a simple transfer of money from one account to another. This seemingly trivial task constitutes a transaction consisting of two updates, one to each account. However, the significance of a transaction lies in its atomicity - either all actions must be executed completely or, in the event of a failure, the partial effects of each incomplete transaction must be undone. Additionally, once a transaction is successfully executed, its effects must persist in the database - *a system failure should not result in the database forgetting about a transaction that successfully completed*. This property is referred to as durability.

In a database system where multiple transactions are executing concurrently, the potential for transactions to see inconsistent intermediate states created by updates of other transactions looms large. Such a situation can result in erroneous updates to data stored in the database. Thus, database systems must provide mechanisms to isolate transactions from the effects of other concurrently executing transactions. This property is referred to as isolation.

Taken as a whole, the transaction-management component of a database system allows application developers to focus on the implementation of individual transactions without worrying about the intricacies of concurrency and fault tolerance, thus ensuring the efficient and secure functioning of the database system.

5.1 Transactions

In the world of database management, it is often the case that multiple operations performed by users appear to be a single, cohesive unit. However, beneath the surface lies a collection of distinct operations that must be executed in proper order to maintain database integrity. For instance, when transferring funds from a checking account to a savings account, a user views this as a singular event, but within the database system, it consists of numerous individual operations that must be executed correctly to ensure consistency.

Enter the **concept of transactions**. Transactions refer to a collection of operations that form a single logical unit of work. Ensuring the proper execution of transactions despite system failures is essential, as the execution must either take place in its entirety or not at all. The management of concurrent transaction execution is also critical, as it must be carried out in a way that prevents the introduction of inconsistency into the database.

This chapter introduces the fundamental principles of **transaction processing**. While concurrent transaction processing and recovery from failures are discussed in greater detail, further topics in transaction processing are explored. It is through the proper handling of transactions that database integrity is maintained, allowing for a smooth and reliable user experience.

Transaction Concept

The concept of a transaction is fundamental to the proper functioning of modern database systems. In essence, a transaction is a unit of program execution that accesses and updates various data items, typically initiated by a user program written in a **high-level data manipulation language**. Transactions are defined by statements or function calls, that mark the beginning and end of the transaction.

One of the key features of transactions is their **atomicity**. A transaction is indivisible, and either executes in its entirety or not at all. If a transaction fails for any reason, any changes it made to the database must be undone. This requirement is difficult to enforce, as changes may have been stored in main-memory variables or written to the database and stored on disk.

In addition to atomicity, transactions must also adhere to the properties of consistency, isolation, and durability, often abbreviated as the ACID properties. Transactions must preserve database consistency, even for application-dependent consistency constraints that cannot be stated using SQL constructs. They must also operate in isolation, without interference from concurrently executing database statements. Finally, a transaction's actions must persist across crashes, ensuring durability.

While the ACID properties are essential to the proper functioning of transactions, the isolation property in particular can have a significant adverse effect on system performance. As a result, some applications may compromise on the isolation property.

Nonetheless, the enforcement of the ACID properties remains a critical concern for transaction processing in modern database systems.

Simple Transaction Model

In this piece, we explore the foundations of transactional databases and delve into the ACID properties that underpin their reliability and consistency. Starting with a simple transaction model, we examine how data moves between disk and memory, with a focus on read and write operations that access and update data items in the database. We illustrate these concepts through a basic bank application that involves the transfer of funds between accounts.

To ensure the integrity of transactions, we explore each of the ACID properties in turn, beginning with **Consistency**, which demands that the sum of **A** and **B** remains unchanged throughout the execution of the transaction. Atomicity, on the other hand, requires that all actions of the transaction are either reflected in the database, or none are. To achieve this, the database system keeps track of the old values of any data that a transaction writes to, allowing for a full rollback in the event of a system failure.

```
Ti : read(A);  
      A:= A - 50; write(A);  
      read(B);  
      B:=B+ 50;  
      write(B).
```

Durability, meanwhile, ensures that once a transaction completes successfully, all updates it made to the database persist, even in the event of a system failure. We assume that data written to disk are never lost, but for protection against data loss in memory, we rely on backup and recovery systems.

Throughout this exploration of transactional databases, we emphasize the importance of ensuring consistency, atomicity, and durability, and highlight the crucial role played by application programmers and database systems in guaranteeing the reliability and safety of these essential systems.

Storage Structure

To grasp the fundamental concepts of atomicity and durability in transactions, we must delve into the intricate world of storage structures. The chapter has already shed light on the various types of storage media, such as volatile and nonvolatile storage, and their relative speeds, capacities, and resiliencies to failure. In addition, we introduce a new type of storage known as stable storage.

Volatile storage refers to information that is typically lost in the event of system crashes. This includes memory and cache memory. Although access to volatile storage is speedy and direct, the loss of data during system crashes is a major drawback. Nonvolatile storage, on the other hand, is capable of surviving system crashes, but it tends to be slower than volatile storage, particularly for random access. Common examples of nonvolatile storage include secondary storage devices such as magnetic disks and flash storage, as well as tertiary storage devices such as optical media and magnetic tapes. However, both secondary and tertiary storage devices are also susceptible to failure, which can lead to data loss.

Stable storage, the most desirable form of storage, ensures that data is never lost. However, the attainment of stable storage is not absolute, since a catastrophic event such as a black hole could potentially destroy all data. Nonetheless, techniques exist that can approximate stable storage by replicating information across multiple nonvolatile storage media, usually disks, with independent failure modes. Careful updates must be made to prevent information loss during the process of updating stable storage. While the distinctions among different storage types may not always be crystal clear, certain systems such as **RAID controllers** with battery backup can provide some main memory with protection against system crashes and power failures.

To achieve transaction durability, it is necessary to write changes to stable storage. In addition, log records must be written to stable storage before any changes are made to the database on disk in order to ensure transaction atomicity. However, the level of durability and atomicity offered by a system depends largely on the stability of its stable storage implementation. While some applications may only require a single copy on disk, others with valuable data and important transactions may necessitate multiple copies or a closer approximation of the concept of stable storage.

Transaction Atomicity and Durability

In the world of database management, ensuring the atomicity and durability of transactions is paramount. A transaction that fails to complete successfully, or in other words, an aborted transaction, must have no impact on the state of the database. To achieve this, the changes made by an aborted transaction must be undone, a process commonly known as rolling back the transaction. The responsibility of **managing transaction** aborts usually falls on the recovery scheme, which maintains a log of all database modifications made by a transaction.

A successful transaction has been committed, and once a transaction has been committed, it transforms the database into a new consistent state that must persist even in the event of a system failure. However, the effects of a committed transaction cannot be undone by aborting it, and the only recourse is to execute a compensating transaction. This responsibility, however, is left to the user, and the database system does not handle it.

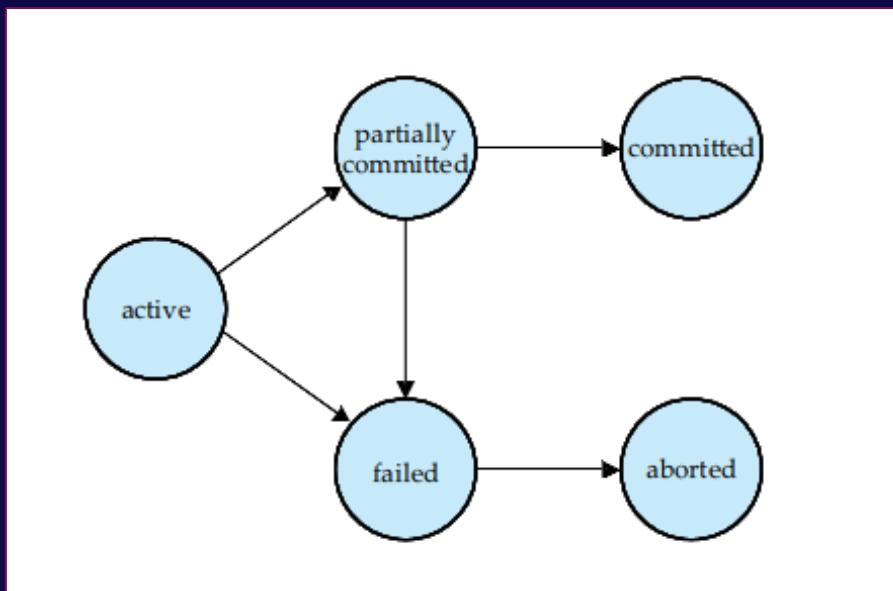


Figure 183 - State diagram of a transaction

To ensure clarity in understanding what constitutes a successful transaction, a **simple abstract transaction model** has been established. This model defines a transaction as being in one of five states: *active*, *partially committed*, *failed*, *aborted*, and *committed*. A transaction starts in the *active* state, and after executing its final statement, enters the *partially committed* state.

The database system then writes enough information to disk that even in the event of a failure, the updates made by the transaction can be re-created when the **system restarts after the failure**. Once the last of this information is written out, the transaction enters the committed state.

In the event of a failure, a transaction enters the failed state and must be rolled back, after which it enters the aborted state. The system then has two options: *it can restart the transaction or kill it*.

Dealing with observable external writes such as writes to *a user's screen or sending emails requires caution*, as such writes cannot be erased once they have occurred. Most systems allow such writes only after the transaction has entered the committed state.

Handling external writes can be more complicated in certain situations, such as the dispensing of cash at an automated teller machine.

Transaction Isolation

In the realm of transaction-processing systems, the issue of consistency amidst concurrent execution is a tricky one. While serial execution of transactions can ensure consistency, allowing concurrency can greatly enhance system throughput and reduce waiting time. However, the execution of multiple transactions concurrently risks violating the isolation property and compromising the integrity of the database. In this regard, a **system of schedules** is presented to identify those executions that can ensure the isolation property and maintain database consistency.

```
T2: read(A);  
    temp := A *0.1;  
    A := A – temp;  
    write(A);  
    read(B);  
    B := B + temp;  
    write(B).
```

Concurrency-control schemes are employed by the database system to regulate the interaction between concurrent transactions and prevent any destruction of database consistency. As current trends in computing make it possible for an increasing number of transactions to run concurrently, it becomes more crucial for database systems to make use of multiple processors and cores to enhance overall system performance.

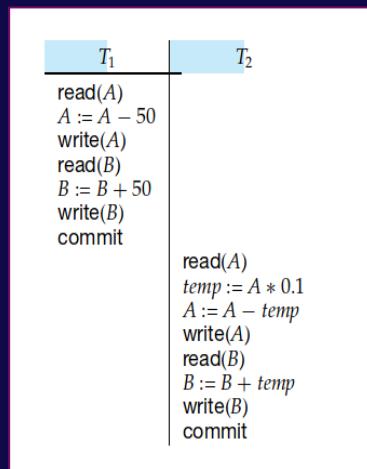


Figure 184 - Schedule 1—a serial schedule in which T1 is followed by T2

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

Figure 185 - Schedule 2—a serial schedule in which T_2 is followed by T_1

To illustrate the concept of schedules and the effects of concurrent execution, we take a simplified banking system with several accounts and transactions that access and update those accounts. Consider two transactions, T_1 and T_2 , that transfer funds from one account to another.

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

T_1	T_2
<pre> read(B) B := B + 50 write(B) commit </pre>	<pre> read(B) B := B + temp write(B) commit </pre>

Figure 186 - Schedule 3—a concurrent schedule equivalent to schedule 1

While executing multiple transactions concurrently can be beneficial, it's important to ensure the preservation of the sum of the amounts in the accounts involved. This can be achieved by employing the appropriate **concurrency-control mechanisms** and ensuring the execution of transactions that are guaranteed to maintain the **isolation property**.

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

Figure 187 - Schedule 4—a concurrent schedule resulting in an inconsistent state

Serializability

In exploring the topic of serializability in database systems, we encounter the challenge of determining when a schedule is **serializable**, particularly when multiple transactions are interleaved. As transactions are essentially programs, it can be difficult to discern the precise operations that they perform and how these operations interact with those of other transactions. However, by considering only read and write instructions, we can simplify the analysis of schedules and focus on a particular form of schedule equivalence known as conflict serializability.

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

Figure 188 - Schedule 3—showing only the read and write instructions

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

Figure 189 - Schedule 5—schedule 3 after swapping of a pair of instructions

To determine whether two consecutive instructions in a schedule conflict with one another, we must ascertain whether they are operations performed by different transactions on the same data item, with at least one of them being a write operation. Depending on the type of instructions involved, the relative order of execution may or may not matter.

For instance, if both instructions are read operations, then their order is **inconsequential**. On the other hand, if one instruction is a read operation and the other is a write operation, the order of execution can significantly impact the final value of the data item in the database state.

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

Figure 190 - Schedule 6—a serial schedule that is equivalent to schedule 3

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

Figure 191 - Schedule 7

To illustrate the concept of conflicting instructions, we analyze several schedules and demonstrate how we can transform them into equivalent schedules that are serializable. By swapping the order of **non-conflicting instructions**, we can create new schedules that preserve the same order of execution for all instructions except for the conflicting ones.

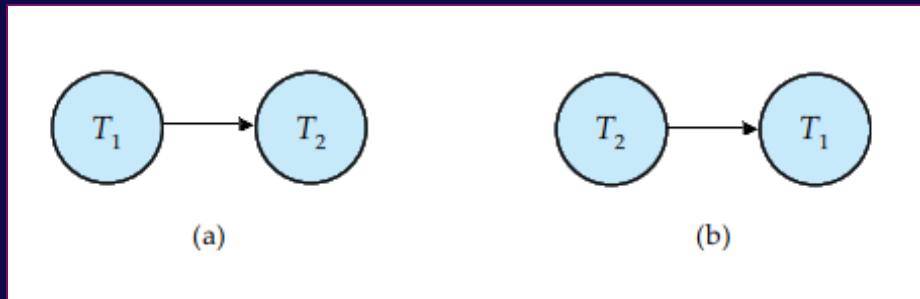


Figure 192 - Precedence graph for (a) schedule 1 and (b) schedule 2

Through this process, we can prove that a given schedule is equivalent to a serial schedule, indicating that it will produce the same final system state as a serial schedule would, regardless of the initial system state.

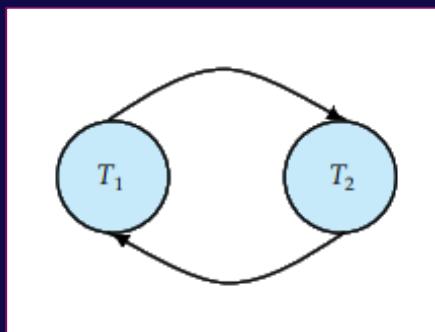


Figure 193 - Precedence graph for schedule 4

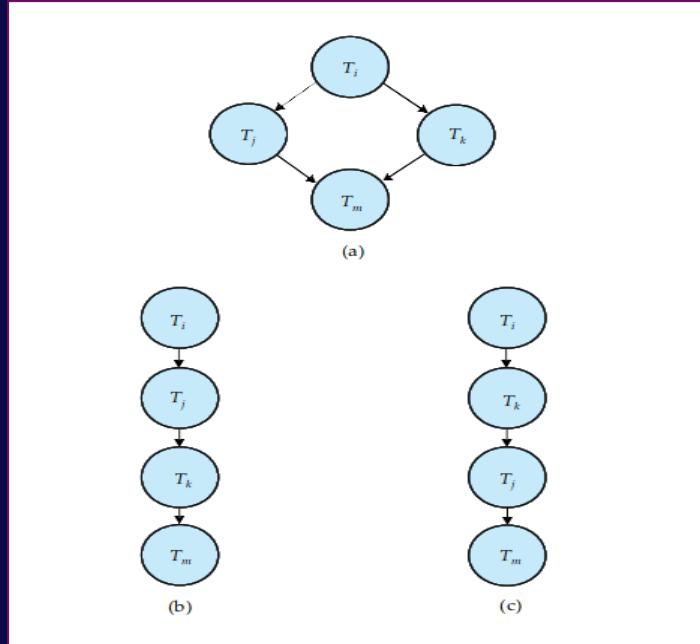


Figure 194 - Illustration of topological sorting

T_1	T_5
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(B)$ $B := B - 10$ $\text{write}(B)$
$\text{read}(B)$ $B := B + 50$ $\text{write}(B)$	$\text{read}(A)$ $A := A + 10$ $\text{write}(A)$

Figure 195 - Schedule 8

By focusing on conflict serializability and the order of read and write instructions in schedules, we can better understand how **concurrency-control mechanisms** can ensure **serializability** in database systems.

Transaction Isolation and Atomicity

In the realm of database systems, the issue of transaction failures during concurrent execution cannot be ignored. To ensure the atomicity property of a transaction, we must address the ramifications of such failures. Specifically, if a transaction fails, it is imperative to undo the effects of that transaction and abort any dependent transactions. This necessitates the placement of restrictions on the types of schedules that are permitted in a system.

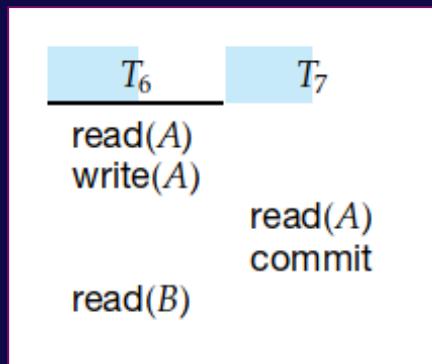


Figure 196 - Schedule 9, a nonrecoverable schedule

One such restriction is that of **recoverable schedules**, where the commit operation of a transaction that has written data must appear before the commit operation of a transaction that reads that same data. If such a schedule is not adhered to, we may find ourselves in a situation where it is impossible to recover from a transaction failure.

Even when a schedule is recoverable, we may still encounter the undesirable phenomenon of cascading rollback. This occurs when a single transaction failure necessitates the rollback of several transactions that have read data written by the failed transaction. To prevent such situations, we must strive for **cascadeless schedules**, where the commit operation of a transaction appears before any read operation of a transaction that has read data written by the first transaction.

T_8	T_9	T_{10}
$\text{read}(A)$		
$\text{read}(B)$		
$\text{write}(A)$		
	$\text{read}(A)$	
	$\text{write}(A)$	
abort		
		$\text{read}(A)$

Figure 197 - Schedule 10

By restricting schedules to only those that are both recoverable and cascadeless, we can minimize the risk of transaction failures and ensure the atomicity property of our transactions. Such restrictions may seem onerous at first, but the potential consequences of failing to adhere to them are too dire to ignore.

Transaction Isolation Levels

As we delve deeper into the world of databases, the concept of transaction isolation levels comes to the forefront. It is a crucial topic that demands the utmost attention of programmers and designers alike. **Serializable** execution is the gold standard, as it ensures that database consistency is maintained, even during concurrent transactions. However, implementing such strict protocols may hamper the performance of certain applications, rendering weaker levels of consistency a more viable option. This, in turn, puts an additional burden on programmers to ensure the correctness of the database.

The **SQL standard** provides a transaction with the option to execute in a **non-serializable** fashion with respect to other transactions. This is achieved by setting the isolation level to "*read uncommitted*," which allows transactions to read uncommitted data items. This feature is beneficial for long transactions whose results do not require precision, and whose serializable execution might interfere with other transactions, causing a delay in their execution.

The **SQL standard** defines four isolation **levels**, namely **Serializable**, **Repeatable read**, **Read committed**, and **Read uncommitted**. Each of these levels offers varying degrees of consistency and concurrency, and programmers can set the level explicitly, or accept the system's default setting.

Implementing a weaker isolation level may seem like a shortsighted decision, as it may risk database consistency for performance. However, in certain cases, the inconsistency that may occur is not relevant to the application, making this trade-off worthwhile. While there are various means of implementing isolation levels, as long as the implementation ensures serializability, the database designer or user need not delve into the details of the implementation, except for dealing with performance issues.

In the real world, we often encounter situations where consistency is not the top priority. For instance, a user browsing a website may see an item in stock, only to find out during checkout that the item is no longer available. Similarly, while booking seats for air travel, the seat availability shown to a traveler is a snapshot of the availability at the time of selection, which may change during the booking process. In such scenarios, nonrepeatable reads are acceptable, and weaker isolation levels can be implemented without worrying about database consistency.

Implementation of Isolation Levels

In the realm of database management, the implementation of isolation levels is of paramount importance in ensuring the consistency and reliability of concurrent transactions. To provide **maximum concurrency** while guaranteeing the *conflict* and *view serializability, recoverability, and cascadelessness* of all possible schedules, various concurrency-control policies have been devised.

One such policy is locking, where transactions acquire locks only on the data items they access, rather than the entire database, allowing other transactions to execute concurrently on other data items. The **two-phase locking protocol** is a commonly used technique that ensures serializability. Another technique, timestamps, assigns each transaction a timestamp that determines the order in which they access data items, with offending transactions being aborted and restarted if their accesses conflict.

The implementation of multiple versions of data items, known as **multi-version concurrency control**, allows transactions to read old versions of data items rather than newer versions written by uncommitted transactions or those that should come later in the serialization order. Snapshot isolation, a widely used technique in this regard, provides each transaction with its own **snapshot of the database**, ensuring that read-only transactions never need to wait, which is a significant source of performance improvement as compared to locking. However, it also presents the challenge of providing too much isolation, where two transactions may not see the updates made by each other.

The innovative concurrency-control policies devised to ensure the conflict and *view serializability, recoverability, and cascadelessness* of all possible schedules demonstrate the ingenuity of the human mind in finding solutions to complex problems. Whether it is **locking, timestamps, or multi-version concurrency-control**, the goal remains the same - to provide **maximum concurrency** while ensuring the consistency and reliability of concurrent transactions.

Transactions as SQL Statements

In an age where data is king, the importance of transactions in SQL statements cannot be understated. Ensuring the ACID properties for transactions is critical to maintaining the integrity of the database. However, when transactions are specified as a sequence of SQL statements, issues arise that were not present in the simpler model of simple reads and writes.

In our simplistic model, we assumed a set of data items already existed and **data-item values** could be changed, but not created or deleted. SQL, on the other hand, allows for insert and delete statements that **create** and **delete data**, respectively. These write operations can change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model.

Consider the following SQL query on our university database that finds all instructors who earn more than **\$90,000**:

```
from instructor  
where salary > 90000;
```

If a user were to insert a new instructor named “*James*” whose salary is **\$100,000**, the result of our query would differ depending on whether the insert occurred before or after the query was executed. This situation is referred to as the phantom phenomenon, as a conflict may exist on “*phantom*” data that was not present at the time of the query.

To address this issue, it is necessary to consider not only the tuples accessed by a transaction but also the information used to find the tuples. This information could be updated by an insertion or deletion, or even an update to a **search-key attribute** in an index.

Thus, while SQL provides a powerful tool for manipulating data, its implications for concurrency control cannot be overlooked. The existence of a conflict may depend on a **low-level query processing decision** by the system that is unrelated to a **user-level view** of the meaning of the two SQL statements. Predicate locking, while expensive and not commonly used in practice, offers an alternative approach to concurrency control by treating an **insert**, **delete**, or **update** as conflicting with a predicate on a relation if it could affect the set of tuples selected by the predicate.

5.2 Concurrency Control

Concurrency control is a crucial aspect of database management that ensures the isolation property of transactions even when they are executed concurrently.

This vital control is maintained through various mechanisms known as **concurrency-control schemes**. The current chapter examines concurrency-control schemes that allow for non-serializable schedules, while the chapter will address the issue of system recovery from failures.

While there is no clear consensus on the best concurrency-control scheme, the most frequently utilized schemes are **snapshot isolation** and **two-phase locking**. This chapter focuses on the administration of concurrently executing transactions, assuming no failures occur, and highlights the advantages of each concurrency-control scheme.

Lock-Based Protocols

Lock-based protocols are a common method used to ensure mutual exclusion and isolation in database transactions. This involves allowing a transaction to access a data item only if it is currently holding a lock on that item. Locking can be done in two modes - shared and exclusive.

	S	X
S	true	false
X	false	false

Figure 198 - Lock-compatibility matrix comp

- **Shared-mode lock** allows a transaction to *read*, but *not write*.
- **Exclusive-mode lock** allows a transaction to both *read* and *write* the data item.

```
T1: lock-X(B);  
      read(B);  
      B := B - 50;  
      write(B);  
      unlock(B);  
      lock-X(A);  
      read(A);  
      A := A + 50;  
      write(A);  
      unlock(A).
```

Figure 199 - Transaction T1

A compatibility function can be defined for **lock modes** and represented as a matrix. Shared mode is compatible with shared mode but not with exclusive mode. Transactions must request an appropriate **lock mode** depending on the operations to be performed on a data item. If the data item is already locked by another transaction in an **incompatible mode**, the **concurrency-control manager** will not grant the lock until all incompatible locks held by other transactions have been released.

```

 $T_2:$  lock-s( $A$ );
    read( $A$ );
    unlock( $A$ );
    lock-s( $B$ );
    read( $B$ );
    unlock( $B$ );
    display( $A + B$ ).
  
```

Figure 200 - Transaction T_2

T_1	T_2	concurrency-control manager
lock-x(B)		
read(B)		grant-x(B, T_1)
$B := B - 50$		
write(B)		
unlock(B)		
	lock-s(A)	grant-s(A, T_2)
	read(A)	
	unlock(A)	
	lock-s(B)	grant-s(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-x(A)		
read(A)		grant-x(A, T_1)
$A := A - 50$		
write(A)		
unlock(A)		

Figure 201 - Schedule 1

An example is given to illustrate the importance of locking and unlocking data items correctly. In a banking scenario, two transactions **T1** and **T2** are executed. If they are executed serially, the correct output is obtained.

```
T3: lock-X(B);
      read(B);
      B := B - 50;
      write(B);
      lock-X(A);
      read(A);
      A := A + 50;
      write(A);
      unlock(B);
      unlock(A).
```

Figure 202 - Transaction T3 (transaction T1 with unlocking delayed)

```
T4: lock-S(A);
      read(A);
      lock-S(B);
      read(B);
      display(A + B);
      unlock(A);
      unlock(B).
```

Figure 203 - Transaction T4 (transaction T2 with unlocking delayed)

However, if they are executed concurrently, then an inconsistent state can be observed if **T1** unlocks a data item too early. Delaying unlocking until the end of the transaction can prevent such issues.

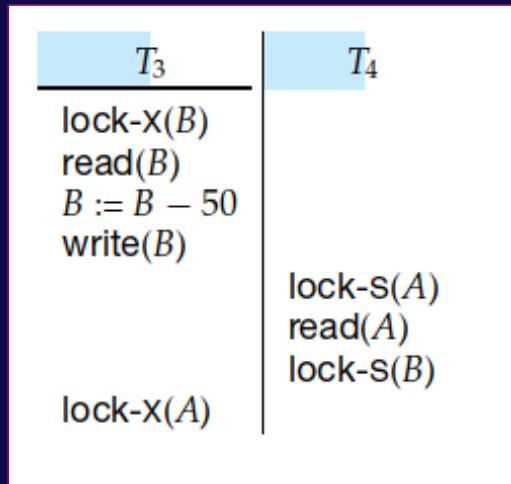


Figure 204 - Schedule 2

Overall, **lock-based protocols** are an effective way to ensure mutual exclusion and isolation in database transactions. Proper use of lock modes and careful locking and unlocking of data items can prevent inconsistent states from occurring.

The concept of **transaction-locking protocols** has garnered much attention from both academics and industry professionals alike. The reason for this interest is twofold:

1. Locking protocols are necessary to ensure the correctness of concurrent access to data items by multiple transactions.
2. Locking protocols enable a database system to enforce a level of isolation between transactions that prevents undesirable phenomena such as *lost updates*, *dirty reads*, and *non-repeatable reads*.

One key aspect of transaction locking protocols is the notion of conflict serializability.

This property ensures that all legal schedules of transactions under a given locking protocol are equivalent to a serial schedule, that is, a schedule in which transactions execute one after the other, without overlapping. To achieve conflict serializability, locking protocols restrict the number of possible schedules that can arise during the execution of a set of transactions.

The **two-phase locking protocol** is a locking protocol that ensures conflict serializability.

Under this protocol, transactions are required to **issue lock** and **unlock requests** in two distinct phases:

- Growing phase
- Shrinking phase

In the growing phase, a transaction can obtain locks on data items but cannot release any locks. In the shrinking phase, a transaction can release locks but cannot obtain any new locks.

```
T8: read(a1);
    read(a2);
    ...
    read(an);
    write(a1).

T9: read(a1);
    read(a2);
    display(a1 + a2).
```

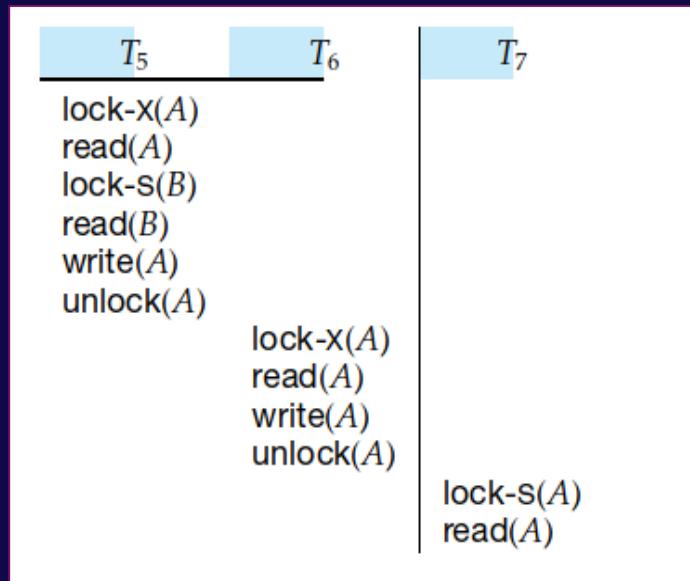


Figure 205 - Partial schedule under two-phase locking

While the **two-phase locking protocol** is an effective mechanism for ensuring conflict serializability, care must be taken to avoid scenarios in which a transaction becomes starved, that is, unable to make progress due to the actions of other transactions.

One way to avoid starvation is to **grant locks** in a manner that prioritizes earlier lock requests over later ones.

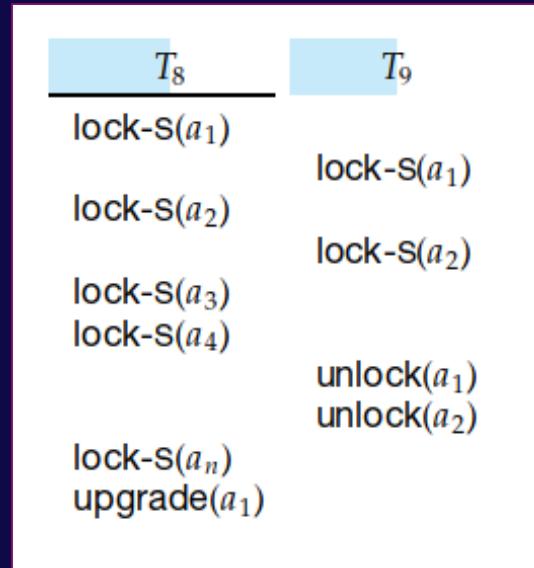


Figure 206 - Incomplete schedule with a lock conversion

Transaction locking protocols are an essential tool for ensuring the correctness and isolation of concurrent transactions in a database system. The **two-phase locking protocol** is one such protocol that guarantees conflict serializability, while also requiring careful consideration of lock-granting mechanisms to avoid transaction starvation.

Locking protocols lie at the heart of database concurrency control, providing a critical mechanism for ensuring the consistency and integrity of transactions operating on shared data.

While strict **two-phase locking** and **rigorous two-phase locking** with lock conversions have found widespread adoption in commercial database systems, there are situations in which these protocols may not be sufficient to ensure conflict serializability. To overcome this, one may either require additional information about the transactions or impose a structural ordering on the database items.

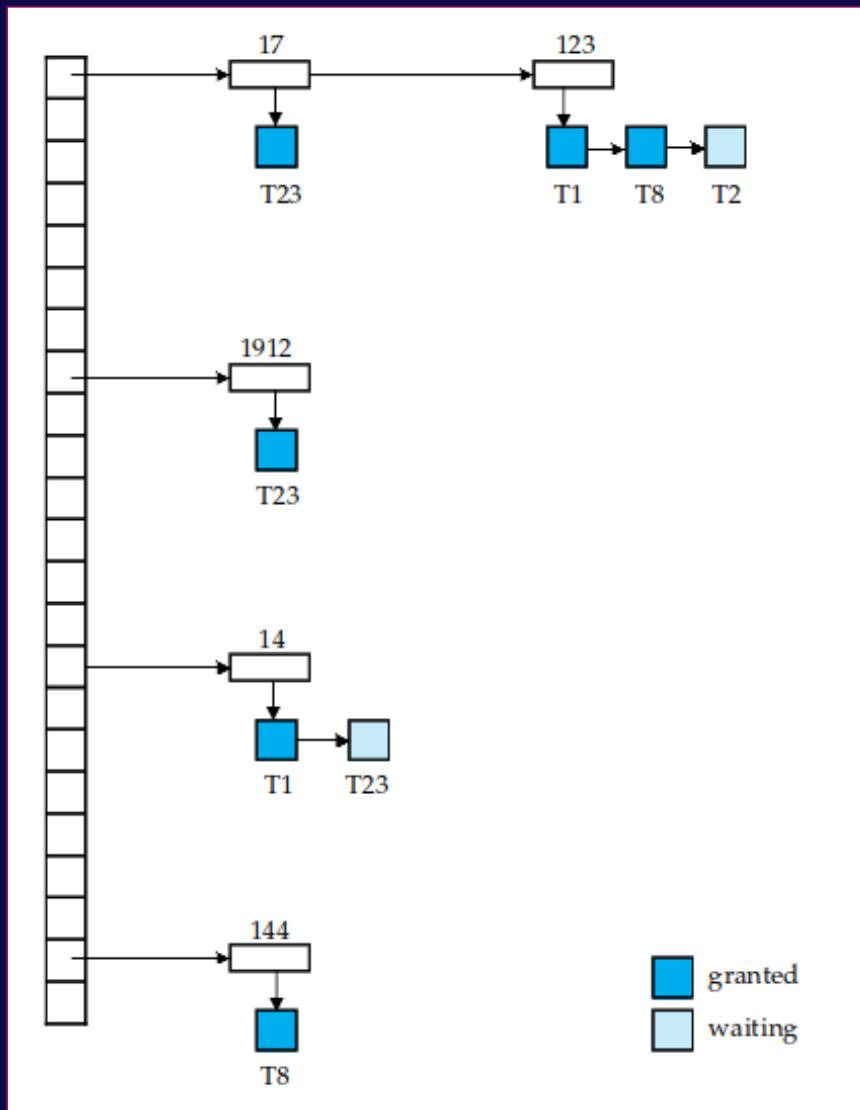


Figure 207 - Lock table

A **well-designed lock manager** is essential for implementing these protocols. It acts as a centralized hub for managing requests for locks on *database items*, *sending messages in reply to transactions*, and *maintaining the current state of each data item*.

The **lock manager** accomplishes this by maintaining a lock table, which contains locks for each data item along with a linked list of records indicating the transaction that made the request, the lock mode requested, and the status of the request.

```
T10: lock-X(B); lock-X(E); lock-X(D);  
      unlock(B); unlock(E);  
      lock-X(G);  
      unlock(D); unlock(G).  
  
T11: lock-X(D); lock-X(H);  
      unlock(D); unlock(H).  
  
T12: lock-X(B); lock-X(E);  
      unlock(E); unlock(B).  
  
T13: lock-X(D); lock-X(H);  
      unlock(D); unlock(H).
```

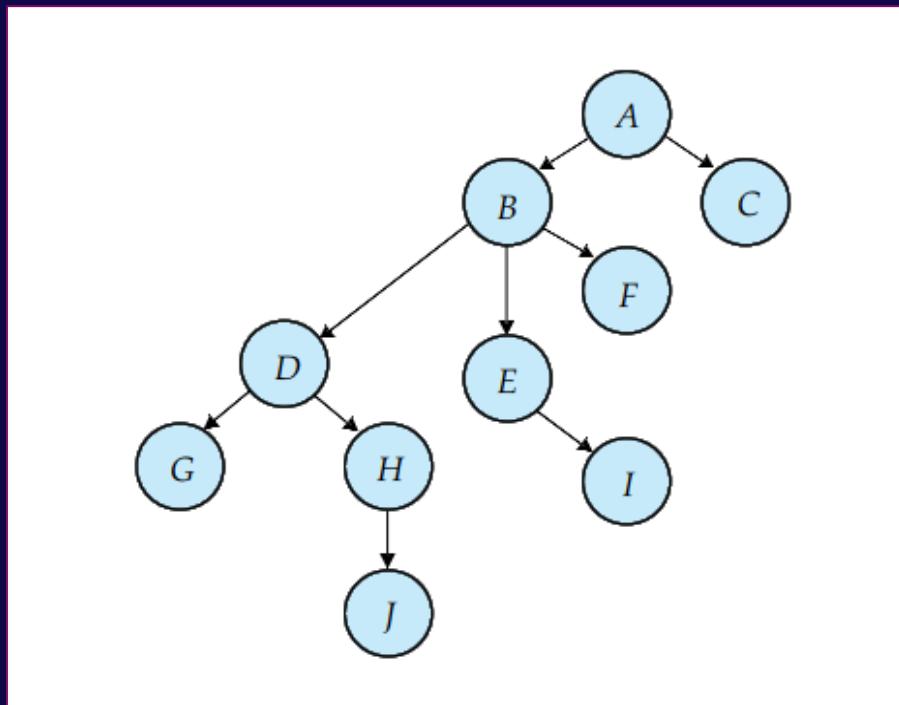


Figure 208 - Tree-structured database graph

The **lock manager processes** requests by adding a record to the appropriate linked list when a **lock request message arrives**, granting requests only if they are compatible with current **locks**, and **deleting records** when a transaction unlocks a data item or aborts. This algorithm guarantees freedom from starvation for lock requests, although it does not prevent deadlocks.

T_{10}	T_{11}	T_{12}	T_{13}
$\text{lock-X}(B)$		$\text{lock-X}(D)$ $\text{lock-X}(H)$ $\text{unlock}(D)$	
$\text{lock-X}(E)$ $\text{lock-X}(D)$ $\text{unlock}(B)$ $\text{unlock}(E)$			$\text{lock-X}(B)$ $\text{lock-X}(E)$
	$\text{unlock}(H)$		
$\text{lock-X}(G)$ $\text{unlock}(D)$			$\text{lock-X}(D)$ $\text{lock-X}(H)$ $\text{unlock}(D)$ $\text{unlock}(H)$
		$\text{unlock}(E)$ $\text{unlock}(B)$	
$\text{unlock}(G)$			

Figure 209 - Serializable schedule under the tree protocol

To develop protocols that are **not two-phase**, we need additional information about how transactions will access the database. **Graph-based protocols** provide a useful approach in this regard, enabling the construction of locking protocols that ensure conflict serializability. By imposing partial ordering on the set of data items, transactions can access database items in a structured manner that facilitates efficient concurrency control.

Deadlock Handling

When a system is in a **deadlock**, it means that a set of transactions have been caught in a deadlock loop, with each waiting for another in the set to relinquish a data item that it needs to proceed. The result is a stalemate, with none of the transactions being able to make any headway.

The only way out of such a situation is to take drastic action, such as **rolling back** some of the transactions involved in the deadlock. This rollback may be partial, with a transaction being rolled back to the point where it obtained a lock whose release resolves the deadlock.

There are two main methods for dealing with the deadlock problem: **prevention and detection** and **recovery**. The former ensures that the system will never enter a deadlock state by ordering requests for locks or requiring all locks to be acquired together. The latter allows the system to enter a deadlock state and then tries to recover from it using a detection and recovery scheme. Both methods may result in a transaction rollback.

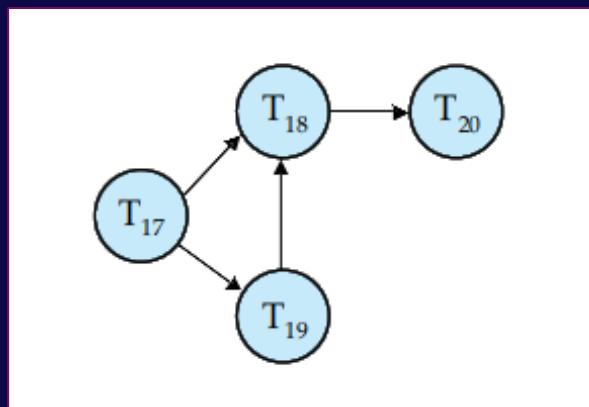


Figure 210 - Wait-for graph with no cycle

One approach to **deadlock prevention** involves ordering all data items and requiring a transaction to lock data items only in a sequence consistent with the ordering. Another approach uses **preemption and transaction rollbacks** to prevent deadlocks. In this approach, a transaction requesting a lock held by another transaction may preempt the lock by rolling back the other transaction and granting the lock.

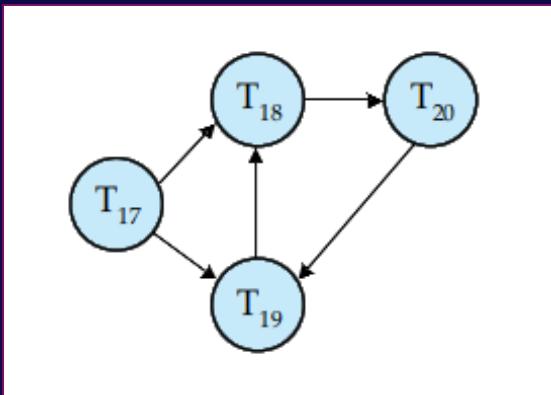


Figure 211 - Wait-for graph with a cycle

Each approach has its own set of advantages and disadvantages, with unnecessary rollbacks being a common problem in many schemes. However, with careful consideration and implementation, the chances of a system being caught in a deadlock can be greatly reduced, and a smooth and efficient operation can be achieved.

Multiple Granularity

The concept of multiple granularity in **concurrency-control schemes** was introduced to address the need for grouping several data items and treating them as one individual synchronization unit. This ingenious mechanism enables the system to define a hierarchy of data granularities, where the smaller granularities are nested within larger ones, as graphically represented by a tree.

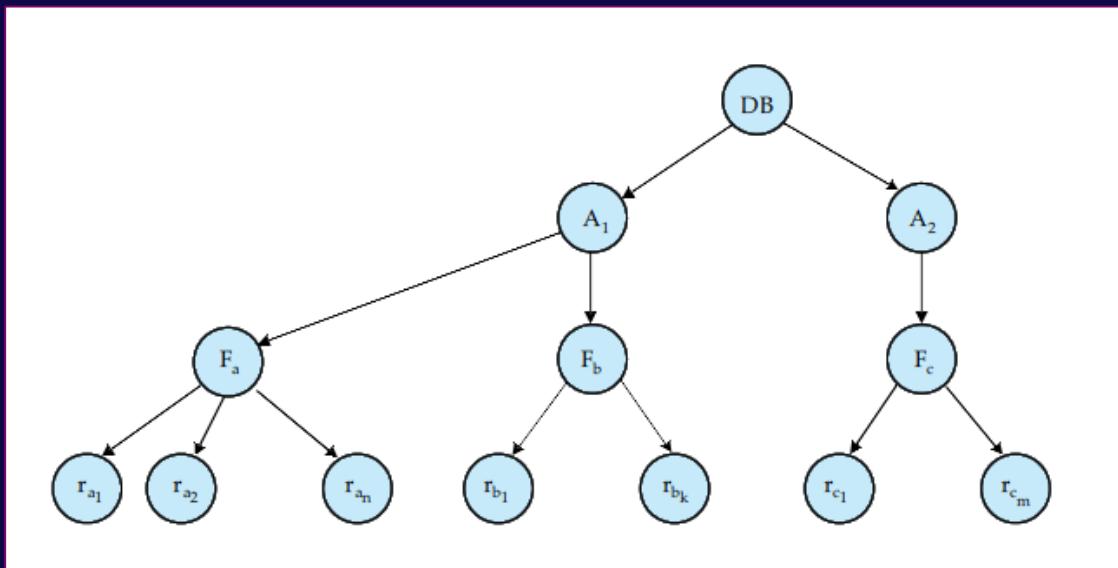


Figure 212 - Granularity hierarchy

Each nonleaf node of this tree, which is significantly different from that used by the tree protocol, represents the data associated with its descendants.

The tree of the figure, for example, consists of four levels of nodes that represent the entire *database*, *areas*, *files*, and *records*.

One of the primary benefits of this **multiple-granularity hierarchy** is that a transaction can lock each node individually using shared and exclusive lock modes, thus ensuring serializability. The locking of a node in either mode also implies that all of its descendants are implicitly locked in the same mode.

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

Figure 213 - Compatibility matrix

To ensure **compatibility** among the various lock modes, a new class of lock modes, called intention lock modes, was introduced. These modes allow explicit locking to be done at a lower level of the tree while also enabling a transaction to traverse the tree from the root to the desired node and lock each node along the way in an intention mode.

This approach significantly enhances the efficiency of **locking large data items** by reducing the time needed to **perform lock requests**. Additionally, it allows for finer control over the concurrency of the system, which can improve its performance and scalability.

Timestamp-Based Protocols

Concurrency control is an essential mechanism for ensuring correctness and consistency. One approach to concurrency control is the use of **timestamp-based protocols**, which rely on assigning unique timestamps to each transaction in the system. These timestamps then determine the serializability order, which ensures that conflicting read and write operations are executed in the correct order.

```
T25: read(B);  
      read(A);  
      display(A + B).
```

To implement this scheme, each data item in the system is associated with two timestamp values that are updated whenever a new read or write instruction is executed. The timestamp-ordering protocol then ensures that any conflicting read and write operations are executed in timestamp order, thereby guaranteeing conflict serializability.

```
T26: read(B);  
      B:=B -50;  
      write(B);  
      read(A);  
      A:= A+ 50;  
      write(A);  
      display(A + B).
```

T_{25}	T_{26}
read(B)	read(B)
read(A)	$B := B - 50$
display($A + B$)	write(B)

T_{25}	T_{26}
read(B)	read(B)
read(A)	$B := B - 50$
display($A + B$)	write(B)

T_{25}	T_{26}
read(B)	read(B)
read(A)	$B := B - 50$
display($A + B$)	write(B)

T_{25}	T_{26}
read(B)	read(B)
read(A)	$B := B - 50$
display($A + B$)	write(B)

Figure 214 - Schedule 3

While this protocol ensures freedom from deadlock, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. Additionally, the protocol can generate schedules that are not recoverable, but this can be addressed by performing all write together at the end of the transaction or by using a limited form of locking.

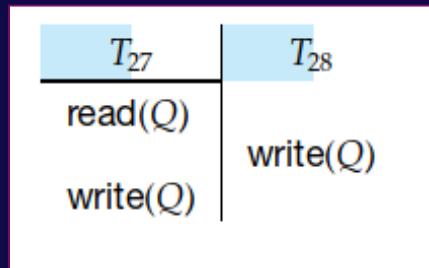


Figure 215 - Schedule 4

It is worth noting that while the **timestamp protocol** and the **two-phase locking protocol** can produce similar schedules, there are schedules that are possible under one protocol but not the other.

Overall, the **timestamp-ordering protocol** represents a powerful tool for ensuring correctness and consistency in database management systems.

Validation-Based Protocols

As transactions vie for **control over database resources**, **validation-based protocols** offer a viable alternative to the traditional concurrency control scheme. In circumstances where the majority of transactions are **read-only**, the occurrence of conflicts may be rare. As such, a **validation-based protocol**, with its minimal overhead, presents a desirable option. However, the unpredictability of conflict occurrence necessitates the need for a scheme that can monitor the system for potential conflicts.

The **validation protocol**, a three-phased process, facilitates the execution of transactions while ensuring a consistent state of the database.

- During the **read phase**, the transaction reads data items and stores them in local variables, while any write operations are made on temporary local variables.
- In the **validation phase**, the transaction undergoes a validation test to determine its suitability for proceeding to the write phase, which entails updating the actual database with data from the local variables.

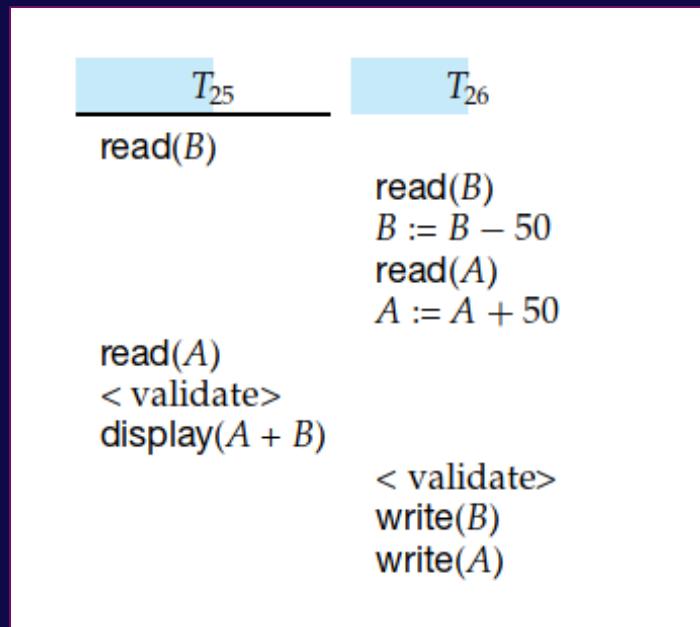


Figure 216 -Schedule 6, a schedule produced by using validation

To determine the **serializability order of transactions**, the protocol uses a timestamp-ordering technique. Each transaction is associated with three timestamps: **start**, **validation**, and **finish**.

By comparing the timestamps of concurrently executing transactions, the validation protocol ensures that any produced schedule is equivalent to a serial schedule.

In the validation test, all transactions with a lower timestamp must meet one of two conditions:

- First, the transaction must have been completed before the current transaction started.
- Alternatively, if the sets of data items written and read by the transactions do not intersect, and the earlier transaction has been completed, the current transaction may proceed.

The **validation-based protocol** offers a streamlined and efficient alternative to concurrency-control schemes, particularly in situations where **read-only transactions** are **prevalent**. By monitoring the system and employing a timestamp-based approach, the protocol ensures the consistency of the database while *minimizing overhead* and *maximizing response time*.

Multiversion Schemes

We explore the concept of **multi-version concurrency control schemes** in database systems. These schemes, unlike the traditional concurrency-control schemes, **avoid delaying or aborting transactions** by keeping old copies of data items in the system.

In a **multi-version scheme**, each write operation creates a new version of the data item, and the concurrency-control manager selects the appropriate version to be read by a transaction. The selected version must ensure serializability and should be quickly determinable for performance reasons.

The **multiversion timestamp-ordering protocol**, which extends the **traditional timestamp-ordering protocol**, ensures serializability by associating a static timestamp with each transaction and a sequence of versions with each data item. Whenever a transaction creates a new version of a data item, it initializes the **W-timestamp** and **R-timestamp fields** to its timestamp. The system updates the **R-timestamp value** of a version whenever a transaction reads its content.

Under this scheme, a **read request** never fails and is never made to wait, which is advantageous for database systems where reading is more frequent than writing. However, the scheme has two drawbacks:

- Reading a data item requires two disk accesses
- Conflicts between transactions are resolved through rollbacks rather than waits

To address these issues, the **multiversion two-phase locking protocol** combines the advantages of multi-version concurrency control with those of two-phase locking.

Update transactions hold all locks up to the end of the transaction and follow rigorous **two-phase locking**, while **read-only transactions** follow the multi-version timestamp-ordering protocol.

These **multi-version schemes** offer an efficient way to ensure serializability in database systems while avoiding the drawbacks of traditional concurrency-control schemes.

Snapshot Isolation

Snapshot isolation is a **widely accepted concurrency-control scheme** that has gained popularity in both *commercial* and *open-source database management systems*, including the likes of *Oracle*, *PostgreSQL*, and *SQL Server*. Introduced earlier in the Section, we now delve deeper into its inner workings to gain a more comprehensive understanding.

The idea behind snapshot isolation is to give a transaction a “*snapshot*” of the database at the beginning of its execution, on which it operates in isolation from concurrent transactions. The data values in this snapshot consist only of **committed transactions**, making it ideal for **read-only transactions** that never wait nor get aborted by the concurrency manager.

However, transactions that update the database must interact with potentially conflicting concurrent update transactions before they can actually place the updates in the database. These updates are kept in the **transaction's private workspace** until the transaction successfully commits, at which point the updates are written to the database.

Deciding whether or not to allow an **update transaction** to commit requires caution since two concurrent transactions might attempt to update the same data item.

- This could result in a lost update if both transactions are allowed to write to the database, where the first update written would be overwritten by the second.
- To prevent this, two variants of snapshot isolation are used, namely, first committer wins and first updater wins, both of which are based on testing the transaction against concurrent transactions.
- Under **first committer wins**, when a transaction enters the partially committed state, it undergoes a test to see if any concurrent transaction has already written an update to the database for some data item that the transaction intends to write. If some such transaction is found, then the transaction aborts;
- Otherwise, the transaction commits and writes its updates to the database.

Under **first updater wins**, the system uses a locking mechanism that applies only to updates. If a transaction attempts to update a data item, it requests a write lock on that data item. If the lock is not held by a concurrent transaction, the transaction can proceed with its execution, including possibly committing.

Insert Operations, Delete Operations, and Predicate Reads

As we delve deeper into the realm of database management, we come across a crucial aspect that plays a significant role in maintaining the integrity and consistency of the database - **concurrency control**. In particular, we examine the impact of operations such as insertions and deletions on concurrency control, which have hitherto been ignored.

As we know, database transactions not only access the existing data items but also **create** or **delete** new data items. Therefore, it becomes essential to understand the effects of such operations on concurrency control. A **logical error** in a transaction can occur when it tries to read or delete a nonexistent data item or read after the deletion or before insertion.

When a **delete instruction** is executed, we must determine when it conflicts with another instruction. If two instructions of two different transactions are deleting the same data item, then a logical error may occur. Similarly, when an insert operation is executed, it conflicts with a delete or read operation on the same data item. Therefore, for concurrency control purposes, an insert operation is treated as a write operation.

Moreover, while dealing with the **timestamp-ordering protocol**, a test similar to that for a write must be performed before a delete operation. If the delete operation passes this test, it can be executed; otherwise, it is rolled back. An insert operation assigns a value to the newly created data item and is given an exclusive lock under the two-phase locking protocol. Under the **timestamp-ordering protocol**, the **R-timestamp** and **W-timestamp** are set to the transaction's timestamp.

```
select count(*)  
      from instructor  
      where dept name = 'Physics' ;  
      insert into instructor  
      values (11111,'Feynman', 'Physics', 94000);
```

However, despite all these measures, a unique problem may still arise - the **phantom phenomenon**. When two transactions, **T30** and **T31**, conflict on a phantom tuple, concurrency control performed at the tuple granularity may go undetected. Thus, the system may fail to prevent a **non-serializable schedule**. In such a case, it becomes imperative to employ techniques like index locking to tackle the issue.

As we unravel the intricacies of database management, we realize the paramount importance of concurrency control, which forms the backbone of any robust and reliable database system.

Concurrency control is a vital concern. To ensure that transactions are executed safely and efficiently, locking protocols are used to prevent conflicts between competing transactions. But when it comes to the phenomenon known as the phantom, conventional locking mechanisms may fall short.

Phantom phenomena arise when a transaction retrieves a set of records based on a certain criterion, but another transaction subsequently modifies the records in such a way that they no longer meet the original criterion. This can lead to conflicts that are difficult to resolve using traditional locking techniques.

To address this issue, an **index-locking protocol** has been developed that takes advantage of the availability of indices on a relation. By **locking index leaf nodes**, instances of the phantom phenomenon are transformed into conflicts on locks, enabling more effective concurrency control.

Under the **index-locking protocol**, every relation must have at least **one index**. Transactions are only allowed to access tuples of a relation after first finding them through one or more of the indices on the relation. For the purpose of the protocol, a relation scan is treated as a scan through all the leaves of one of the indices.

When **performing a lookup**, a transaction must acquire a shared lock on all the **index leaf nodes** that it accesses. And if a transaction wishes to **insert**, **delete**, or **update** a tuple in a relation, it must obtain exclusive locks on all index leaf nodes that are affected by the operation.

Of course, the **index-locking protocol** has its limitations. Locking an **index leaf node** can prevent updates to the node even if the update would not actually conflict with the predicate. A variant known as **key-value locking** is presented as a means of minimizing such false lock conflicts.

Despite these challenges, the **index-locking protocol** remains a powerful tool for effective concurrency control in database systems. By using this approach, transactions can execute safely and efficiently, even in the face of phantom phenomena and other tricky concurrency issues.

Weak Levels of Consistency in Practice

In this piece, we delve into the intricacies of transactional consistency in databases, particularly in the context of weaker levels of consistency. We first highlight older terminology relating to weaker consistency levels and their relation to the levels specified in the SQL standard.

The article then proceeds to tackle the issue of concurrency control for transactions involving user interaction. We explore the concept of **degree-two consistency**, which is designed to avoid cascading aborts without necessarily ensuring serializability. This locking protocol for degree-two consistency uses shared and exclusive lock modes that function differently from those in two-phase locking.

T_{32}	T_{33}
lock-s(Q) read(Q) unlock(Q)	lock-X(Q) read(Q) write(Q) unlock(Q)
lock-s(Q) read(Q) unlock(Q)	

Figure 217 - Nonserializable schedule with degree-two consistency

Furthermore, the article explores cursor stability, which is a form of degree-two consistency used for programs that iterate over tuples of a relation using cursors. This method ensures that the tuple currently being processed by the iteration is locked in shared mode, while any modified tuples are locked in exclusive mode until the transaction commits.

However, while cursor stability can improve system performance, it is limited to specialized situations with simple consistency constraints, and its use requires that applications be coded in a way that ensures database consistency despite the possibility of non-serializable schedules.

Finally, the article delves into concurrency control across user interactions, which requires a different approach to traditional protocols. We explore various options, such as snapshot isolation, timestamp protocols, and validation, before delving into an alternative concurrency control scheme that uses version numbers stored in tuples to avoid lost updates.

Concurrency in Index Structures

Index structures are the backbone of efficient and speedy queries. However, when it comes to implementing concurrency control on these structures, lock contention becomes a major issue, leading to a decreased level of concurrency. But, lo and behold, there is a way to maintain accuracy while avoiding such lock contention, thus achieving non-serializable concurrent access to index structures.

To this end, two techniques for managing concurrent access to **B+** trees have been proposed. The first is called the "*crabbing protocol*," wherein a shared lock is first placed on the root node before traversing down the tree with shared locks on child nodes, eventually reaching a **leaf node** where an exclusive lock is placed for insertion or deletion. The protocol then works its way back up the tree, propagating any necessary splitting or coalescing operations.

However, for even greater concurrency, a modified version of **B+** trees called **B-link trees** comes into play. This technique allows for the avoidance of holding locks on one node while acquiring locks on another node, thanks to each node, including internal nodes, maintaining a pointer to its right sibling.

In this **locking protocol**, every node must be locked in shared mode before it is accessed, with a **non-leaf node's lock** being released before any other lock is requested. A split during lookup may mean that the desired search key value is no longer in the accessed node's range of values, in which case, the system searches the sibling node by following the right-sibling pointer. Leaf nodes are locked following the two-phase locking protocol to prevent the phantom phenomenon.

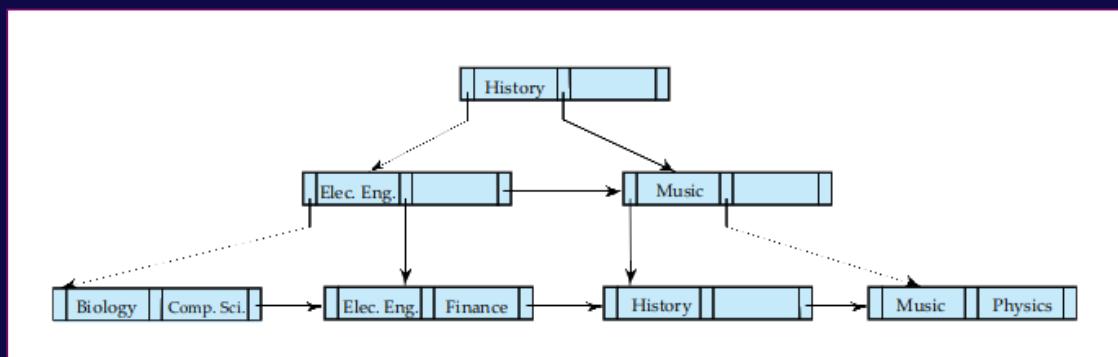


Figure 218 - B-link tree for department file with $n = 3$

For **insertion** and **deletion**, the rules for the lookup are followed to locate the leaf node where the insertion or deletion will be made. The **shared-mode lock** on this node is then upgraded to exclusive mode, and the insertion or deletion is performed. Leaf nodes affected by the operation are locked following the **two-phase locking protocol**.

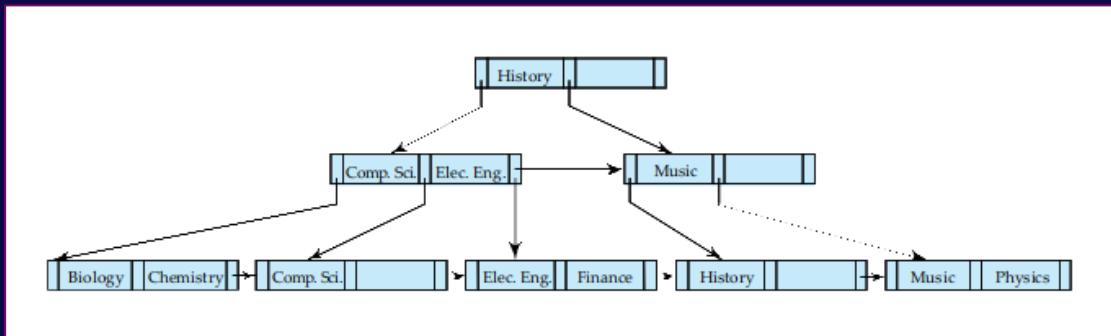


Figure 219 - Insertion of "Chemistry" into the B-link tree of Figure

Finally, splits are handled by creating a new node as the right sibling of the original node, with both **nodes' right-sibling pointers** being set. The transaction then releases the exclusive lock on the original node, requesting one on the parent to insert a pointer to the new node. Coalescing operations follow a similar process.

5.3 Recovery System

In the fast-paced world of computing, where information is the lifeblood of business and communication, a system failure can be catastrophic. A single glitch can cause a domino effect that results in the loss of vital data, leaving organizations vulnerable to enormous financial losses and reputational damage. This is why a **recovery system** is an essential component of any database system.

As with any technological device, a computer system is susceptible to a wide range of potential failures. These can be caused by a variety of factors, *from hardware issues to natural disasters*, and even malicious *acts of sabotage*. Given the severity of such threats, a database system must be designed with the ability to preserve the atomicity and durability properties of transactions.

To this end, an effective **recovery scheme** is required. Such a scheme must be capable of restoring a database to its **pre-failure state**, thereby ensuring the consistency and integrity of the system. In addition, it must provide high availability to minimize any downtime that may occur after a failure. In short, a robust and reliable recovery system is an indispensable tool in the modern era of computing.

Failure Classification

In the world of computer systems, failures are an inevitability. From *hardware malfunctions* to *software bugs*, a variety of issues can cause a system to come to a screeching halt. The impact of these failures can be devastating, leading to data loss and system downtime. To mitigate the damage caused by these failures, recovery systems must be put in place.

One of the key components of a **recovery system** is the ability to classify different types of failures. This classification allows for targeted and efficient recovery efforts. In this chapter, we focus on three main types of failures: **transaction failure**, **system crash**, and **disk failure**.

Transaction failures can be caused by either **logical** or **system errors**:

- **Logical errors** occur when a transaction is unable to continue due to internal conditions, such as bad input or resource limitations.
- **System errors**, on the other hand, occur when the system enters an undesirable state, such as a deadlock, that prevents the transaction from continuing.
- In both cases, recovery algorithms must be put in place to ensure that the atomicity and durability of transactions are preserved.

System crashes, caused by hardware malfunctions or software bugs, result in the loss of volatile storage content and bring transaction processing to a halt. However, nonvolatile storage content remains intact and is not corrupted, thanks to fail-stop assumptions and internal checks in well-designed systems.

Disk failures, such as head crashes or data-transfer operation failures, result in the loss of disk block content. To recover from these failures, copies of the data on other disks or archival backups on tertiary media, such as DVDs or tapes, must be utilized.

The recovery algorithms put in place have two parts:

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

By properly classifying failures and implementing targeted recovery systems, we can minimize downtime and preserve the integrity of our valuable data.

Storage

The storage and access of data is a paramount concern. We delved into the various media that can be **employed to store** and **access data**, highlighting differences in *terms of speed, capacity, and resilience to failure*. To ensure the stability and security of data, stable storage is a crucial component of recovery algorithms. However, stable storage is not invincible to disaster. While **RAID systems**, such as mirrored disks, provide a layer of protection against data loss due to single disk failure, they are powerless against natural disasters like fires or floods. Thus, a remote backup system that outputs blocks to a computer network at a different location can help ensure data stability in the event of a disaster.

Despite the importance of stable storage, data transfer between memory and disk storage is not without its challenges. This transfer can result in partial or total failure, so the system must **detect data-transfer failure** and invoke a **recovery procedure to restore the block** to a consistent state. To minimize the impact of such failures, the system must maintain two physical blocks for each logical database block, with the output operation completed only after both blocks have been successfully written. However, if a system fails while blocks are being written, the two copies of a block may be inconsistent with each other. In such cases, the system must examine both copies of the blocks during recovery to determine the appropriate action. To reduce the cost of recovery, a small amount of nonvolatile RAM can be employed to keep track of block writes that are in progress.

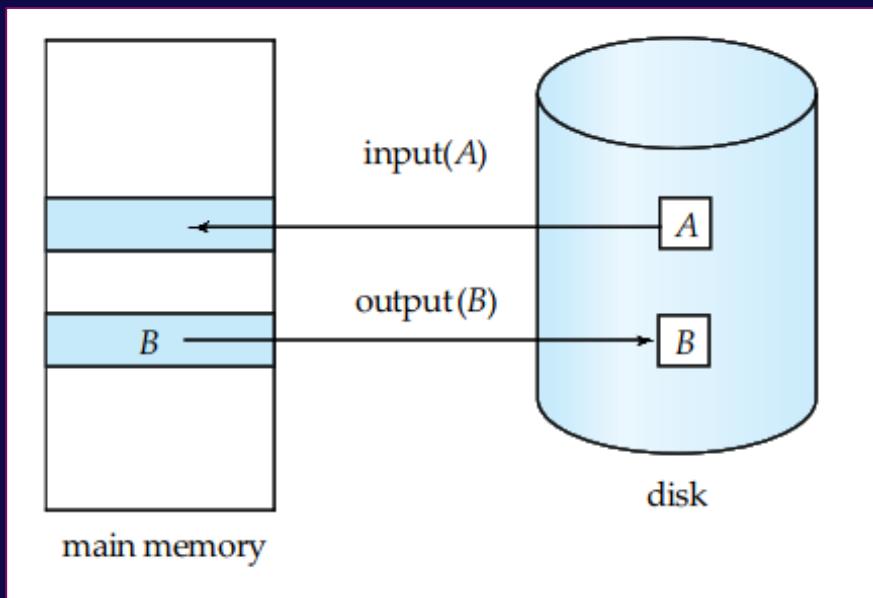


Figure 220 - Block storage operations

Data access is also a critical component of database management. The database system resides on nonvolatile storage, and the database is partitioned into fixed-length storage units called blocks. Transactions input information from the disk to the main memory, and then output the information back onto the disk in block units. In this process, physical blocks are transferred to the main memory as buffer blocks, which reside temporarily in the disk buffer. Initiating block movements between the disk and main memory is accomplished through input and output operations.

Recovery and Atomicity

In the world of database management, the need for recovery and atomicity is paramount. Consider the example of a banking system, where a transaction **T_i** transfers **\$50** from **account A** to **account B**. In the event of a system crash during **T_i's** execution, where the **output(BA)** has taken place, but the **output(BB)** has not, there is no way to know the fate of the transaction. Upon system restart, the inconsistency of the initial values of **A** and **B** violates the atomicity requirement for **T_i**.

To ensure atomicity, output operations must be performed without modifying the database itself. The most widely used method for recording database modifications is the log, which is a sequence of log records. An update log record describes a **single database write** and includes fields such as the **transaction and data-item identifiers**, the old and new values of the data item, and more. By maintaining a log of all update activities, committed transactions can be reflected in the database during recovery actions after a crash, while aborted transactions can be prevented from persisting in the database.

Although **shadow-copy schemes** can be used for small databases, copying a large database would be extremely expensive. A variant of shadow copying, called shadow-paging, reduces copying by using a page table containing pointers to all pages. However, shadow-paging is not widely used in databases due to its inability to work well with concurrent transactions.

As transactions occur in a database system, it is crucial to ensure the durability of their updates even in the event of a system crash. To accomplish this, the system maintains a log of all updates and operations performed by transactions and periodically outputs these logs to stable storage.

When a transaction commits, its commit log record is output to stable storage, signifying the end of the transaction and allowing for its updates to be redone in the event of a system crash. Conversely, if a system crash occurs before a commit log record is an output to stable storage, the transaction is rolled back.

```
T0: read(A);  
A:= A - 50;  
write(A);  
read(B);  
B:=B+ 50;  
write(B).  
C:=C- 10  
write(C)
```

```
<T0 start>  
<T0, A, 1000, 950>  
<T0, B, 2000, 2050>  
<T0 commit>  
<T1 start>  
<T1, C, 700, 600>  
<T1 commit>
```

Figure 221 - Portion of the system log corresponding to T0 and T1

Recovering from a system crash involves using the log to identify which transactions need to be redone and which need to be undone. **Redo procedures** set all data items updated by a transaction to their new values, while **undo procedures** restore these items to their old values. Redo and undo procedures are carried out by scanning the **log and performing actions** on each log record encountered.

To ensure the correctness of the recovered database, the order of updates performed during redo and undo procedures must match the original order of updates.

Log	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 1000, 950 \rangle$	$A = 950$
$\langle T_0, B, 2000, 2050 \rangle$	$B = 2050$
$\langle T_0 \text{ commit} \rangle$	
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 700, 600 \rangle$	$C = 600$
$\langle T_1 \text{ commit} \rangle$	

Figure 222 - State of system log and database corresponding to T0 and T1

In the case of a system crash, transactions are either redone or undone based on the presence of certain log records. If a **transaction's log** contains a start record but no commit or abort record, the transaction needs to be undone. Conversely, if the log contains a start record along with a commit or aborts record, the transaction needs to be redone. This procedure ensures the atomicity of transactions and allows for a speedy and efficient recovery process.

$\langle T_0 \text{ start} \rangle$ $\langle T_0, A, 1000, 950 \rangle$ $\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0 \text{ start} \rangle$ $\langle T_0, A, 1000, 950 \rangle$ $\langle T_0, B, 2000, 2050 \rangle$ $\langle T_0 \text{ commit} \rangle$ $\langle T_1 \text{ start} \rangle$ $\langle T_1, C, 700, 600 \rangle$	$\langle T_0 \text{ start} \rangle$ $\langle T_0, A, 1000, 950 \rangle$ $\langle T_0, B, 2000, 2050 \rangle$ $\langle T_0 \text{ commit} \rangle$ $\langle T_1 \text{ start} \rangle$ $\langle T_1, C, 700, 600 \rangle$ $\langle T_1 \text{ commit} \rangle$
(a)	(b)	(c)

Figure 223 - The same log, shown at three different times

The **log-based recovery techniques** used in database systems are essential for maintaining data durability and ensuring that transactions can be recovered in the event of a system crash. By using the log to identify and redo/undo transactions, the system can maintain the integrity of its data and minimize recovery time.

The process of identifying the transactions that require **redoing** or **undoing** can be **arduous** and **time-consuming**, given that it involves searching through the **entire log**. Furthermore, redoing transactions that have already made updates to the database can lead to longer recovery times, despite being harmless. To mitigate these issues, database systems employ checkpointing mechanisms.

One such approach involves outputting all modified buffer blocks to disk, preventing any updates from occurring during the checkpoint operation, and outputting all log records currently residing in the main memory. This creates a stable storage checkpoint that lists all transactions active at the time of the checkpoint.

This method enables recovery procedures to streamline their operations, as the presence of a record in the log permits a straightforward analysis of the transactions requiring undoing or redoing. Transactions that are completed before the checkpoint, such as **T_i**, require no redoing, as their modifications must have been written to the database before or as part of the checkpoint. On the other hand, transactions that began execution after the checkpoint, such as **T**, need to undergo either undo or redo operations.

By examining the part of the log that starts with the last **checkpoint log record**, we can determine which transactions require undoing or redoing. Transactions that have no or records in the log are undone, while transactions that do have either record are redone.

The set of transactions requiring analysis only includes those that began executing after the checkpoint. The remaining completed transactions require no analysis, thereby minimizing recovery time.

A fuzzy checkpointing scheme that allows transactions to perform updates during buffer block outputting is also available. Additionally, a more **flexible checkpointing** and **recovery scheme** that relaxes the update prevention requirement and only outputs some modified buffer blocks to disk is possible. These schemes are further elaborated on in later sections.

Recovery Algorithm

In a groundbreaking advancement, we delve into the heart of the recovery algorithm - an elusive yet critical piece of technology that remains shrouded in mystery. Thus far, we have identified the transactions that require rollback or recovery, yet a concrete algorithm for executing such tasks eluded us until now. In this definitive presentation, we will uncover the full extent of the recovery algorithm, utilizing log records to recover from transaction failure and a combination of the latest checkpoint and log records to recover from a system crash.

To initiate transaction rollback during normal operation, we must traverse the log in reverse, finding each log record of the form. Upon discovery, we restore the value **V1** to data item **Xj**, creating a special redo-only log record, denoting the value being restored to data item **Xj** during the rollback. These compensation log records do not require undoing information since we never undo such an undo operation.

Upon encountering, the **scan halts**, and a **log record is added to the log**. Notably, every update action performed by the transaction or on behalf of the transaction has now been recorded in the log, paving the way for the subsequent phases of recovery.

Recovery actions, when the database system is restarted after a crash, involve two phases. The redo phase replays the updates of all transactions, scanning the log forward from the last checkpoint. The list of transactions to be rolled back, the undo list, is initially set to the list **L** in the log record. As each log record of the form is encountered, we redo the operation, *i.e.*, we write the value **V2** to data item **Xj**. Whenever is found, **Ti** is added to the undo list, while removes **Ti** from the undo list. The undo list thus contains the list of all incomplete transactions, *i.e.*, they neither committed nor completed rollback before the crash.

In the **undo phase**, the system rolls back all transactions in the undo list, scanning the log backward from the end. Upon discovering a log record belonging to a transaction in the undo list, the system performs undo actions just as if the log record had been found during the rollback of a failed transaction. When is found for a transaction T_i in the undo list, a log record is written to the log, and T_i is removed from the undo list. The **undo phase** concludes once **undo** list becomes *empty*.

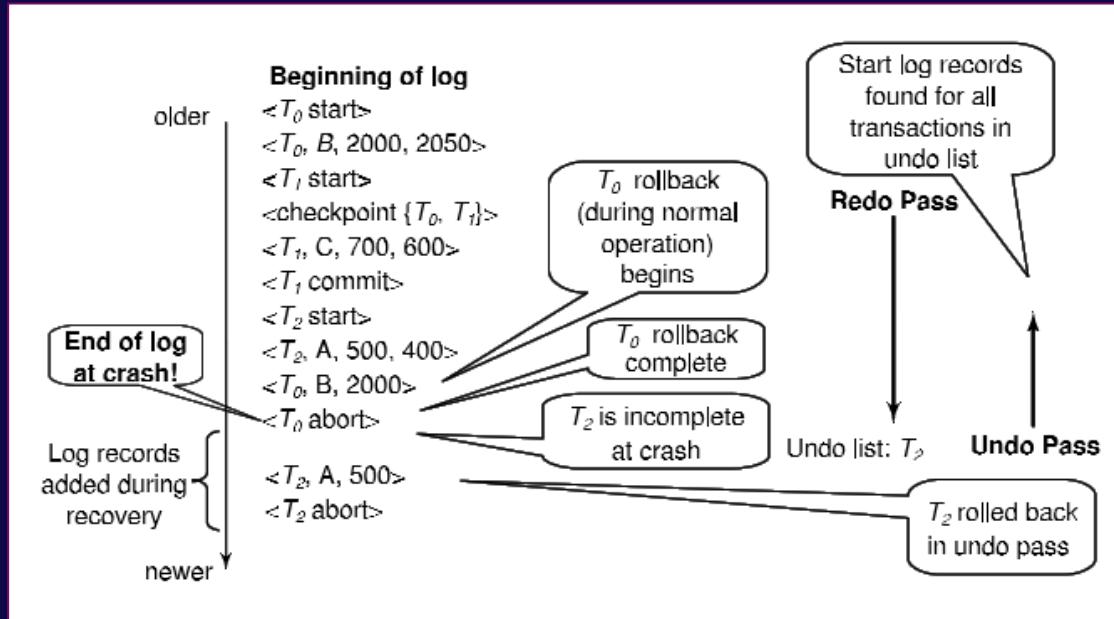


Figure 224 - Example of logged actions, and actions during recovery

Despite repeating every log record since the latest checkpoint record in the **redo phase**, this process is essential, for it repeats all the update actions executed after the checkpoint, including those of incomplete transactions and the actions performed to **roll back failed transactions**. This process called repeating history, simplifies **recovery schemes** and is deemed indispensable in executing seamless and successful recovery.

Buffer Management

In this section, we delve into the intricacies of a **crash-recovery scheme** that ensures data consistency while minimizing overhead on interactions with the database. As we navigate this complex terrain, we come across several subtle details that require our full attention.

One of the key considerations is **log-record buffering**. Traditionally, every log record is output to stable storage at the time it is created, resulting in a high overhead on system execution. To address this issue, log records can be temporarily stored in a log buffer in the main memory before being output to stable storage. This approach allows **multiple log records** to be gathered and **output to stable storage** in a single operation, thereby reducing the output of each log record to a much smaller scale.

`< T0 start > < T0, A, 1000, 950 >`

However, the downside of log buffering is that a log record may reside only in volatile storage for a considerable time before being output to stable storage. To address this issue, additional requirements need to be imposed on the recovery techniques to ensure transaction atomicity. The **write-ahead logging (WAL)** rule specifies that before a block of data in the main memory can be output to the database, all log records pertaining to data in that block must have been output to stable storage.

In addition, we need to consider the use of a **two-level storage hierarchy**, where the system stores the database in nonvolatile storage (disk) and brings blocks of data into the main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block in main memory when another block needs to be brought into memory. This requires careful consideration of the policies for writing modified blocks to disk when transactions commit.

The implementation of a **crash-recovery scheme** that ensures **data consistency** and **imposes minimal overhead** on interactions with the database requires careful attention to detail and a deep understanding of the system's underlying architecture. Only by navigating this complex terrain with precision and skill can we ensure the seamless operation of the database system, and provide users with a reliable and efficient computing experience.

Failure with Loss of Nonvolatile Storage

In a bold departure from previous discussions centered around the **failure of volatile storage**, we now turn our attention to the rare but nonetheless consequential failure of nonvolatile storage in the context of disk storage. While the loss of content in nonvolatile storage is infrequent, the gravity of such a failure necessitates a preparedness strategy.

Enter the **database dumping scheme**, which involves a periodic transfer of the entire database contents to stable storage, such as magnetic tapes, to create an archival dump. In the event of a nonvolatile storage failure that leads to physical database block loss, the system uses the most recent dump to restore the database to a previous consistent state and subsequently leverages the log to bring the database to the most **up-to-date consistent state**.

To recover from partial nonvolatile storage failure, such as the loss of a single or few blocks, only the affected blocks need restoration, with corresponding redo actions performed solely for those blocks. It is worth noting that the system requires no undo operations in this process.

As with most high-stakes solutions, the **database dumping scheme** comes with a considerable cost. The need to copy the entire database to stable storage, resulting in significant data transfer, and the halting of transaction processing during the dump procedure, leading to **wasted CPU cycles**, are two significant drawbacks. However, researchers have developed fuzzy dump schemes, akin to fuzzy-checkpointing schemes, that allow transactions to remain active during the dump procedure, minimizing the costs of the process.

Furthermore, database systems support an **SQL dump** that writes out **SQL DDL statements** and **SQL insert statements** to a file, facilitating the recreation of the database. These dumps are particularly useful during data migration to a different instance or version of the database software.

Early Lock Release and Logical Undo Operations

In this section, we delve into the intricacies of early lock release and logical undo operations in database systems. By treating an index, such as a **B+-tree**, as normal data, we can boost concurrency using the **B+-tree concurrency-control algorithm**. However, early lock release can lead to the updating of a **B+-tree** node by multiple transactions, making undo operations challenging.

To address this issue, we introduce the **concept of logical operations**, which are operations that release locks early and require logical undo operations. This approach not only applies to indices but also to other frequently accessed system data structures, such as blocks containing records of a relation, free space in a block, and free blocks in a database.

To prevent concurrent transactions from executing conflicting actions, **operations acquire lower-level locks** while executing and release them upon completion. Transactions, however, must retain a higher-level lock in a two-phase manner. When a lower-level lock is released, the operation cannot be undone using old values of updated data items and must be undone through a compensating operation, also known as a **logical undo operation**.

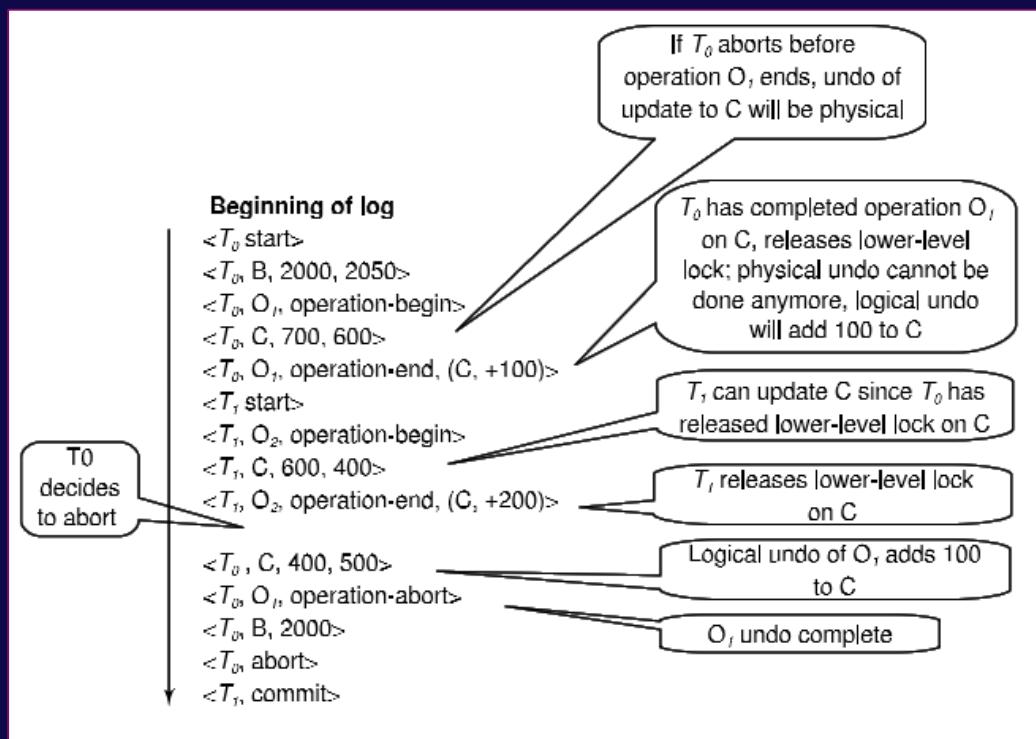


Figure 225 - Transaction rollback with logical undo operations

To facilitate logical undo operations, a transaction creates a **log record before an operation** to modify an index is performed. The system generates update log records for each update performed by the operation, and upon completion, writes an operation-end log record containing **undo information**. It is crucial to acquire sufficient lower-level locks during an operation to enable a subsequent logical undo operation.

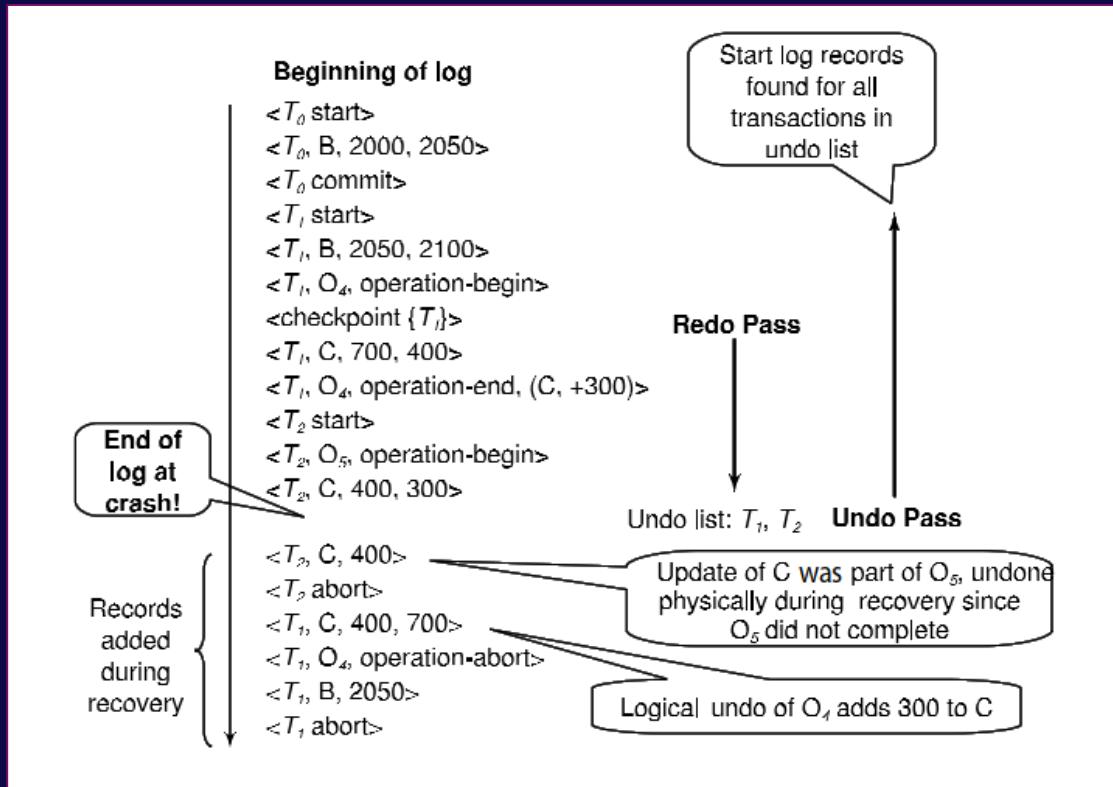


Figure 226 - Failure recovery actions with logical undo operations

Notably, **logical logging** is used exclusively for undo, while physical logging is used for redo operations. The state of the database after a system failure may reflect some updates of an operation and not others, rendering logical redo or undo operations impossible. To perform logical redo or undo, the database state on **disk must be operation consistent**.

While the intricacies of **early lock release** and **logical undo operations** may seem daunting, they are critical to database systems' performance and integrity. By leveraging the concepts introduced in this section, database systems can achieve greater concurrency and reliability.

ARIES

The **ARIES recovery method** is a remarkable feat of engineering that has pushed the boundaries of what was previously thought possible in the field of recovery techniques. It employs a sophisticated set of algorithms to reduce the overhead of checkpointing and speed up recovery times, at the cost of increased complexity.

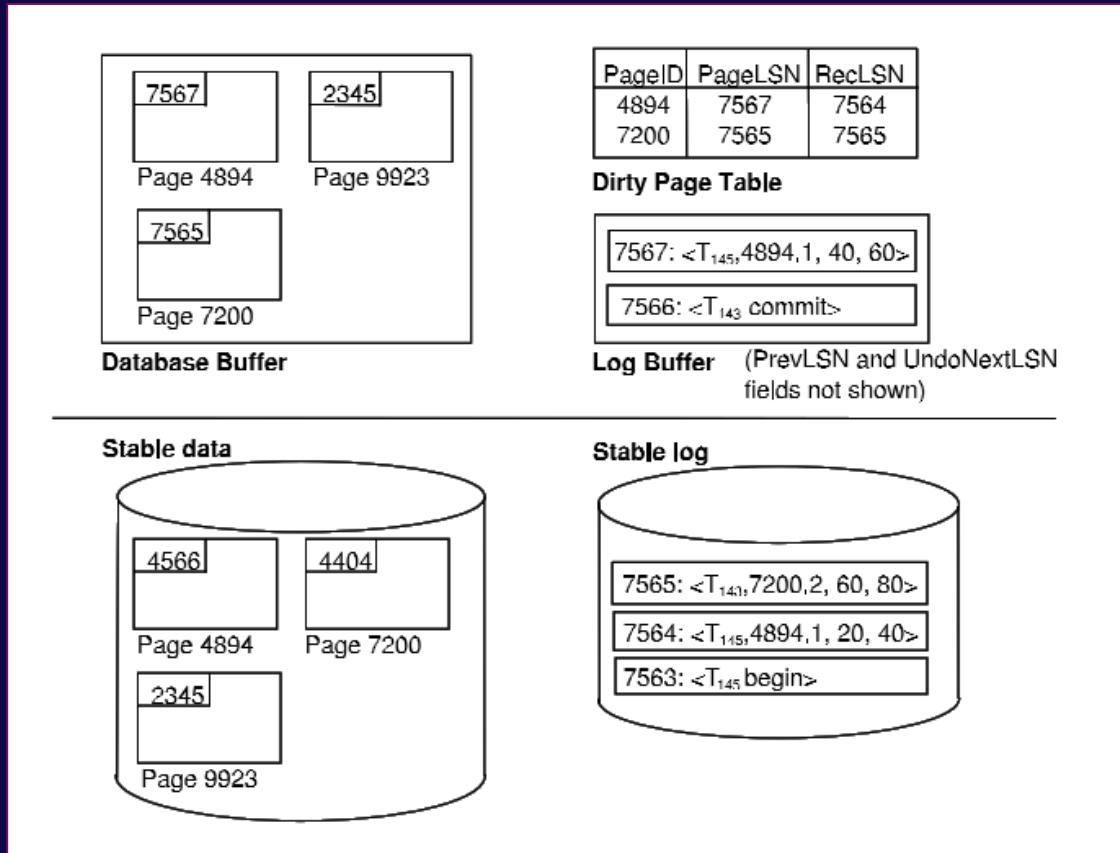


Figure 227 - Data structures used in ARIES

At the heart of ARIES lies its use of **log sequence numbers (LSNs)** to uniquely identify each log record and locate it on disk. ARIES splits the log into multiple log files, each with a file number, and generates an LSN that consists of a file number and an offset within the file. This approach enables ARIES to efficiently retrieve and apply log records during the redo phase of recovery.

Moreover, ARIES supports physiological redo operations that allow for physical changes to a page to be represented logically, resulting in smaller log records that reduce the overhead of recovery. Additionally, ARIES utilizes a dirty page table to **minimize unnecessary redos during recovery**, and a **fuzzy-checkpointing scheme** that continuously flushes dirty pages in the background, avoiding the need to write them to disk during checkpoints.

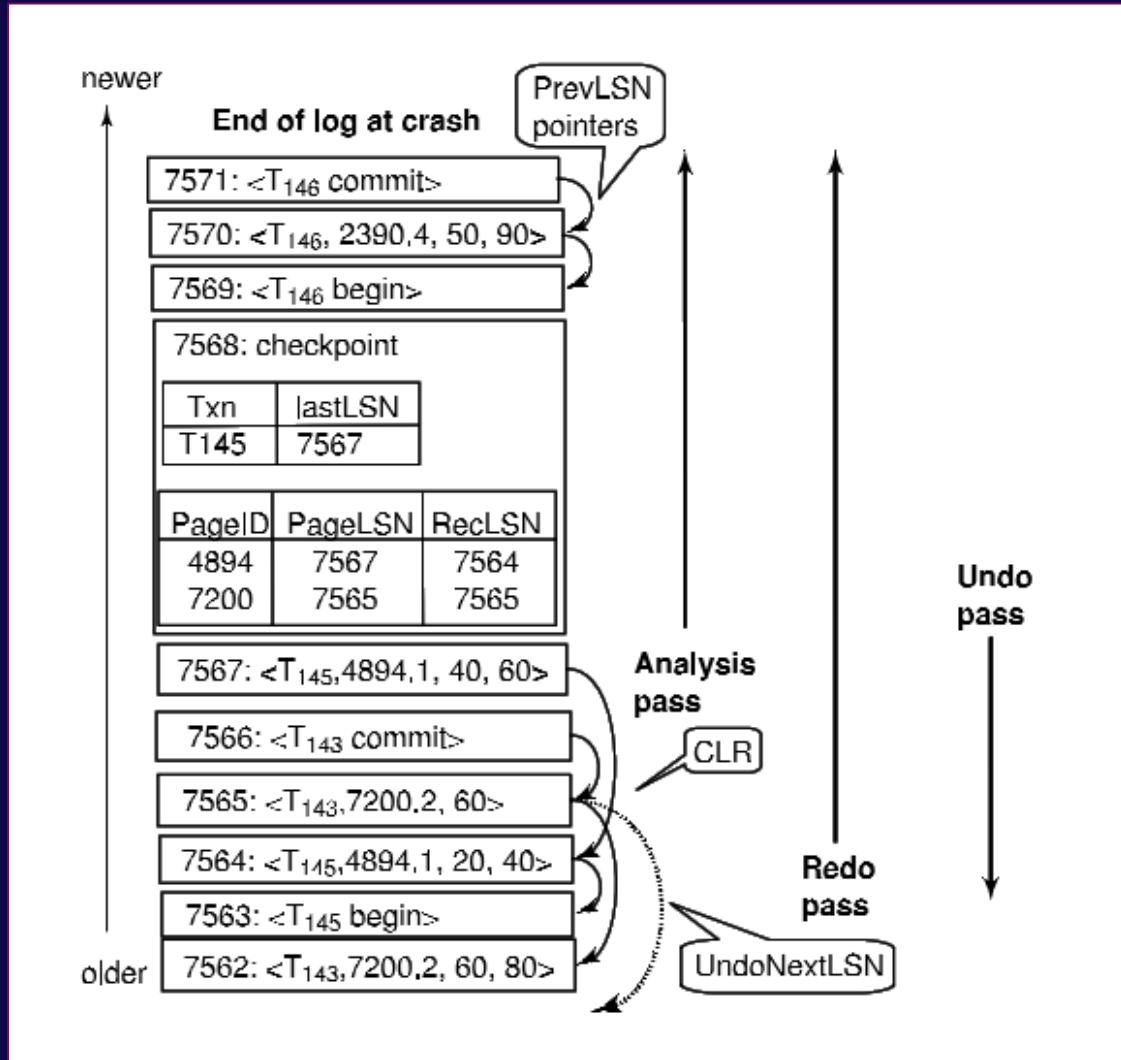


Figure 228 - Recovery actions in ARIES

The use of **PageLSNs** is another critical aspect of ARIES that helps ensure idempotence in the presence of physiological redo operations. By storing the LSN of its log record in the **PageLSN field** of a page, ARIES can avoid executing any log records with LSN less than or equal to the PageLSN of a page, as their actions are already reflected on the page.

Remote Backup Systems

In a world where environmental disasters loom large, traditional transaction-processing systems prove to be vulnerable and fragile. With fires, floods, earthquakes, and other calamities posing an imminent threat, transaction-processing systems must evolve and adapt to ensure high availability, even in the face of system failures and disasters.

The answer to this dilemma lies in remote backup systems. A primary site carries out transaction processing, while a remote backup site acts as a backup, housing all the data from the primary site. **Synchronization** is key to this arrangement, and all log records from the primary site are sent to the remote backup site to ensure updates are kept up to date.

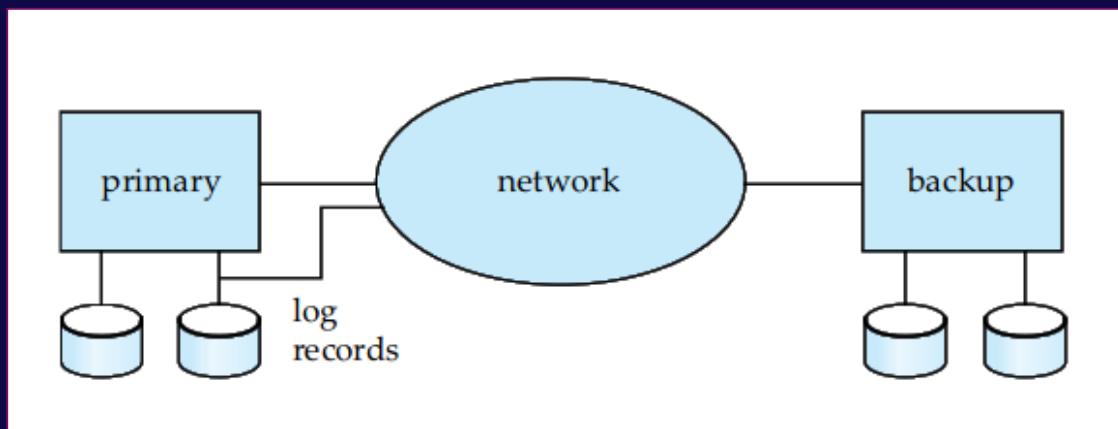


Figure 229 - Architecture of remote backup system

The **remote backup site** must be physically separated from the primary site to protect it from disasters that might befall the primary. When the primary site fails, the remote backup site takes over processing. Recovery actions that would have been performed at the primary site are executed, using its (*perhaps outdated*) copy of the data from the primary and the log records received from the primary.

However, several critical issues need to be addressed when designing a remote backup system. The system must detect when the primary site has failed, which is achieved by maintaining several communication links with **independent modes of failure**. The transfer of control from the primary to the backup site must be smooth, and if control must be transferred back, the old backup site can pretend to have failed, resulting in the old primary taking over.

In terms of durability, the remote backup site can periodically process the **redo log** records that it has received and can perform a checkpoint so that earlier parts of the log can be deleted. In a **hot-spare configuration**, the remote backup site continually processes redo log records, and as soon as the failure of the primary is detected, the backup site completes recovery and is ready to process new transactions.

The degree of durability can vary, and different schemes are available to ensure updates are durable. The **one-safe scheme** is the most basic, but it has the problem of lost updates. The **two-very-safe scheme**, while effective, can result in lower availability. The **two-safe scheme** provides better availability while avoiding the problem of lost transactions faced by the one-safe scheme.

Remote backup systems provide an effective solution to the problem of **transaction-processing systems' vulnerability** to **environmental disasters**. By addressing several critical issues and implementing different durability schemes, remote backup systems can ensure high availability and protect against data loss.

6 SYSTEM ARCHITECTURE

The architecture of a database system is influenced by the computer system it runs on. There are centralized database systems, where one server machine handles database operations and distributed databases that span multiple geographically separated machines. The different processes that enable database functionality are outlined in detail for **server** and **client-server architectures**. The chapter also describes parallel computer architectures and parallel database architectures suitable for different types of parallel computers. Furthermore, architectural issues related to building distributed database systems are discussed.

The subsequent chapter delves into how database actions, particularly query processing, can be implemented to take advantage of parallel processing. In the following chapter, various issues that arise in a distributed database system are discussed, such as storing data, ensuring transactional atomicity across multiple sites, concurrency control, and providing high availability in case of failures. The chapter also covers cloud-based data storage systems, distributed query processing, and directory systems.

6.1 Database-System Architectures

The architecture of a database system is a critical determinant of its performance and efficiency, and is heavily influenced by the underlying computer system. The aspects of computer architecture that impact database systems the most are networking, parallelism, and distribution.

The advent of networking has led to the development of **client-server database systems**, which allow tasks to be divided between server and client systems. Similarly, parallel processing within a computer system has facilitated faster response times and increased throughput, leading to the development of parallel database systems. Distributing data across multiple sites enables organizations to store data where it is generated or most needed, and helps ensure that data remains accessible even in the event of natural disasters. This has given rise to **distributed database systems**, which handle geographically or administratively distributed data spread across multiple database systems.

In this chapter, we examine the architecture of database systems, starting with traditional centralized systems and covering client-server, parallel, and distributed database systems.

Centralized and Client–Server Architectures

In the ever-evolving world of computer systems architecture, the dichotomy between **centralized** and **client-server systems** remains at the forefront of discussion.

In a centralized system, databases operate on a single computer, with no interaction with other systems. This system varies in scale, ranging from a **simple single-user** database on a personal computer to a **high-performance database on a server**. Contrastingly, a client-server system features a division of functionality between a central server and several client systems.

A modern general-purpose computer system, for instance, consists of one to a few processors and device controllers linked by a common bus. While each processor possesses local cache memories to hasten data access, device controllers are responsible for specific devices like disk drives and video displays. **Single-user systems**, such as personal computers, function independently, while multi-user systems support a large number of remotely connected users.

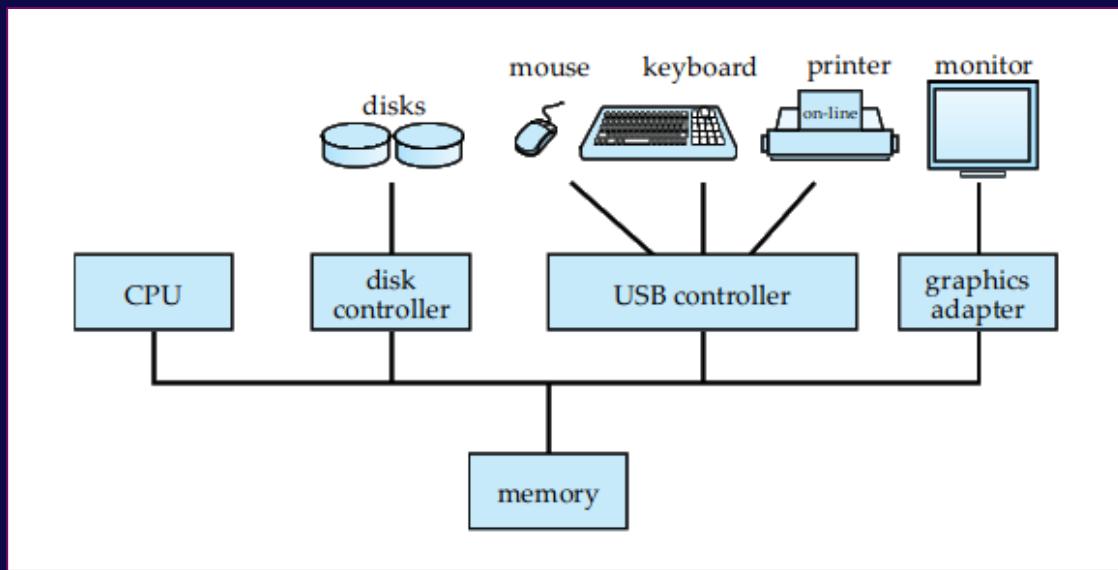


Figure 230 - A centralized computer system

While **single-user databases** do not provide many facilities for multiuser databases, such as concurrency control and advanced SQL queries, databases designed for multi-users provide full transactional features. In terms of parallelism, databases running on systems with coarse-granularity parallelism, with only a few processors, do not attempt to partition a single query. In contrast, machines with fine-granularity parallelism, with a high number of processors, aim to parallelize single tasks, such as queries, submitted by users.

As personal computers became faster, more powerful, and affordable, **centralized system architecture** became obsolete. Personal computers replaced terminals connected to centralized systems, and as a result, centralized systems evolved into server systems that respond to requests from client systems. Today, centralized systems act as server systems that cater to the demands of client systems.

The functionality of database systems can be divided into the front end and the back end. The back end manages access structures, query evaluation and optimization, concurrency control, and recovery. The front end consists of tools such as the **SQL user interface**, forms interfaces, report generation tools, and data mining and analysis tools. The interface between the front end and the back end is via SQL or an application program.

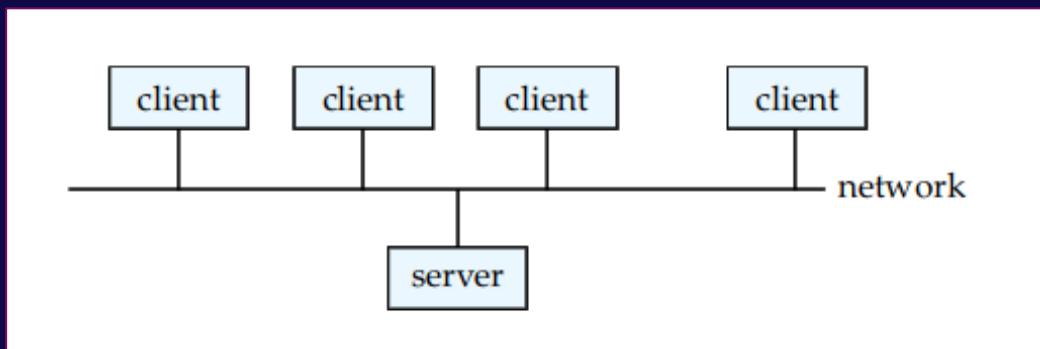


Figure 231 - General structure of a client–server system

In the realm of **client-server architecture**, standards such as **ODBC** and **JDBC** were developed to interface clients with servers. A three-tier architecture is adopted for systems dealing with a large number of users. Here, the front end is a Web browser that talks to an application server, while the application server acts as a client to the database server.

Parallelism will emerge as a critical issue in the future design of database systems as processors are expected to have more cores. While centralized systems have given way to the client-server architecture, the debate on centralized versus client-server systems is ongoing in the world of computer systems architecture.

Server System Architectures

In the realm of server system architectures, there exist two fundamental categories: **transaction servers** and **data servers**.

The former, also known as **query-server systems**, allow clients to send requests to execute a specific action, following which the results are sent back to the client. These requests are typically specified in SQL or through a specialized application program interface. The latter, data servers, enable clients to interact with servers to read or update data in files or pages. Database systems offer a range of functionalities, such as indexing facilities and transaction facilities, to manage data consistency in case of machine or process failures.

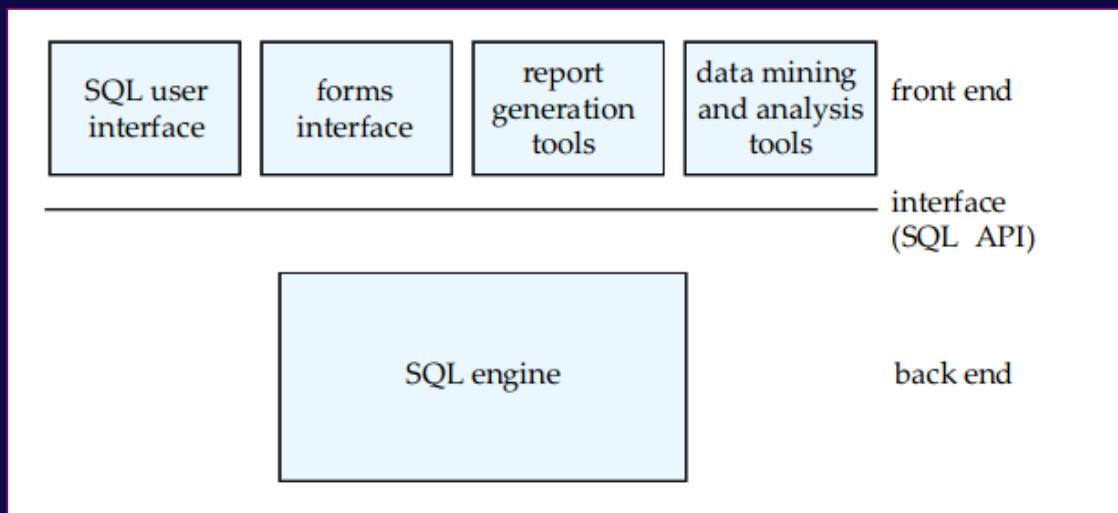


Figure 232 - Front-end and back-end functionality

Of the two, **the transaction-server architecture** reigns supreme, with *shared-memory-based* processing playing a critical role in executing complex queries with multiple simultaneous requests. At the heart of the **transaction-server system** is a complex web of processes working in harmony to ensure seamless and error-free query execution.

These include server processes, lock manager processes, database writer processes, log writer processes, checkpoint processes, and process monitor processes. The shared memory, which stores all shared data, plays a vital role in ensuring smooth data access across processes.

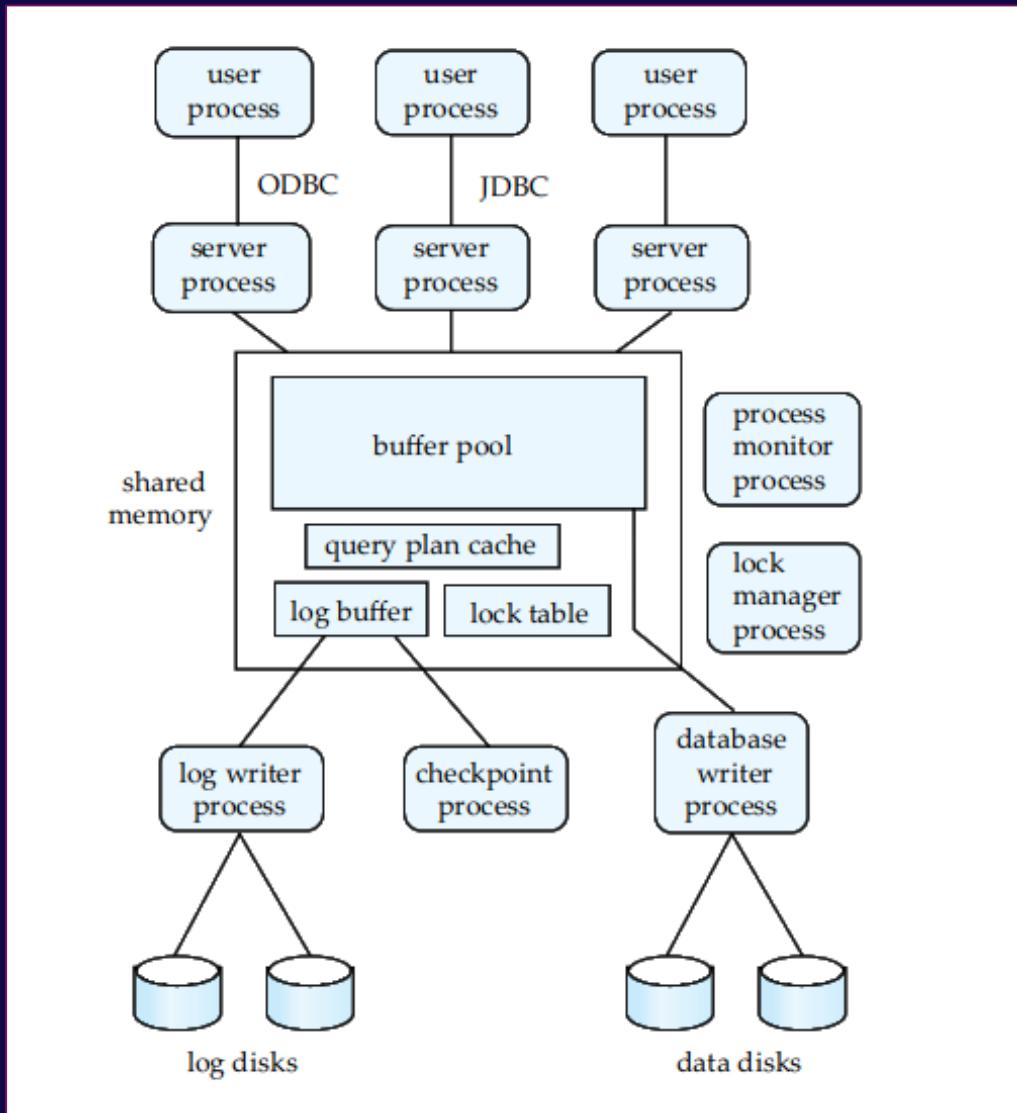


Figure 233 - Shared memory and process structure

Given the **shared-memory architecture**, it is crucial to have a mechanism in place to ensure mutual exclusion so that only one process modifies a data structure at any given time. This is achieved through the use of semaphores, special atomic instructions supported by the computer hardware. Alternative mechanisms can also be employed with less overhead.

In contrast, data-server systems are employed in **high-speed LAN environments**, where client machines possess processing power similar to the server machine. Such systems prioritize shipping data to the client machines, making it possible to perform **computation-intensive tasks** locally. Unlike transaction-server systems, data servers are not widely used and lack the shared-memory architecture that makes the former so efficient.

The intricacies of server system architectures are vast and varied, with transaction-server systems playing a dominant role in today's data-intensive landscape. Their efficient shared-memory-based processing is critical to executing complex queries with multiple simultaneous requests, making them an indispensable tool in modern-day data management.

Parallel Systems

In the age of Big Data, parallel systems are increasingly in vogue. With the ability to harness multiple processors and disks in tandem, parallel machines have emerged as a powerful tool for addressing the challenges posed by today's **data-intensive applications**. These systems are particularly useful for processing enormous databases, with capacities of the order of terabytes (*that is, 10^{12} bytes*), or for handling thousands of transactions per second. Traditional centralized and client-server database systems simply lack the muscle to keep pace with such daunting workloads.

Parallelism involves performing numerous operations simultaneously, in contrast to the sequential processing of serial systems. A typical parallel system can be either coarse-grain, with a small number of **high-performance processors**, or massively parallel or fine-grain, utilizing thousands of smaller processors. The vast majority of high-end machines today incorporate some degree of coarse-grain parallelism, with at least two or four processors. However, commercial parallel computers boasting hundreds of processors and disks are now readily available.

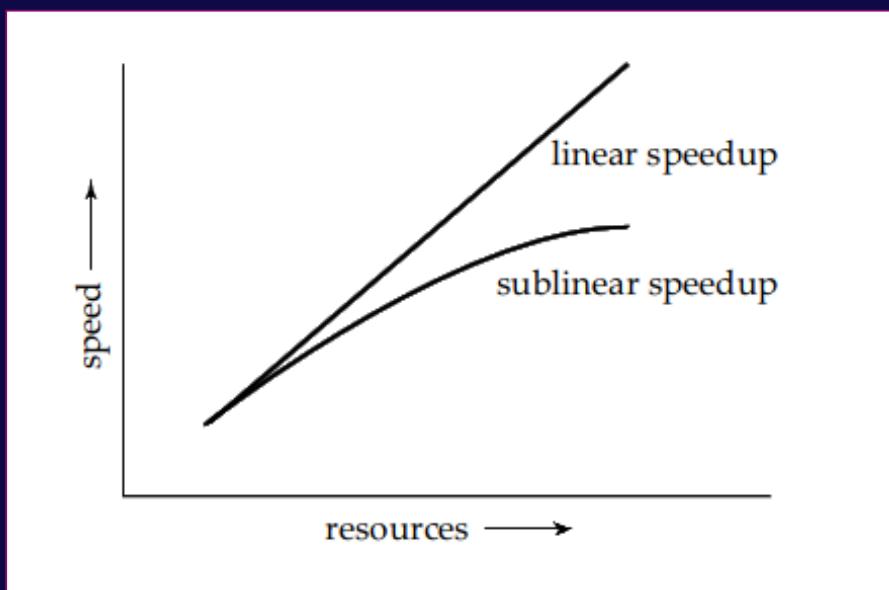


Figure 234 - Speedup with increasing resources

The performance of a database system can be measured in terms of throughput, the number of tasks completed in a given time interval, or response time, the time required to execute a single task from submission to completion.

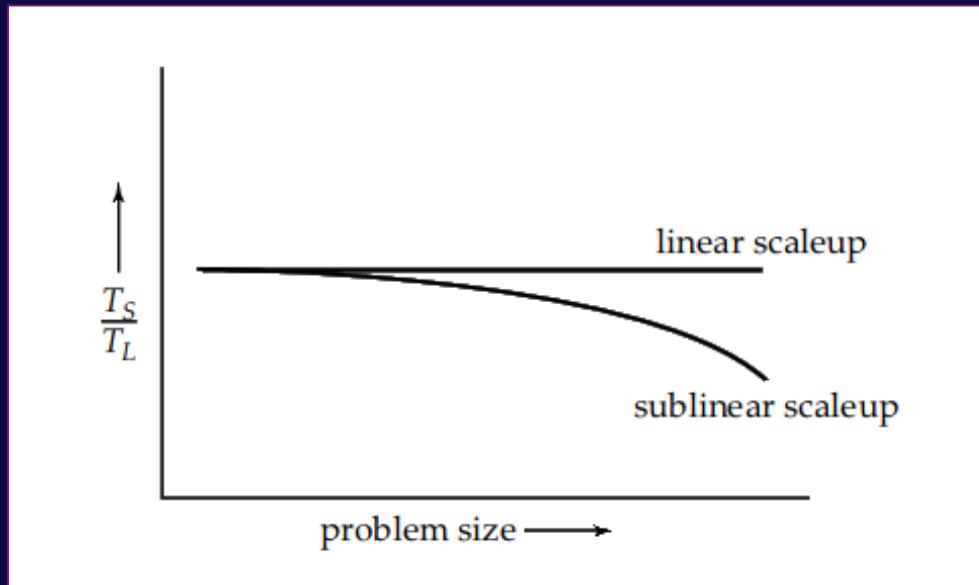


Figure 235 - Scaleup with increasing problem size and resources

The processing of small transactions can be improved by performing many in parallel, while the processing of large transactions can benefit from **sub-task parallelism**. Two fundamental measures of parallelism are speedup and scaleup. Speedup is the decrease in execution time achieved by increasing the degree of parallelism. In contrast, scaleup is the capacity to handle larger tasks by increasing the degree of parallelism.

- **Linear speedup** occurs when the speedup is **N**, reflecting the proportional increase in system resources as compared to a smaller system. Sublinear speedup occurs when the speedup is less than **N**.
- Similarly, linear scaleup is defined as $TL = TS$, where **TL** is the execution time of a task on a parallel machine **ML** that is **N** times larger than **MS**, the machine executing the same task in **TS**.
- On the other hand, sublinear scaleup is characterized by $TL > TS$.

Parallel database systems are evaluated in terms of batch and transaction scaleup, both critical for measuring the efficiency of parallel processing.

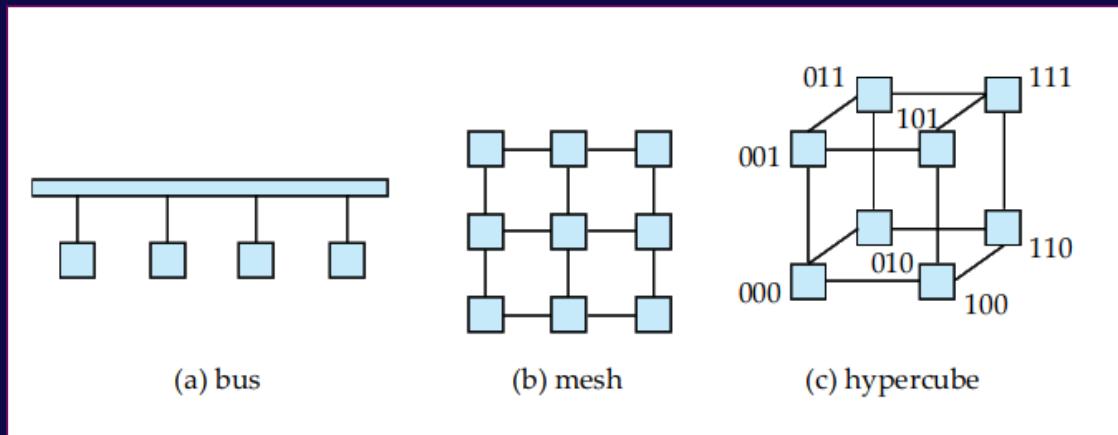


Figure 236 - Interconnection networks

For **batch scaleup**, the size of the database grows with tasks that are large jobs whose runtime depends on the size of the database. Transaction scaleup applies when the rate of database transactions increases in tandem with the size of the database. In both cases, **parallelism** enables efficient and concurrent processing across multiple processors.

Increasing parallelism is an essential tool for businesses as they manage their growth in the face of ever-increasing data volumes and transaction rates. However, several factors can hamper parallel processing, including start-up costs, overhead costs associated with synchronization, and the limited availability of parallel algorithms for certain kinds of data-intensive tasks.

Despite these challenges, parallel systems have become a critical tool in managing the data-driven requirements of modern enterprises.

Distributed Systems

In an increasingly interconnected world, the demand for **distributed database systems** has grown significantly. These systems enable the sharing of data across different sites, from workstations to mainframe systems, and have a range of benefits such as providing **autonomy to individual sites** and ensuring the availability of data even if one site fails.

Consider, for example, a banking system consisting of four branches spread across different cities. Each branch has its database of accounts, while a separate site maintains information about all the branches. A local transaction in this system involves accessing data only from the site where the transaction was initiated, while a global transaction accesses data across different sites.

However, implementing a distributed database system is no easy task. One critical issue that arises is the atomicity of transactions that run across multiple sites. If not designed carefully, these transactions may lead to inconsistencies in the system, with transactions committing at one site and aborting at another.

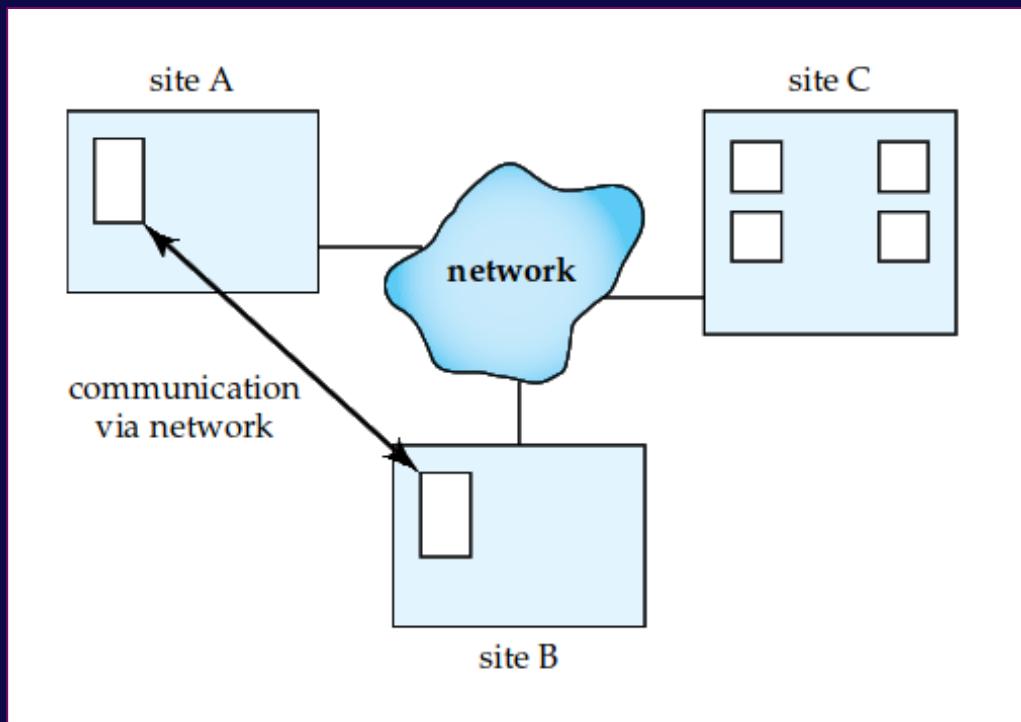


Figure 237 - A distributed system

To address this challenge, the **two-phase commit protocol (2PC)** is widely used. This protocol enables each site to execute a transaction until it enters the partially committed state, leaving the commit decision to a single coordinator site. The transaction is considered ready at a site when it reaches this state. The coordinator then decides whether to commit to the transaction based on the readiness of all sites involved.

Although recovery from failure is more complex in distributed systems than in centralized systems, the increased availability of the system outweighs this challenge. The ability of most of the system to continue operating despite the failure of one site is crucial for database systems used in real-time applications, where the loss of access to data may result in the loss of potential business to competitors.

While distributed database systems have their challenges, they play a critical role in enabling the sharing of data across different sites and ensuring the availability of data even in the face of failures. The use of protocols such as **2PC** is crucial to ensure the atomicity of transactions across multiple sites, enabling the system to operate smoothly and without inconsistencies.

Network Types

In the realm of computer networking, two types of networks reign supreme: **local-area networks (LANs)** and **wide-area networks (WANs)**. Their primary distinction is based on the geographic area they cover, with LANs confined to small areas such as a single building or cluster of adjacent buildings and WANs encompassing vast distances that can stretch across an entire country or the entire globe.

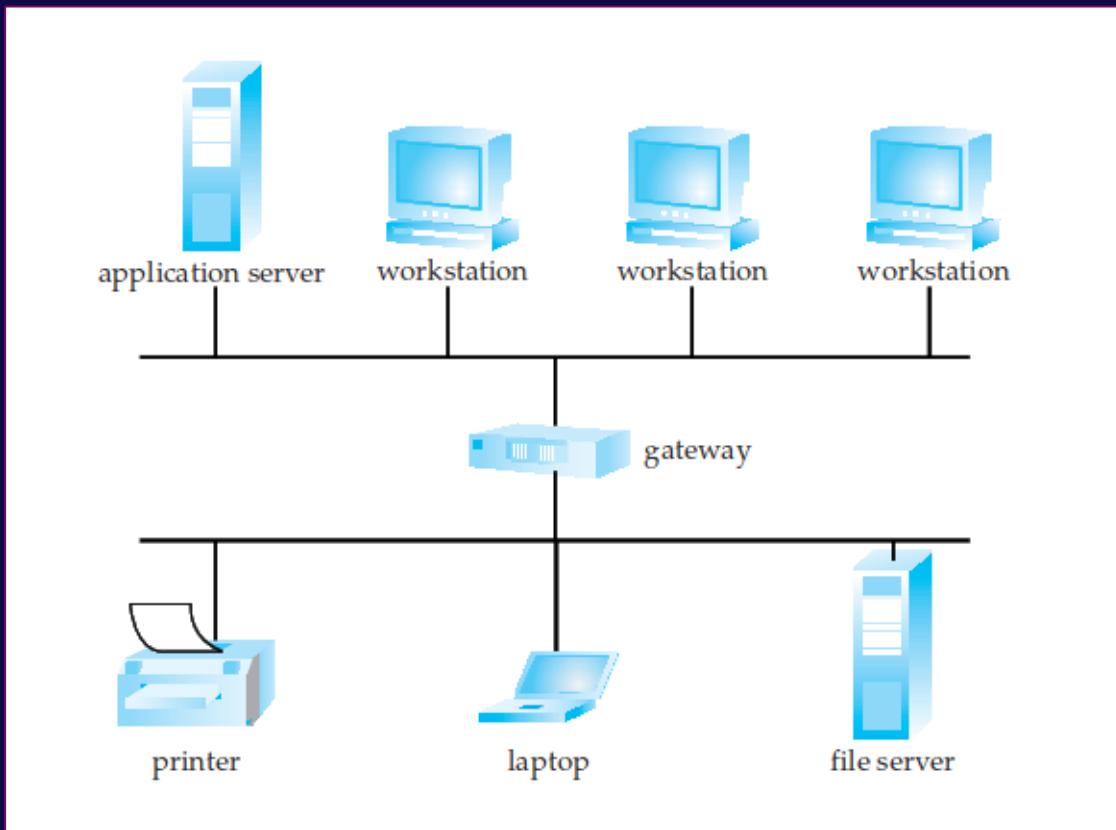


Figure 238 - Local-area network

LANs emerged as a means for computers to communicate and share data in the early 1970s. These networks are typically used within an office environment, where all sites are located within close proximity to one another. Consequently, **LANs** boast faster communication links with lower error rates compared to their WAN counterparts.

LANs utilize a variety of communication links such as twisted pair, coaxial cable, fiber optics, and wireless connections, which offer communication speeds ranging from tens of megabits per second to **1 gigabit** per second for **Gigabit Ethernet**. The more recent **10-gigabit Ethernet standard** also exists.

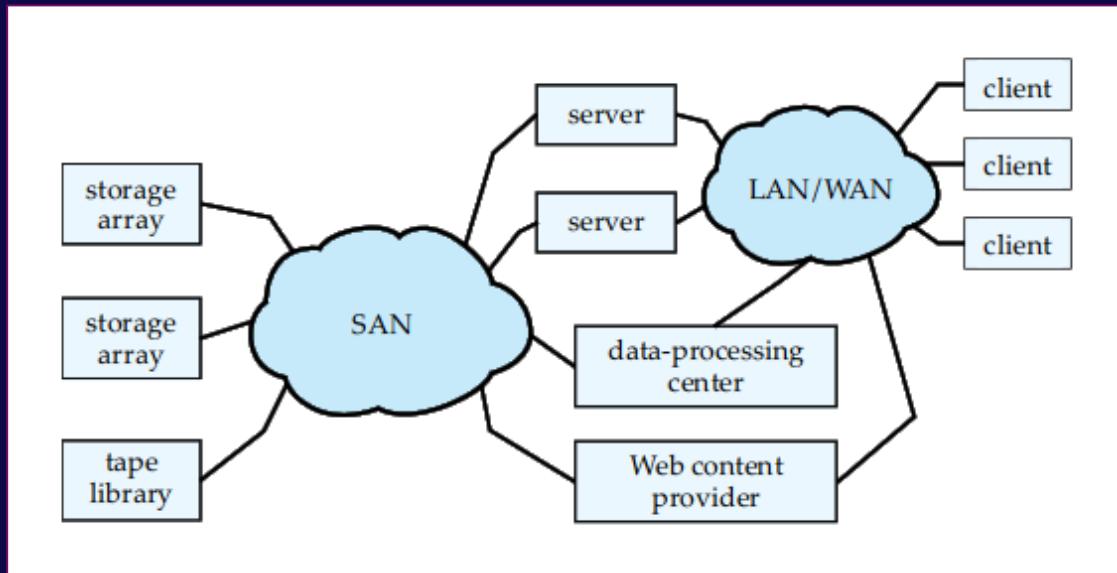


Figure 239 -Storage-area network

Furthermore, the **storage-area network (SAN)** is a specialized type of **high-speed LAN**, which serves to connect large banks of storage devices to computers that require data access. **SANs** facilitate *large-scale shared-disk systems* and offer the same benefits as *shared-disk databases*, including scalability through the addition of more computers, as well as high availability of data even if individual computers fail. **RAID organizations** are employed in storage devices to ensure the consistent availability of data even in the event of disk failures. Multiple redundancy paths between nodes are established in **SANs**, thereby ensuring that the network continues to function if a component such as a link or a connection to the network fails.

WANs, on the other hand, emerged in the late 1960s as an academic research project, mainly aimed at providing efficient communication among remote sites and allowing hardware and software to be shared conveniently and economically by a wide community of users.

The first **WAN** designed and developed was the Arpanet, which started in 1968 as a four-site experimental network and has since grown into the Internet, a worldwide network of networks with hundreds of millions of computer systems.

- **WANs** typically utilize fiber-optic lines and satellite channels, with data rates ranging from a **few megabits per second** to **hundreds of gigabits per second**.
- However, WANs contend with significant latency challenges, as messages may take a few hundred milliseconds to be delivered across the world due to the speed of light and queuing delays at several routers.
- For applications that rely on geographically distributed data and computing resources, system performance has to be carefully designed to **minimize latency's impact**.
- **Discontinuous connection WANs**, such as mobile wireless connections, and continuous connection WANs, such as the wired Internet, are the two primary classifications of WANs.
- Networks that aren't continuously connected typically do not allow transactions across sites, but they may maintain local copies of remote data and refresh them periodically.

For consistency-critical applications, groupware systems such as Lotus Notes enable remote data updates to be made locally, with the updates subsequently propagated back to the remote site at regular intervals. Nonetheless, conflicting updates may arise at different sites, necessitating detection and resolution mechanisms for such conflicts.

6.2 Parallel Databases

In this riveting chapter, we delve deep into the intricate world of parallel databases and their foundational algorithms. Specifically, we explore the relational data model and the revolutionary techniques of distributing data across multiple disks and executing relational operations in parallel. These groundbreaking approaches have paved the way for the remarkable achievements of parallel databases, and we uncover their secrets with fervent curiosity and meticulous attention to detail.

Introduction

Over two decades ago, parallel database systems were on the brink of extinction, with even their staunchest supporters doubting their viability. However, today they are being effectively marketed by practically every database-system vendor. This transformation can be attributed to several trends.

First, the **transactional requirements** of organizations have grown with increasing computer usage. Additionally, the growth of the World Wide Web has led to an abundance of sites with millions of viewers, and the increasing amounts of data collected from these viewers have produced extremely large databases at many companies.

Moreover, organizations use these vast amounts of data, such as data about customer behavior, to plan their activities and pricing. Such queries, known as **decision-support queries**, may require terabytes of data. Single-processor systems cannot handle such a large volume of data at the required rates.

The **set-oriented nature of database queries** naturally lends itself to parallelization. Several commercial and research systems have demonstrated the power and scalability of parallel query processing.

As microprocessors have become cheaper, parallel machines have become increasingly common and affordable. Additionally, individual processors have themselves become parallel machines through multicore architectures.

Parallelism is also used to provide scaleup, where increasing workloads are handled without increased response time via an increase in the degree of parallelism. The different architectures for parallel database systems: are shared memory, shared disk, shared-nothing, and hierarchical architectures.

I/O Parallelism

In the realm of database management, the technique of **I/O parallelism** has emerged as a potential game-changer. At its core, **I/O parallelism** is the practice of reducing the time needed to retrieve data from a disk by distributing the data across multiple disks, in what is known as data **partitioning**. Horizontal partitioning is the most common method, whereby the tuples of a relation are dispersed across several disks, each tuple residing on its disk.

Partitioning data over multiple disks requires a partitioning strategy, and there are three basic methods of data partitioning. The first is the **round-robin scheme**, which sends tuples to disks in any order, ensuring that each disk has a roughly equivalent number of tuples. Secondly, hash partitioning divides tuples among disks based on a hash function and the value of one or more partitioning attributes. Finally, range partitioning allocates tuples according to contiguous attribute-value ranges, resulting in each disk holding a unique range of tuples.

Once data has been partitioned, it **can be read from or written to multiple disks simultaneously**, dramatically enhancing transfer rates. However, different partitioning techniques provide varying levels of efficiency when it comes to accessing data. For instance, the round-robin scheme is best suited for applications that scan the entire relation sequentially for each query. In contrast, **hash partitioning** is ideal for **point queries** based on the partitioning attribute, while range partitioning is suitable for point and range queries on the partitioning attribute.

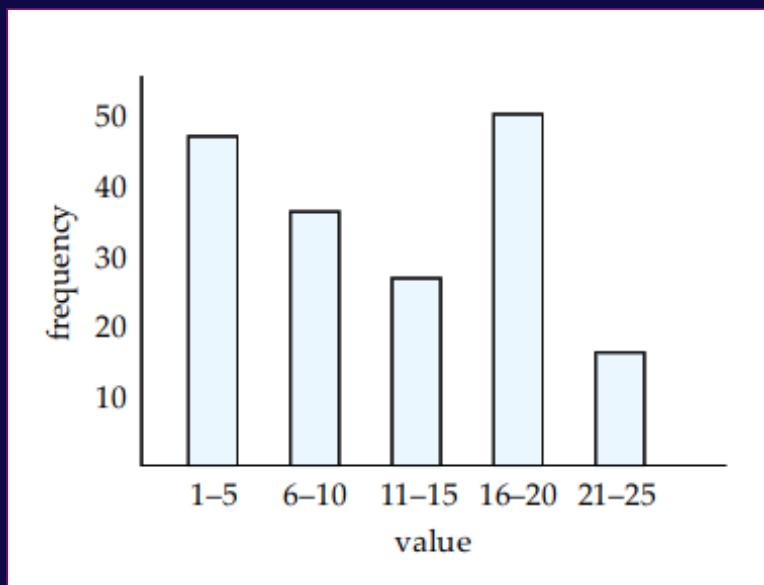


Figure 240 - Example of histogram

Range partitioning offers the added advantage of narrowing the search to only the disks that may have the data of interest, which can result in higher throughput and better response times. Despite these benefits, partitioning data is not without its limitations, as different partitioning techniques may not be optimal for all types of queries. Nevertheless, the promise of faster data retrieval times and higher throughput makes **I/O parallelism** an enticing prospect for database administrators looking to optimize the performance of their systems.

Interquery Parallelism

In the realm of database management systems, interquery parallelism is a crucial concept that warrants closer examination. In simple terms, this form of parallelism involves the **simultaneous execution of multiple queries or transactions**. The benefits of interquery parallelism are numerous; it can dramatically increase transaction throughput, allowing a larger number of transactions to be processed per second. However, it is worth noting that the response times of individual transactions remain unaffected by this type of parallelism.

Interquery parallelism is the most straightforward form of parallelism to implement in a database system, particularly in a shared-memory parallel system. This is because even sequential database systems can support concurrent processing. In contrast, shared-disk or **shared-nothing architectures** present more of a challenge, as processors must coordinate their efforts to perform certain tasks such as locking and logging. To prevent two processors from updating the same data independently at the same time, cache-coherency protocols are necessary.

To ensure that the latest version of data is available in a processor's buffer pool, a shared-disk system protocol requires a transaction to lock a page in shared or exclusive mode and read the most recent copy of the page from the shared disk immediately after obtaining a shared or exclusive lock. More sophisticated protocols eliminate the need for repeated reading and writing to disk, avoiding the overhead associated with the previous protocol. The **shared-disk protocols** can be extended to **shared-nothing architectures**, whereby requests to read or write a page are sent to the home processor of the page.

As examples, the **Oracle** and **Oracle Rdb systems** are two shared-disk parallel database systems that support interquery parallelism. Despite the complexities involved in supporting interquery parallelism, its potential benefits make it a valuable concept in database management systems.

Intraquery Parallelism

There lies a natural propensity towards the parallelization of operations. Given that relational operations work on relations with copious amounts of tuples, it is only fitting that they should be executed in parallel on various subsets of said relations. This type of parallelism within operations is what is commonly referred to as intraoperation parallelism.

We can find a thorough analysis of the parallel versions of a few of the most common relational operations. These operations **are parallel sort, parallel join, and parallel aggregate**.

When it comes to **parallel sorting**, one can envision a scenario in which a relation is located on **n disks** and **range-partitioned** on the attributes on which it is to be sorted. This allows each partition to be sorted separately and the results to be concatenated to form the entire sorted relation. Furthermore, because the tuples are partitioned across n disks, the time required to read the entire relation is notably reduced by parallel access.

In cases where the relation has been partitioned differently, there are two main approaches to sorting it. Firstly, we can **range-partition** it on the sort attributes and then sort each partition separately. Alternatively, we can utilize a parallel version of the external sort-merge algorithm.

As for parallel external sort merge, the **operation proceeds** in a sequence of actions. Initially, each processor locally sorts the data on its disk. Then, the system merges the sorted runs on each processor to get the final sorted output. This merging process can be **parallelized by range-partitioning** the sorted partitions at each processor across the various processors and sending the tuples in sorted order so that each processor receives the tuples in sorted streams. Each processor then performs a merge on the streams as they are received to obtain a single sorted run. Finally, the system concatenates the sorted runs from the processors to produce the ultimate result.

When it comes to parallel join algorithms, the idea is to split the pairs to be tested over several processors, allowing each processor to compute part of the join locally.

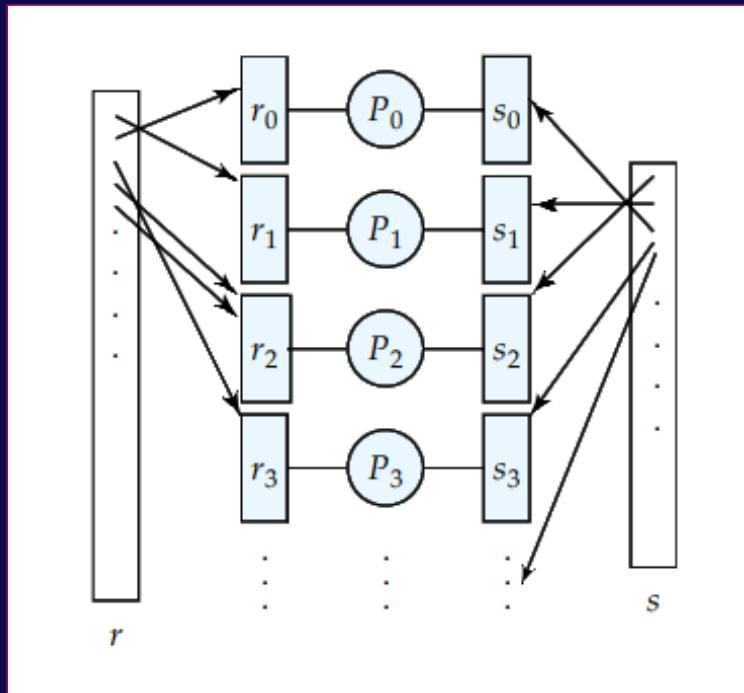


Figure 241 - Partitioned parallel join

The system then collects the results from each processor to produce the final output. For certain types of joins, such as **equi-joins** and **natural joins**, it is possible to partition the two input relations across the processors and compute the join locally at each processor. This is referred to as partitioned join.

All in all, the utilization of intraoperation parallelism in database systems provides a promising means of increasing efficiency and streamlining the execution of common relational operations.

There exist complex data sets that pose unique challenges for efficient data processing. Join operations, a fundamental operation in database systems, often require specialized techniques for efficient parallelization.

One such technique is the **Fragment-and-Replicate join**, which can parallelize join operations where partitioning is not applicable due to inequality join conditions. This technique involves partitioning one relation and replicating the other across all processors. The system then computes the join of each partition of the relation with the replicated relation at each processor. The general case of **Fragment-and-Replicate join** involves partitioning both relations into multiple partitions and replicating the partitions across processors in a specific configuration.

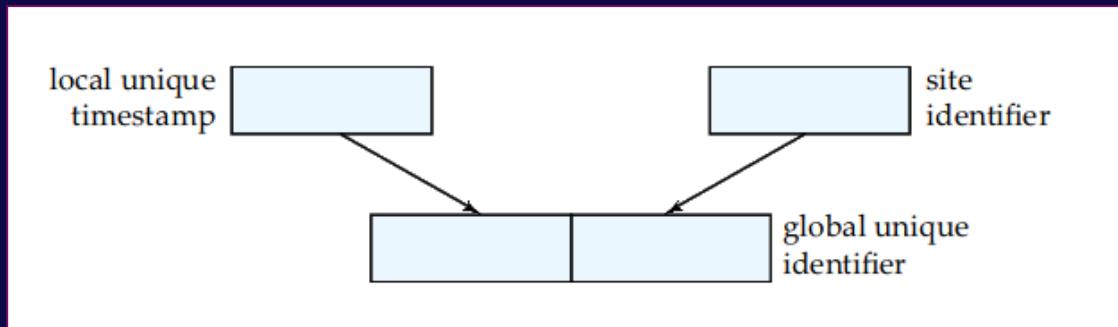


Figure 242 - Fragment-and-replicate schemes

Another technique is the **Partitioned Parallel Hash Join**, which parallelizes the Partitioned Hash Join algorithm by partitioning both relations and distributing the smaller relation across processors. The system then performs a two-phase hash join process, whereby each processor executes the hash join on local partitions of the relations to produce a partition of the final result. The hash join at each processor is independent of that at other processors, and can leverage optimizations such as caching incoming tuples in memory to **minimize I/O costs**.

These techniques offer powerful solutions for parallelizing join operations in large and complex data sets, enabling faster and more efficient data processing.

Interoperation Parallelism

There exist two forms of interoperation parallelism that can offer significant efficiency gains: **pipelined parallelism** and **independent parallelism**.

Pipelining has been a vital source of computational economy in database query processing. Its primary advantage lies in the fact that the output tuples of one operation, say **A**, can be consumed by a second operation, **B**, even before the former has produced its complete set of output tuples.

Thus, intermediate results need not be written to disk, making pipelining an appealing choice in a sequential evaluation.

Parallel systems also make use of pipelining for the same reason as sequential systems - *to minimize computational redundancy*. However, pipelines can be a source of parallelism in their own right, similar to instruction pipelines in hardware design. By running operations **A** and **B** concurrently on different processors, we can achieve pipelined parallelism.

However, it's worth noting that pipelined parallelism has its limitations, particularly in terms of scalability. Pipeline chains generally don't attain sufficient length to provide a high degree of parallelism. Furthermore, relational operators that don't produce output until all inputs have been accessed, such as the **set-difference operation**, cannot be pipelined.

Finally, pipelining offers only marginal speedup when one operator's execution cost is much higher than those of the others. Nonetheless, it can be a valuable tool when working with a lower degree of parallelism.

Query Optimization

Query optimization lies at the heart of the success of relational technology.

It aims to identify the most efficient execution plan among the plethora of options available that yields the same results as the given query. However, optimizing queries in **parallel computing environments** is a much more complicated task. The costs involved are more intricate, and factors such as partitioning costs, resource contention, and skew must be considered.

To optimize queries in **parallel systems**, one must make several critical decisions, including how to parallelize each operation, how many processors to use for each, what operations to pipeline across processors, and which operations to execute sequentially. These decisions are critical in determining the execution tree's schedule.

The optimization process also involves allocating resources, such as processors, disks, and memory, to each operation in the tree. For example, executing operations with significant computational requirements in parallel may not be the best approach if the communication overhead exceeds the computational requirements.

To optimize parallel queries, **heuristic approaches** are typically adopted to reduce the large number of execution plans that must be considered. One popular heuristic is to consider only those execution plans that parallelize every operation across all processors and do not use any pipelining. The second heuristic is to choose the most efficient sequential evaluation plan and then parallelize its operations.

Moreover, optimizing **physical-storage organization** is another aspect of query optimization. This is a complex process that involves choosing the best physical organization based on the expected mix of queries.

Design of Parallel Systems

The design of large-scale parallel database systems is a complex and multifaceted challenge that demands careful consideration of numerous interrelated issues. In this chapter, we have examined two key aspects of parallelization: **data storage** and **query processing**. However, parallel loading of data from external sources and ensuring system availability in the face of hardware failures are also crucial considerations in designing such systems.

The probability of failure of a processor or disk in a large parallel system is significantly higher than in **a single-processor system** with a single disk, making resilience to failure a critical concern. A well-designed system can continue to operate even if a component fails, with data replicated across multiple processors and requests for data automatically rerouted to backup sites. It is essential that replicas of data are partitioned across multiple processors to prevent bottlenecks.

Furthermore, when dealing with **large volumes of data**, simple operations like creating indices and schema changes can take a significant amount of time, making it unacceptable for the database system to be unavailable while such operations are in progress. Most database systems now support online operations, allowing for index construction, insertion, deletion, and updates on a relation even as an index is being built on the relation.

In recent years, several companies have developed new parallel database products, including **Netezza**, **DATAAllegro**, **Greenplum**, and **Aster Data**. Each of these products manages the partitioning of data and parallel processing of queries across the database instances. These systems leverage the data storage, query processing, and transaction management features of existing databases and focus on data partitioning, interprocessor communication, parallel query processing, and optimization. **Netezza** and **DATAAllegro** even offer data warehouse “*appliances*,” including hardware and software, making it easier for customers to build parallel databases.

6.3 Distributed Databases

In the realm of database management systems, distributed databases stand out as a unique entity, set apart from their parallel counterparts. In contrast to tightly-coupled processors that constitute a single, centralized database system, distributed databases are made up of loosely-coupled sites that do not share physical components. This **decentralized structure** can pose challenges in transaction and query processing but also offers opportunities for increased availability and fault tolerance.

In this chapter, we delve into the complexities of distributed databases, beginning with a classification of these systems as either homogeneous or heterogeneous. We explore various approaches to storing data in a distributed database, as well as models for **transaction processing**, **atomic transactions**, and **concurrency control**. In addition, we discuss how replication can be leveraged to ensure high availability in the face of failures.

Furthermore, we tackle the unique issues associated with query processing in distributed databases, as well as the complexities of handling heterogeneous databases. Finally, we examine directory systems as a specialized form of distributed databases.

Homogeneous and Heterogeneous Databases

Homogeneous and heterogeneous distributed database systems differ in their degree of similarity and cooperation among sites.

A **homogeneous system** involves sites that share the same database management software, operate with mutual awareness, and collaborate in the processing of user requests.

Such a system requires local sites to relinquish a portion of their autonomy in terms of schema modification and database management. In contrast, a heterogeneous system allows for variation in the database management software, schema, and local policies at each site. This level of flexibility allows sites to retain greater control over their local data management while participating in a distributed network.

As we explore the complexities of distributed databases in this chapter, we use the example of a banking database to illustrate the various concepts and challenges involved.

```
branch(branch.name, branch.city, assets)
account (account.number, branch.name, balance)
depositor (customer.name, account.number)
```

Figure 243 - Banking database

Distributed Data Storage

In the world of distributed databases, there are two main approaches to storing relations. The first is **replication**, where multiple identical copies of a relation are stored at different sites. The second is **fragmentation**, where a relation is divided into several fragments, each of which is stored at a different site.

Replication provides high availability and increased parallelism, but comes with a downside: **updates are more complex**, and the system must ensure consistency across all replicas. On the other hand, fragmentation allows for more efficient data transfer and simplified concurrency control but requires additional steps to ensure that the original relation can be reconstructed from its fragments.

```
account1 = Omegabranchname="Hillside"  
          (account)  
  
account2 = Omegabranchname="Valleyview"  
          (account)  
  
R = R1 U R2 U ... U Rn  
ri = PiRi (r)  
r = r1 X r2 X r3 X ... X rn
```

Horizontal **fragmentation** partitions a relation into subsets, while vertical fragmentation decomposes the schema of a relation. Regardless of the approach, the goal is always to ensure that data is stored in a way that is efficient, secure, and easily accessible. In a banking system, for example, an account can be associated with the site in which it was opened, simplifying the management of replicated data.

Ultimately, the best approach to distributed data storage will depend on the specific needs of the system in question. Replication and fragmentation can be used together to achieve optimal results. But careful consideration must be given to balancing the benefits of availability and parallelism against the complexity and overhead of updates and concurrency control. In the end, the right choice will depend on the unique requirements of each distributed database system.

Distributed Transactions

Access to data items is achieved through transactions that ensure the **ACID properties**. However, these transactions come in two types - **local** and **global** - with the latter being more complex due to multiple sites being involved in execution. Ensuring the **ACID** properties of local transactions can be achieved through well-established techniques. However, the task becomes much more daunting for global transactions, as the failure of one site or a communication link can lead to erroneous computations.

In this context, the chapter of the database management book examines the system structure of distributed databases and their possible failure modes. The chapter delves into the protocols for ensuring **atomic commit of global transactions**, concurrency control in distributed databases, and how a distributed database can continue functioning despite various types of failure.

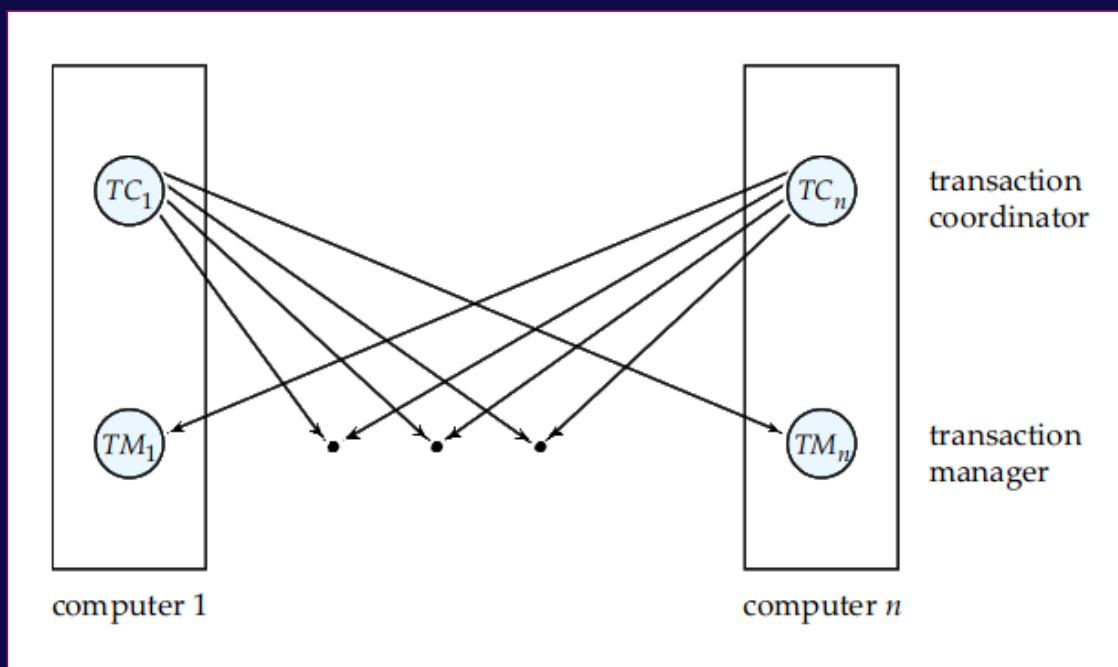


Figure 244 - System architecture

In a distributed system, each site has its own local transaction manager, which manages the execution of those transactions that access data stored locally. The transaction manager is responsible for maintaining a log for recovery purposes and participating in an appropriate **concurrency-control scheme** to coordinate the concurrent execution of transactions. The transaction coordinator subsystem, which is not needed in a centralized environment, is responsible for coordinating the execution of all transactions initiated at a site. It starts the execution of the transaction, breaks it into subtransactions, and distributes them to the appropriate sites for execution. The coordinator also coordinates the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

A **distributed system** may suffer from the same types of failures as a *centralized system*, such as *software errors*, *hardware errors*, and *disk crashes*. However, there are additional types of failure with which we need to deal in a distributed environment, such as failure of a site, loss of messages, failure of a communication link, and network partition. The loss or corruption of messages is always a possibility in a distributed system, and transmission-control protocols such as **TCP/IP** are used to handle such errors.

The chapter concludes that a system is partitioned if it has been split into two or more subsystems, called partitions, that lack any connection between them. It's noteworthy that, under this definition, a partition may consist of a single node.

Commit Protocols

Maintaining transactional atomicity is crucial for preserving data consistency. To ensure that all sites in which a transaction has been executed agree on the final outcome of its execution, a **commit protocol** must be executed by the transaction coordinator. The **two-phase commit (2PC) protocol**, among the simplest and most commonly employed, operates as follows:

- When a transaction **T** completes its execution, the coordinator **Ci** initiates *Phase 1* of the **2PC** by adding the record to the log and sending a *prepare T* message to all sites at which **T** executed. Upon receiving the message, the transaction manager at each site determines whether it is willing to commit its portion of **T**. If the answer is affirmative, it adds a record to the log and replies to **Ci** with a *ready T message*.
- If the answer is negative, it adds a record to the log and responds to **Ci** with an *abort T message*. In *Phase 2*, when **Ci** receives responses to the prepared *T* message from all sites or a predetermined time interval has elapsed, **Ci** determines whether the transaction can be committed or aborted based on the responses received.
- If **Ci** received a *ready T message* from all sites, **T** is committed; otherwise, it is aborted.

In the event of a participating site failure, the coordinator takes appropriate action depending on the stage of the commit protocol reached prior to the failure.

- After recovery, a participating site examines its log to determine the fate of incomplete transactions. The site executes **redo(T)** if the log contains a record, **undo(T)** if it contains an record, and, if it contains a record, consults with the coordinator to determine the transaction's fate.
- If the coordinator is available, the site executes either **redo(T)** or **undo(T)** based on the coordinator's decision, whereas, if the coordinator is unavailable, the site contacts other sites to obtain the transaction's fate.

While the **2PC protocol's simplicity and wide use** make it an appealing choice for database administrators, the **three-phase commit (3PC) protocol** offers certain advantages over **2PC**. However, **3PC**'s increased complexity and overhead may be a deterrent.

The problem of **two-phase commit** is a thorn in the side of many software applications. The notion of a single transaction that spans multiple sites can lead to **blocking** and **severe disruptions** in the flow of data. However, a new approach has emerged that sidesteps the issue of distributed commitment and promises to bring a more reliable and efficient solution to the table: persistent messaging.

- Imagine the scenario of transferring funds between two banks, each with its own computer. If a transaction is initiated to span the two sites and use a **two-phase commit to ensure atomicity**, the blocking of updates to the total bank balance can have serious consequences for all other transactions at each bank.
- In contrast, consider the process of transferring funds via bank check: the bank first deducts the amount of the check from the available balance and prints out a check. The check is then physically transferred to the other bank where it is deposited.
- After verifying the check, the bank increases the local balance by the amount of the check. The check constitutes a message sent between the two banks. In a networked environment, persistent messages provide the same service as the check, but much faster.

Persistent messages are guaranteed to be delivered to the recipient exactly once, neither less nor more, regardless of failures, if the transaction sending the message commits. In contrast, regular messages may be lost or may even be delivered multiple times in some situations. However, error handling is more complicated with persistent messaging than with a two-phase commit. The exception handling code provided by the application is then invoked to deal with the failure.

The implementation of persistent messaging can be achieved by utilizing a set of protocols. The sending site protocol works by writing a record containing the message in a special relation message to send, instead of directly sending out the message.

- The message is also given a unique message identifier.
- A message delivery process monitors the relation, and when a new message is found, it sends the message to its destination.
- The usual database concurrency-control mechanisms ensure that the system process reads the message only after the transaction that wrote the message commits; if the transaction aborts, the usual recovery mechanism would delete the message from the relation.

Similarly, the **receiving site protocol** works by running a transaction that adds the message to a special received messages relation provided it is not already present in the relation (*the unique message identifier allows duplicates to be detected*). After the transaction commits, or if the message was already present in the relation, the receiving site sends an acknowledgment back to the sending site.

While the types of exception conditions may arise depending on the application, it is clear that the benefits of eliminating blocking are well worth the extra effort required to implement systems that use persistent messages.

In fact, few organizations would agree to **support two-phase commit** for transactions originating outside the organization, since failures could result in the blocking of access to local data. As such, persistent messaging is quickly becoming a crucial component in the realm of distributed transactions.

Concurrency Control in Distributed Databases

In the realm of distributed databases, the importance of concurrency control schemes cannot be overstated. In a recent study, we presented novel approaches that extend the concepts of locking protocols to this distributed landscape, showcasing how they can be modified to ensure transaction atomicity and maintain global data consistency.

Our research explores the use of shared and exclusive lock modes to safeguard data integrity, and we examine several different approaches, including the **single lock-manager** and **distributed lock-manager** approaches. While both approaches offer certain advantages, they also come with their own set of challenges.

The **single lock-manager** approach boasts a simple implementation and streamlined deadlock handling, as all lock and unlock requests are centralized at a single site. However, this approach creates a bottleneck at that site, making the system vulnerable to failure if the site goes down.

On the other hand, the **distributed lock-manager** approach distributes the **lock-manager function** across several sites, reducing the likelihood of failure. But this approach comes with its challenges, including more complex deadlock handling and the possibility of intersite deadlocks.

$$Q_r + Q_w > S \text{ and } 2 * Q_w > S$$

In addition, we examine the primary copy and majority protocols as possible solutions for dealing with replicated data. These protocols allow for **efficient concurrency control**, but they also have their limitations in terms of accessibility and the need for all replicas to be available for write operations.

Overall, our study delves deep into the intricate challenges of concurrency control in distributed databases, providing valuable insights into the strengths and limitations of different approaches. These findings have the potential to pave the way for more robust and efficient systems in the future.

As we delve into the intricate world of distributed databases, we encounter a fundamental challenge that has been previously addressed in centralized databases: *how to efficiently manage timestamps in a distributed environment*. This is a crucial task, for timestamps serve as the backbone in deciding the serialization order of transactions across the distributed system.

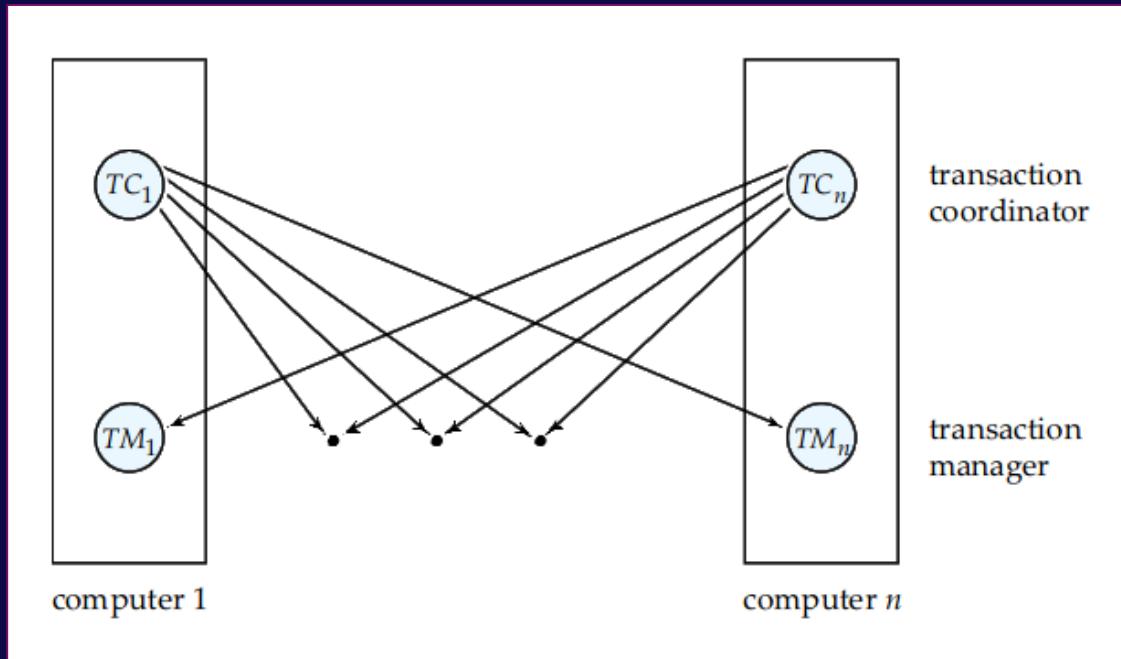


Figure 245 - Generation of unique timestamps

In the pursuit of a solution, we find two primary methods for generating unique timestamps - **centralized** and **distributed**. In the former, a single site distributes the timestamps using either a logical counter or its local clock. In the latter, each site generates a unique local timestamp using a logical counter or its local clock, which is then concatenated with the site identifier to create a unique global timestamp.

However, we must ensure that the local timestamps are generated fairly across the system, for a fast site's logical counter could inadvertently generate larger timestamps than slower sites.

This challenge is resolved by implementing logical clocks within each site that are synchronized and incremented only when a new transaction visits a site with a higher timestamp.

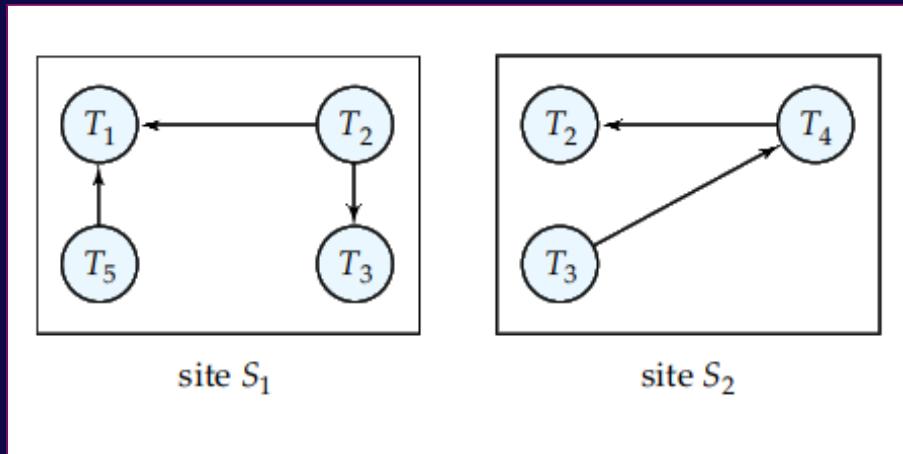


Figure 246 - Local wait-for graphs

In addition to timestamping, replication is another vital aspect of distributed databases that must be approached with caution. The **master-slave replication model** allows updates at a primary site, which are then propagated automatically to replicas at other sites. This approach is useful in creating a copy of the database to run large queries and for distributing information from a central office to branch offices of an organization. However, it does not permit transactions to update replicas at remote sites, thereby limiting its effectiveness.

In contrast, the multimaster replication model permits updates at any replica of a data item, which is then automatically propagated to all other replicas. One approach to updating replicas is to use an immediate update with a two-phase commit, utilizing one of the **distributed concurrency-control techniques**.

However, many database systems use the biased protocol, where writes have to lock and update all replicas and read lock and read any one replica as their **currency-control technique**.

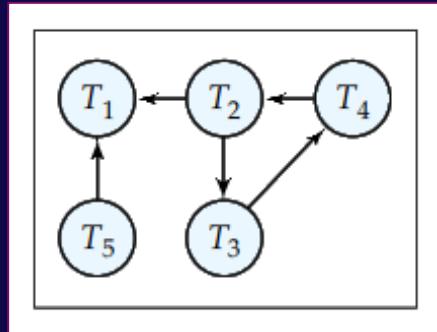


Figure 247 - Global wait-for graph

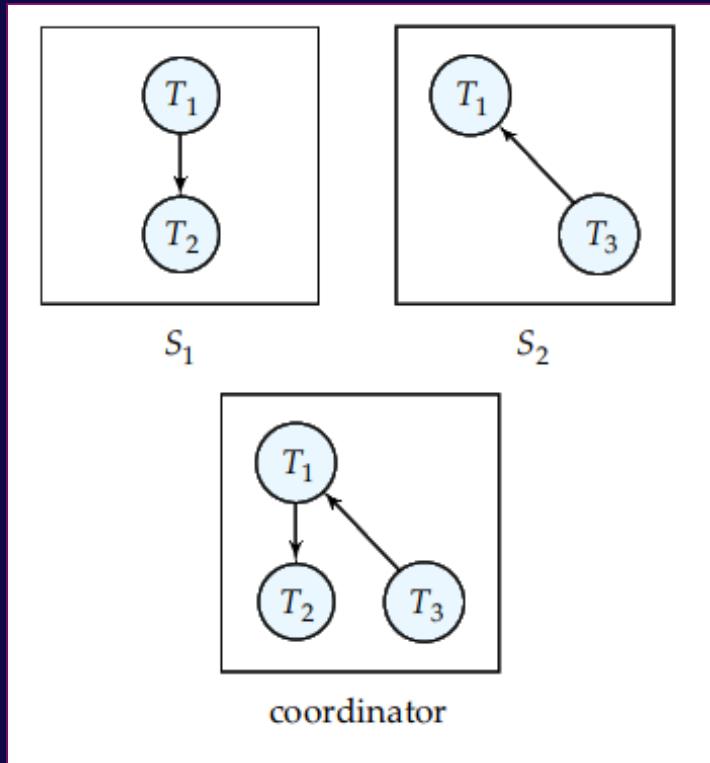


Figure 248 - False cycles in the global wait-for graph

Another alternative to immediate updates is the lazy propagation of updates to other sites, which allows transaction processing to continue even when a site is disconnected from the network. However, this comes at the cost of consistency, as updates may be performed concurrently at multiple replicas.

Therefore, certain schemes are employed to mitigate these issues, such as translating updates at replicas into updates at a primary site, which are then propagated lazily to all replicas to ensure serial ordering of updates.

In the world of distributed databases, we must strike a delicate balance between **efficiency**, **consistency**, and **availability**, for each aspect has its advantages and drawbacks. Nevertheless, with careful consideration and the implementation of appropriate techniques, we can create robust and reliable distributed databases that enable seamless communication and data management across a vast network of sites.

Availability

As a distributed database system expands in size, failures become more probable and the need for high availability becomes paramount. The ability to continue functioning during these failures referred to as robustness, is a critical goal in distributed computing. In order to achieve this robustness, the system must detect failures, reconfigure itself, and recover when a processor or a link is repaired.

Handling different types of failures requires different approaches. For example, message loss is typically addressed through retransmission, while network partition is mitigated by attempting to find an alternative route for the message. However, it can be challenging to differentiate between site failure and network partition, making it important to use multiple links between sites to ensure connectivity even if one link fails.

In the event of a failure, a distributed system must initiate a procedure to reconfigure and continue normal operation. If transactions were active at a failed site, they should be aborted promptly to avoid impeding other transactions at operational sites. If replicated data are stored at a failed site, the catalog must be updated so that queries do not reference the copy at the failed site.

It is important to design a **reconfiguration scheme** that works correctly in case of network partitioning. In particular, situations must be avoided where two or more central servers are elected in distinct partitions or where more than one partition updates a replicated data item. While traditional database systems prioritize consistency, some modern applications value availability more. Replication protocols must be designed accordingly for these systems.

The majority-based approach to distributed concurrency control is a viable solution for mitigating failures. By sending **lock-request messages** to more than half of the replicas of a replicated data object, transactions ensure that the majority of replicas are consistent. Read operations look at all replicas on which a lock has been obtained, while writes read all the replicas to find the highest version number. As long as a majority of replicas are read during the version number search and the commit contains a majority of replicas of all objects written to, the two-phase commit protocol can be used as usual on the available sites.

Robustness and high availability are essential in distributed database systems. **Mitigating failures** requires a combination of approaches, and a reconfiguration scheme that can handle network partitioning is key. By utilizing the majority-based approach to distributed concurrency control, modern applications can prioritize availability while ensuring consistency.

Ensuring both consistency and availability has long been a coveted goal, the holy grail of system design. Alas, this ideal has proven to be nothing more than a mirage, as the CAP theorem has taught us. In its eloquent and concise formulation, the theorem states that no distributed database can possess all three properties of consistency, availability, and partition tolerance.

This sobering truth is underscored by the protocols we have explored thus far, which require a (*weighted*) majority of sites to be available for updates to proceed. As a consequence, in the event of network partitions, the system may become completely unavailable for updates and even reads, depending on the read-quorum.

While the "*write-all-available*" protocol does provide availability, it does so at the cost of consistency. Thus, designers are forced to make a trade-off, deciding which of the two properties they will prioritize. Such a decision depends on the particular use case and its requirements.

Moreover, some of the algorithms discussed in this context require the use of a coordinator, whose failure can bring the entire system to a grinding halt. To circumvent this, the use of a backup coordinator is advocated, whose role is to maintain enough information locally to allow it to assume the role of coordinator with minimal disruption to the distributed system. While this backup approach does have overhead during normal processing, it allows for fast recovery from coordinator failure.

Ultimately, as distributed systems continue to proliferate and evolve, so too will the challenges that they face. But one thing remains clear: the quest for both consistency and availability will continue to captivate and motivate system designers, as they strive to strike the right balance in the ever-shifting landscape of distributed systems.

Distributed Query Processing

The intricacies of distributed query processing were explored, with a focus on finding an optimal strategy to compute a query's answer. While minimizing disk access is the primary criterion for centralized systems, several other factors must be taken into account in a distributed system, including data transmission costs over the network and the potential for improved performance through parallel processing.

One example discussed in the chapter involves the query "*Find all the tuples in the account relation.*" While seemingly straightforward, the fragmentation and replication of the account relation present challenges in choosing an optimal strategy for processing the query. Exhaustive enumeration of alternative strategies may not be practical, making the use of query optimization techniques crucial.

```
Omegabranch name = "Hillside" (account)
    account1 ∪ account2
        Omegabranch name = "Hillside" (account1 ∪ account2)
            Omegabranch name = "Hillside" (account1) ∪ Omegabranch name
            = "Hillside" (account2)
                Omegabranch name = "Hillside" (account1)
                    Omegebranch name = "Hillside" (account2)
                        Omegabranch name = "Hillside" (Omegabranch name =
                        "Valleyview" (account))
```

Join processing was also examined in the context of distributed systems. The chapter explores several potential strategies for processing the relational algebra expression "*account X depositor X branch,*" with considerations such as data volume and transmission costs playing a significant role in determining the optimal approach.

Optimizing join operations can have a significant impact on overall system performance. In this vein, the semijoin strategy presents an intriguing approach to minimizing network costs during the execution of join operations. This technique involves computing a semijoin of the two input relations, which is a subset of the original join output that can be used to filter unnecessary data before it is shipped across the network.

The **semijoin strategy** can be quite effective, particularly when a large proportion of tuples in one of the input relations do not contribute to the join result. By shipping only a reduced subset of data, the cost savings can be significant. However, as with any optimization technique, there are trade-offs to consider. For example, the cost of computing the semijoin at one site and then shipping it to another site may outweigh the benefits in some cases.

When it comes to **parallelizing join operations**, there are many possible strategies to consider. In one approach, the input relations are split across different sites, and each site computes a partial join result that is then pipelined to the final output site. This technique can significantly reduce the overall execution time for join operations, particularly for large joins involving multiple relations.

Effective join optimization is essential for achieving high performance in relational databases. While there are many different strategies to consider, the semijoin and parallelization techniques described here offer promising ways to minimize network costs and maximize system efficiency.

Heterogeneous Distributed Databases

In today's world of complex and dynamic database systems, the challenge of integrating data from diverse and **heterogeneous sources** can be a formidable task. This is where the multi-database system, a software layer that sits on top of existing database systems, comes into play. Its role is to provide a unified view of data and enable efficient query processing while allowing local databases to retain a high degree of autonomy.

However, the road to achieving a unified view of data is riddled with technical and organizational difficulties. For instance, each local database system may use a different data model, and thus a common data model must be used to create the illusion of a single, integrated database system. Furthermore, the provision of a common conceptual schema is a complicated task, as it involves the integration of separate schemas into one common schema, which can be complicated by semantic heterogeneity.

Query processing in a heterogeneous database can also be a challenging task, as queries on the global schema have to be translated into queries on local schemas at each site, and query results have to be translated back into the global schema. Wrappers can be used to simplify this process, and even provide a relational view of non-relational data sources such as Web pages, flat files, and hierarchical and network databases.

The task of integrating data from heterogeneous databases is a complicated one, but the advantages of multi-database systems far outweigh their overhead. Through a common data model and careful query processing, a unified view of data can be achieved without requiring physical database integration.

Cloud-Based Databases

Cloud computing has revolutionized the way enterprises handle their computing needs. With the advent of cloud-based databases, businesses can now access a plethora of computing services, including **data storage services**, **map services**, and **other services** that can be accessed through a web-service application programming interface. This innovative concept has gained immense popularity since it eliminates the need for a large system-support staff and allows new enterprises to commence operations without making significant, up-front capital investments in computing systems.

Many vendors offer cloud services, ranging from traditional computing vendors to industry giants such as **Amazon** and **Google**. Web applications with millions to hundreds of millions of users have driven the demand for cloud-based databases. These applications require high scalability and availability, and traditional database applications cannot meet these needs. Hence, several cloud-based data-storage systems have been developed in recent years to serve the needs of such applications.

Cloud-based databases have features of both homogeneous and heterogeneous systems. The data is owned by one organization but stored on computers that are owned and operated by another organization. This poses some transaction processing challenges, but many of the organizational and political challenges that arise in heterogeneous systems are avoided.

Despite the tremendous benefits of cloud-based databases, several technical as well as non-technical challenges need to be addressed. These include data representation, availability, security, and privacy concerns. Hence, it is imperative that cloud-based databases continue to evolve to meet the changing needs of businesses and users. The world of cloud computing is ever-evolving, and businesses must adapt to stay ahead of the curve.

As we delve into the intricate workings of cloud data storage systems, a clear picture emerges of the strategies used to manage extremely large amounts of data. The key, we discover, lies in effective partitioning and retrieval techniques, which ensure that data is divided into manageable, bite-sized chunks, or "*tablets*", for efficient processing.

Unlike traditional parallel databases, **partitioning in cloud data storage systems** is not always pre-determined. Instead, data is partitioned into small, easily-manageable units based on the **search key**, ensuring that requests for **specific key values** are directed to a single tablet. This approach avoids overburdening the system with multiple requests and allows for incremental load handling as more servers are added.

At the heart of this process lies the assignment of a master site for each tablet, where all updates are routed and **propagated to replicas**. To keep the load on each tablet in check, dynamic partitioning allows for the breaking up of large tablets into smaller parts or sharing the load across multiple sites. Meanwhile, a tablet controller site ensures efficient mapping of requests to the correct site, with replication of mapping information on router sites ensuring no single site is overburdened.

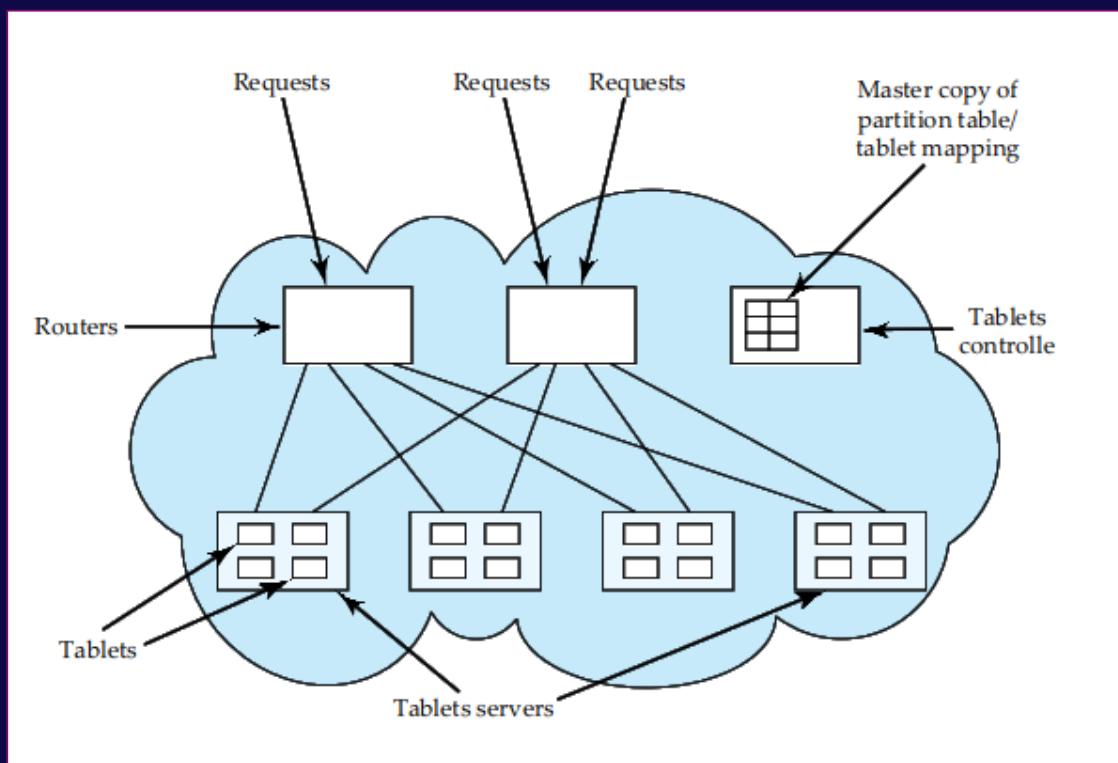


Figure 249 - Architecture of a cloud data storage system

While such systems typically do not fully support ACID transactions, data replication across multiple machines in a cluster ensures that normal operations can continue even with multiple sites down. With data replication across geographically distributed clusters, data is protected against data center failures, making cloud data storage systems both efficient and reliable.

Indeed, as we peer into the intricacies of these systems, we are left with a sense of awe at the sheer scale and complexity of the processes involved. Yet, through the judicious application of partitioning and retrieval techniques, and the deployment of efficient replication strategies, cloud data storage systems stand ready to meet the challenges of modern data management with confidence and aplomb.

Directory Systems

Amidst the evolving landscape of modern organizations and networked communication, the directory systems of yore have found a new lease on life, bringing with them an unprecedented level of accessibility and organization to employee data. Once a physical listing of employee information, the directory system has now taken on a new form in the digital age, allowing for quick and efficient searches through a comprehensive online network.

- *ldap://codex.cs.yale.edu/o=Yale University,c=USA*
- *ldap://codex.cs.yale.edu/o=Yale University,c=USA??sub?cn=Silberschatz*

These systems not only serve the needs of humans but also the programs that require access to the same information. While database systems are perfectly capable of storing the various types of employee data, directory access protocols simplify the process by catering to a specific type of access. Moreover, directory systems provide a simple yet effective way to name objects hierarchically, similar to file system directories, enabling a distributed directory system to specify the information stored in each of the directory servers.

With this design, directory servers can automatically forward queries between different locations, providing users with greater autonomy and control.

```
#include <stdio.h>
#include <ldap.h>
main() {
    LDAP *ld;
    LDAPMessage *res, *entry;
    char *dn, *attr, *attrList[] = {"telephoneNumber", NULL};
    BerElement *ptr;
    int vals, i;
    ld = ldap.open("codex.cs.yale.edu", LDAP.PORT);
    ldap.simple.bind(ld, "avi", "avi-passwd");
    ldap.search.s(ld, "o=Yale University, c=USA", LDAPSCOPE.SUBTREE,
                  "cn=Silberschatz", attrList, /*attrsonly*/ 0, &res);
    printf("found %d entries", ldap.count.entries(ld, res));
    for (entry=ldap.first.entry(ld, res); entry != NULL;
         entry = ldap.next.entry(ld, entry))
    {
        dn = ldap.get.dn(ld, entry);
        printf("dn: %s", dn);
        ldap.memfree(dn);
        for (attr = ldap.first.attribute(ld, entry, &ptr);
             attr != NULL;
             attr = ldap.next.attribute(ld, entry, ptr))
        {
            printf("%s: ", attr);
            vals = ldap.get.values(ld, entry, attr);
            for (i=0; vals[i] != NULL; i++)
                printf("%s, ", vals[i]);
            ldap.value.free(vals);
        }
    }
    ldap.msgfree(res);
    ldap.unbind(ld);
}
```

Figure 250 - Example of LDAP code in C

One widely used directory access protocol is the **Lightweight Directory Access Protocol (LDAP)**, which provides many features of the more complex *X.500 protocol* but with less complexity, making it the go-to solution for many organizations. Entries in **LDAP** directories similarly store information about objects, with each entry containing a **distinguished name (DN)** composed of a sequence of **relative distinguished names (RDNs)**. These entries may also have attributes, including telephone numbers and postal addresses, with the option of multiple values for each attribute.

All in all, directory systems have proved themselves to be invaluable resources for organizations, serving not only to provide easy access to employee data but also to authenticate users and store various types of information. By utilizing relational databases to store data, these systems have streamlined access to information, making them a fundamental element of modern organization.

7 DATA WAREHOUSING, DATA MINING, AND INFORMATION RETRIEVAL

Database queries are often used to extract specific information from a database, but when it comes to developing corporate strategies, analysts need access to large amounts of data from multiple sources. This is where a data warehouse comes in, serving as a repository of data gathered from various sources and stored under a common schema. **Complex aggregations** and **statistical analyses** are then performed on the data, often utilizing SQL constructs for data analysis. Data mining techniques may also be used to discover patterns and rules from the data, providing valuable insights for businesses.

In addition to structured data, there is also a vast amount of unstructured textual data on the internet and intranets. Information retrieval is the field of querying such unstructured data, with a focus on ranking the query results. Despite being around for several decades, information retrieval has seen tremendous growth with the development of the *World Wide Web*.

7.1 Data Warehousing and Mining

Businesses are now utilizing the vast amount of data available online to enhance their decision-making processes.

This involves two main aspects:

- Data warehousing
- Data analysis

Data warehousing involves collecting data from various sources and storing it in a central repository, while addressing challenges like dealing with dirty data and efficient storage techniques. Data analysis, on the other hand, aims to extract knowledge or information from the collected data that can aid business decisions.

Techniques like **online analytical processing (OLAP)** using SQL constructs and graphical interfaces are used for data analysis, while data mining is another approach that focuses on identifying patterns in large volumes of data.

Decision-Support Systems

In today's fast-paced and highly competitive business world, decision-making is a critical aspect of any successful enterprise. And as data continues to proliferate at an unprecedented rate, it has become increasingly clear that effectively harnessing this data is essential to making informed business decisions. Enter **decision-support systems**.

Decision-support systems, as their name suggests, provide high-level information extracted from **transaction-processing systems** to **enable decision-makers** to make informed choices. From deciding which products to stock in a retail store to identifying the target demographic for a particular product, these systems provide invaluable insights that can give organizations a competitive edge.

However, managing the vast quantities of data generated by transaction-processing systems presents a host of challenges, including the need to deal with **dirty data** and **store and index large volumes of information efficiently**. Furthermore, while SQL is a valuable tool for online analytical processing, it is not always adequate for complex statistical analyses. Therefore, tools such as **SAS** and **S++** have been developed to aid in statistical analysis and have been integrated with databases to enable large-scale data analysis.

One significant challenge faced by organizations is dealing with the diverse sources of data they use to make business decisions. This is where data warehouses come in. These centralized repositories gather data from multiple sources under a unified schema and provide a single interface to the user. The data can then be analyzed using a variety of techniques, including data mining, which utilizes statistical and artificial intelligence techniques to identify patterns in large volumes of data.

Indeed, decision support covers a broad range of areas, from statistical analysis to data mining, all of which play a crucial role in helping organizations make informed decisions. Whether a company chooses to utilize all of these tools or only a select few, it is clear that data-driven decision-making has become an essential aspect of modern business strategy.

Data Warehousing

In the world of modern business, large companies generate vast quantities of data from various sources, including local branches, operational systems, and external sources such as credit bureaus and mailing lists. This data is complex and diverse, with different schemas and structures, which can make it cumbersome for corporate decision-makers to **access the information** they require.

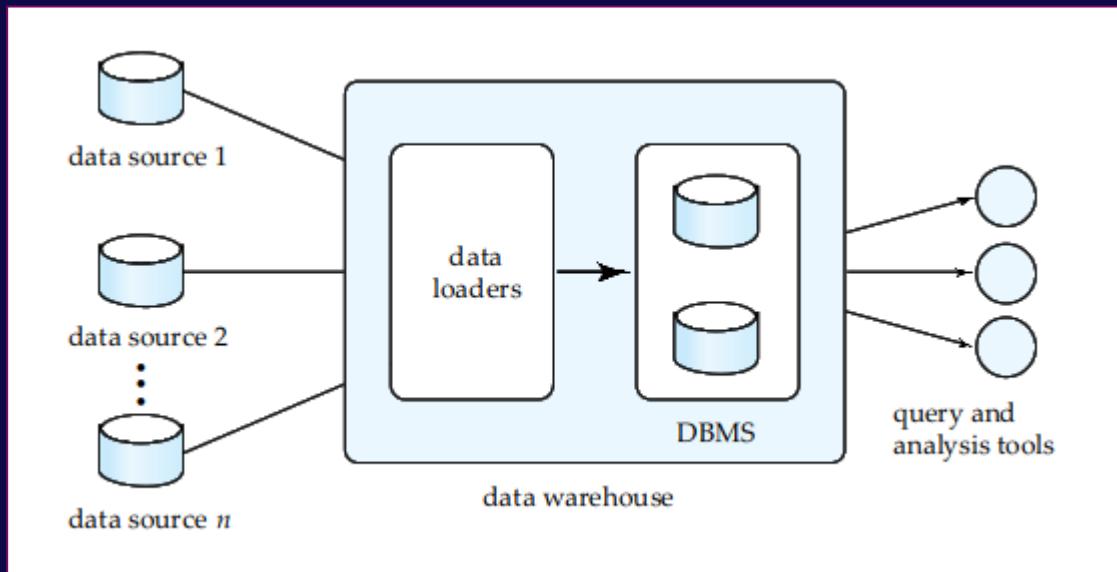


Figure 251 - Data-warehouse architecture

Enter the data warehouse. A repository of information from multiple sources, a data warehouse stores data under a unified schema at a single site, allowing **access to historical data** and **providing decision-makers** with a single consolidated interface to data. By utilizing a data warehouse, decision-makers can ensure that online transaction-processing systems are not affected by the decision-support workload.

Building a data warehouse is a complex process that involves addressing a number of critical issues. One such issue is the decision of when and **how to gather data**. Data sources may transmit new information continually or periodically, and a data warehouse may send requests for new data to sources. Schema integration and data transformation and cleansing are also critical tasks in building a data warehouse.

The process of correcting and preprocessing data is known as **data cleansing** and involves **addressing minor inconsistencies** such as misspelled names and incorrect addresses. Tools such as graphical transformation tools are used to support data transformation.

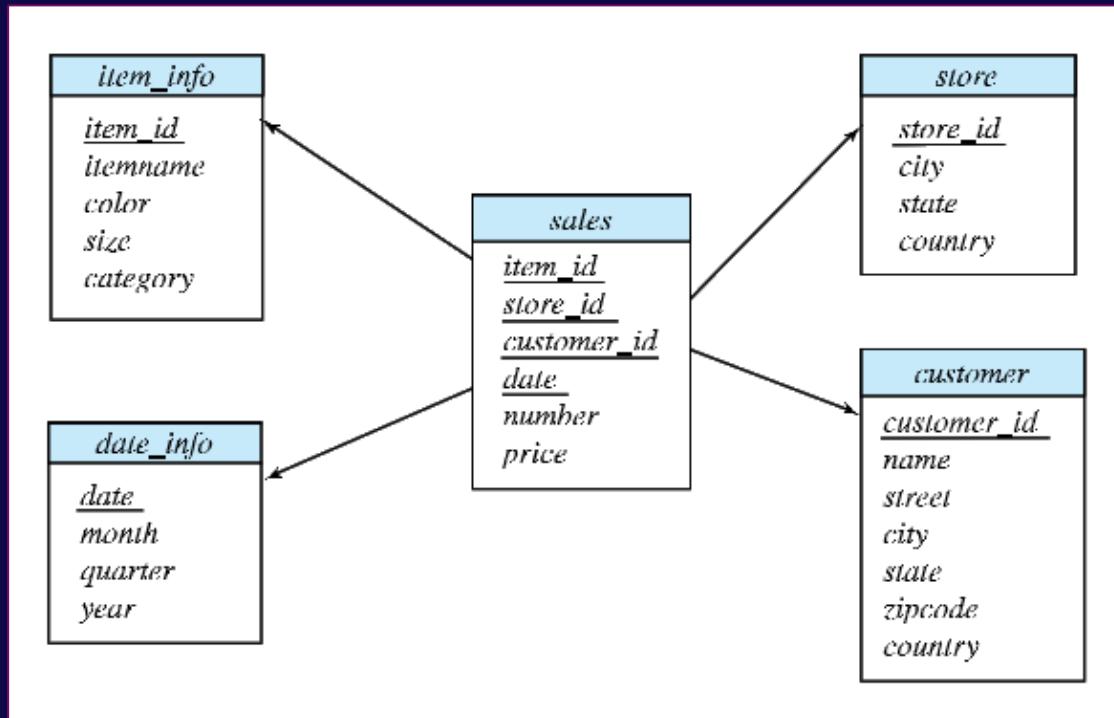


Figure 252 - Star schema for a data warehouse

Propagating updates and deciding what data to summarize are also essential elements in building a data warehouse. The raw data generated by transaction-processing systems may be too large to store online, so storing summary data obtained by aggregation on a relation is an effective solution. Finally, getting data into a data warehouse involves **extract, transform, and load (ETL)** tasks that encompass extraction of data from sources and loading data into the data warehouse.

Data warehousing is an essential component of modern business that provides decision-makers with a unified interface to complex data, ensuring online transaction-processing systems remain unaffected. The process of building a data warehouse is complex, but an end result is a powerful tool that allows for informed decision-making based on comprehensive historical data.

Data Mining

The process of data mining, a term loosely used to describe the **semiautomated analysis of large databases** in order to uncover **useful patterns**, represents a critical area of inquiry in the realm of computer science. In many ways, it is akin to knowledge discovery in **artificial intelligence** or **statistical analysis**, as it seeks to reveal underlying rules and patterns from within a given set of data. However, data mining sets itself apart from these other disciplines by focusing specifically on large datasets stored on disk - in other words, it deals with "*knowledge discovery in databases.*"

Various types of knowledge can be derived from a database, with some represented by a set of rules. For example, consider the informal rule that "*young women with annual incomes greater than \$50,000 are the most likely people to buy small sports cars.*" Of course, these rules are not always universally true and have varying degrees of "*support*" and "*confidence*." Other types of knowledge may be represented by equations, algorithms, or other predictive mechanisms.

Different techniques are used to uncover various types of patterns, such as **predictions** or **associations**. However, the process of data mining is not wholly automated, as there is often a manual component involved in **pre-processing data** and **post-processing discovered patterns** to find novel and useful insights. Additionally, there may be more than one type of pattern that can be discovered from a given database, and manual interaction may be necessary to identify useful patterns.

Despite these challenges, the knowledge that can be gained from data mining has numerous applications. Predictive modeling is a common application, where data mining can be used to determine credit risk or predict which customers may switch to a competitor. Associations can also be uncovered, such as patterns in purchasing behavior that can be exploited to increase sales. In some cases, data mining can even uncover previously unknown causal relationships, such as the link between a newly introduced medicine and cardiac problems.

Data mining represents a powerful tool for uncovering hidden knowledge within vast amounts of data. Although the process is not wholly automated and requires manual intervention, the insights that can be gained through data mining have far-reaching implications for a wide range of industries and fields.

Classification

In the realm of data mining, prediction is a lynchpin of the discipline, and no other predictive model is as widely used as classification. The art of classification is predicated on the notion of divvying up data into **mutually exclusive clusters**, such that a new data point can be assigned to one of these groups based on its attributes. For instance, imagine a credit card company seeking to determine the creditworthiness of a new applicant.

```
 $\forall$  person P,  
P.degree = masters and P.income > 75, 000  
 $\Rightarrow$  P.credit = excellent  
 $\forall$  person P, P.degree = bachelors or  
(P.income  $\geq$  25, 000 and P.income  $\leq$  75, 000)  $\Rightarrow$  P.credit = good
```

To that end, the firm surveys its existing customers and rates each of them as *excellent*, *good*, *average*, or *bad* based on their payment history. Subsequently, the company aims to discern rules that can predict how a new applicant should be classified based on certain attributes, like *income*, *age*, *educational background*, and so on, which the company has gathered. Building a classifier involves creating a training set using the available data, and using this set to create rules that will enable the classification of new instances.

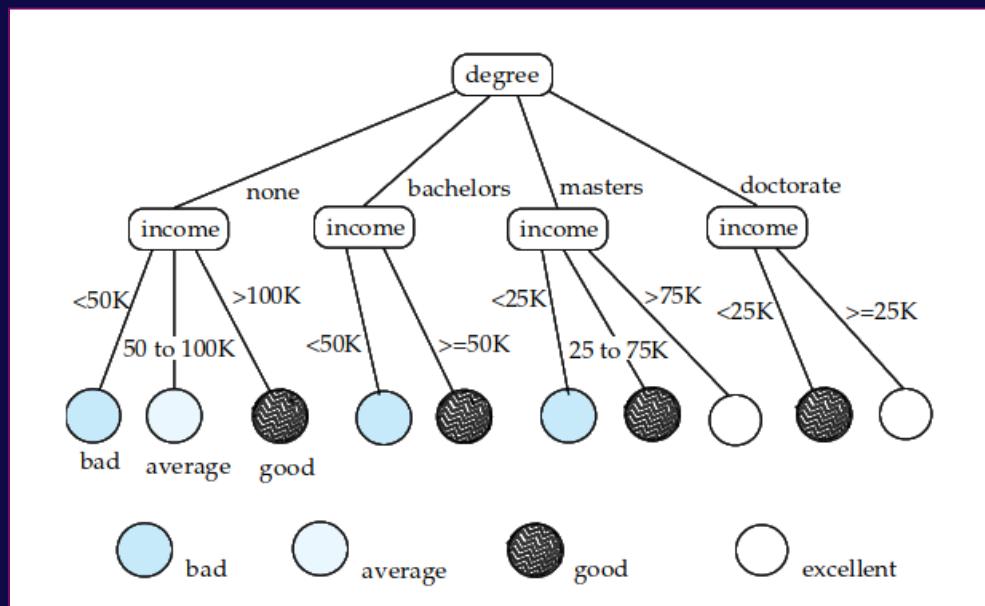


Figure 253 - Classification tree

One widely utilized type of classifier is the **decision-tree classifier**. As its name suggests, this model uses a tree structure to organize its classification criteria. The leaves of the tree correspond to distinct classes, while each internal node is characterized by a predicate or function. To classify a new data point, one begins at the root of the tree and navigates through the tree based on the attribute values of the data point. Each internal node evaluates a function or predicates to determine which child to traverse to next. The process continues until a leaf node is reached, and the class associated with that leaf node is the predicted class for the new data point.

The task of building a **decision-tree classifier** begins with a greedy algorithm that recursively constructs the tree, starting from the root node and working its way downwards.

$$\text{Gini}(S) = 1 - \sum_{i=1}^k p_i^2$$

Figure 254 - Formula

$$\text{Entropy}(S) = - \sum_{i=1}^k p_i \log_2 p_i$$

Figure 255 - Formula

$$\text{Purity}(S_1, S_2, \dots, S_r) = \sum_{i=1}^r \frac{|S_i|}{|S|} \text{purity}(S_i)$$

Figure 256 - Formula

Initially, the root node encompasses all the training instances. The algorithm then divides the data at each node based on an attribute such that each child node has a subset of the data that satisfies the partitioning condition.

```
Information gain(S, {S1, S2, ..., Sr}) = purity(S) - purity(S1, S2, ..., Sr)
```

$$\text{Information content}(S, \{S_1, S_2, \dots, S_r\}) = - \sum_{i=1}^r \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Figure 257 - Formula

$$\frac{\text{Information gain}(S, \{S_1, S_2, \dots, S_r\})}{\text{Information content}(S, \{S_1, S_2, \dots, S_r\})}$$

Figure 258 - Formula

If almost all the instances in a node belong to a single class, that node becomes a leaf node for that class. If not, the algorithm continues partitioning the data until the tree is fully grown. The attribute used for partitioning at each internal node is chosen based on criteria that minimize the number of misclassification.

```
procedure GrowTree(S)
    Partition(S);

procedure Partition (S)
    if (purity(S) > δp or |S| < δs) then
        return;
    for each attribute A
        evaluate splits on attribute A;
    Use best split found (across all attributes) to partition
    S into S1, S2, ..., Sr;
    for i = 1, 2, ..., r
        Partition(Si);
```

Figure 259 - Recursive construction of a decision tree

The task of classification has been a crucial pursuit of artificial intelligence. Among the many algorithms utilized for classification, **decision-tree construction** stands as a popular method for its ease of interpretation and computation.

The crux of this technique lies in the evaluation of **various attributes** and **partitioning conditions**, with the ultimate goal of **maximizing the information-gain ratio**. This process is performed recursively, with each split resulting in a new set of instances for further evaluation. The algorithm halts when the purity of a set is sufficiently high or when the set is too small for further partitioning, resulting in a leaf node with the majority class of its elements. Though decision-tree construction has its merits, other classification techniques have risen to prominence, such as neural net and **Support Vector Machine classifiers**.

Bayesian classifiers, in particular, have been quite useful in their approach to finding the probability of an instance belonging to a given class.

This is accomplished through the estimation of the probability distribution of attribute values for each class, with the class possessing the maximum probability assigned to the instance in question.

The calculation of such probabilities involves **Bayes' theorem**, which relates the probability of an instance belonging to a class with the probability of generating that instance given the class.

The task of classification is a **multifaceted one**, with a plethora of algorithms and techniques at our disposal. Each method has its own strengths and weaknesses, and the choice of which to employ ultimately depends on the specific task at hand.

Association Rules

In the world of retail, understanding the **associations between different items** that customers purchase is key to optimizing sales and improving customer experience. By **identifying these patterns**, businesses can suggest associated items to customers, place products in **strategic locations**, and offer targeted discounts that incentivize customers to purchase related items. The process of discovering these associations involves generating association rules that are defined by support and confidence metrics.

The support metric measures the fraction of the population that satisfies both the **antecedent** and the **consequent** of a rule, while confidence measures how often the **consequent** is **true** when the antecedent is true. By identifying sets of items that have sufficient support, called large itemsets, businesses can output all rules with sufficient confidence that involve all and only the elements of the set.

Generating large item sets can be a computationally intensive process, particularly when the number of items is large. However, optimizations have been developed to eliminate most sets with very low support from consideration. These techniques use multiple passes on the database, considering only some sets in each pass.

$i_1, i_2, \dots, i_n \Rightarrow i_0$

In the a priori technique, for example, sets with single items are considered in the first pass, followed by sets with two items in the second pass, and so on. By the end of each pass, sets with sufficient support are outputted, providing businesses with valuable insights into the associations between different products.

In the end, the ability to generate association rules that accurately capture the behavior of customers can provide businesses with a significant competitive advantage. By leveraging these insights to optimize sales and improve customer experience, businesses can increase their bottom line while building a loyal customer base.

Other Types of Associations

Data mining is a complex process that involves more than just finding associations between items. While plain association rules are useful in some cases, they have limitations. One such limitation is that many associations are predictable, such as the relationship between cereal and bread. A large number of people buying both items does not necessarily indicate a connection between them. This is where correlations come in. By examining positive and negative correlations between items, we can identify more interesting associations that deviate from expected **co-occurrence patterns**.

Another important class of data-mining applications is **sequence associations**. Stock market analysts, for example, are interested in finding *associations among stock-market price* sequences to make informed investment decisions. By discovering deviations from temporal patterns, we can identify unexpected changes that may be of interest. These mining techniques can help us identify correlations that may be overlooked by plain association rules.

Clustering

Clustering is a process of **grouping similar items together**, and it has been extensively studied by the statistics and database research communities. One way to formalize the problem of clustering is to **group points into sets** such that the average distance of points from the centroid of their assigned cluster is minimized. Another way is to minimize the average distance between every pair of points in each cluster. **Hierarchical clustering**, which creates clusters at different levels of hierarchy, is used in biology to classify related species and in internet directory systems to cluster related documents. In fact, hierarchical clustering algorithms can be classified as agglomerative clustering algorithms, which start by building small clusters and then create higher levels, or divisive clustering algorithms, which first create higher levels of hierarchical clustering and then refine each resulting cluster into lower-level clusters.

Clustering also has interesting applications in predicting a person's likely preferences for new movies or books on the basis of their past preferences and the preferences of others with similar past preferences. To find people with similar past preferences, clusters of people are created based on their preferences for movies, and the accuracy of clustering can be improved by previously clustering movies by their similarity. The process of clustering can be repeated to reach an equilibrium, and given a new user, a cluster of users most similar to that user can be found based on the user's preferences for movies already seen. Predictions for new movies or books can then be made based on the preferences of that user's cluster. This problem is an instance of collaborative

filtering, where users collaborate in the task of filtering information to find information of interest.

Other Forms of Data Mining

In the realm of data mining, there exist additional techniques beyond the widely-used methods. One such approach is **text mining**, which applies data-mining principles to textual documents. Through the use of clustering tools, **users can locate previously visited pages** by analyzing the common words present on each page. Furthermore, pages can be automatically classified into a Web directory based on their similarity to other pages.

Data visualization systems offer an alternative means of examining and understanding large volumes of data. These systems employ visual displays, such as maps, charts, and other graphical representations, to convey information in a compact and intuitive manner. For example, by encoding problem locations in a specific color on a map, users can easily identify areas where production issues arise. Data-visualization systems do not automatically detect patterns, but they provide users with the tools to uncover meaningful relationships and associations through visual analysis.

Text mining and **data visualization** are valuable additions to the data-mining toolkit. These techniques help users to uncover hidden insights within complex datasets, providing a means of analysis that complements more traditional approaches.

7.2 Information Retrieval

Unstructured textual data presents a unique set of challenges that require specialized tools and techniques. Enter information retrieval, the art of querying unstructured textual data. Unlike the rigidly structured data of relational databases, **textual data requires** a more nuanced approach that emphasizes issues such as *keyword-based querying, document relevance, and document analysis, classification, and indexing*.

Information retrieval has taken on a whole new dimension with the **advent of web search engines**. These engines go beyond the mere retrieval of documents, instead seeking to satisfy the information needs of users by providing relevant information in response to keyword queries. In this fast-paced world of information overload, effective information retrieval is more important than ever, and the tools and techniques that underlie it will continue to evolve and improve as we seek to unlock the secrets hidden within vast oceans of unstructured data.

Overview

The field of information retrieval has undergone a remarkable evolution alongside the development of databases. The traditional model in information retrieval presupposes a vast number of unstructured documents, with the aim of locating relevant information based on user input such as keywords or example documents. The internet, however, presents a unique challenge, with its vast sea of information and the attendant difficulty of finding anything of interest. **Information retrieval** has been crucial in harnessing the potential of the web and making it a valuable resource, particularly for researchers.

The typical examples of information retrieval systems are online document-management systems, such as library catalogs or newspaper archives, which rely on a collection of **unstructured documents**, and **HTML pages** are considered documents in the context of the web. These systems allow users to retrieve a particular document or class of documents using a set of keywords associated with the document. The search process is based on the keyword or set of keywords supplied by the user, and documents containing those keywords are retrieved.

Keyword-based information retrieval is not only used for retrieving textual data but also for other types of data, such as *video* and *audio data*, which are associated with descriptive keywords. For instance, a video movie might have keywords such as its *title*, *director*, *actors*, and *genre*, while an image or video clip may have tags describing the image or video clip associated with it. However, there are significant differences between this model and the models used in traditional database systems.

Database systems deal with several operations not addressed in information-retrieval systems, such as updates and transactional requirements of **concurrency control** and **durability**. Information-retrieval systems, on the other hand, focus on the issue of querying collections of unstructured documents, addressing keyword queries, and ranking documents on the degree of relevance to the query.

In addition to simple keyword queries, which are just **sets of words**, **information-retrieval systems** also allow query expressions formed using keywords and logical connectives such as and, or, and not. **Full-text retrieval** is essential for unstructured documents, as all words are considered keywords in full-text retrieval. The system estimates the relevance of documents to a query to show them in order of estimated relevance, using information about term occurrences and hyperlink information to estimate relevance.

Modern information-retrieval systems, exemplified by web search engines, have evolved beyond just retrieving documents based on a ranking scheme. Search engines now aim to satisfy a user's information needs, judging what topic a query is about, and

displaying not only top-ranked documents but also other types of information about the topic. For example, a search engine may display cricket scores for the query term “*cricket*” and images of New York for a query “*New York*” alongside web pages related to the topic.

Relevance Ranking Using Terms

As the world of information retrieval expands, the challenge of finding relevant documents in response to a query becomes more daunting. The proliferation of data on the web has only exacerbated this problem, as keyword searches often yield hundreds of thousands of results. Full-text retrieval is even more challenging, as documents can contain many irrelevant terms, and irrelevant documents can be mistakenly retrieved.

To combat this issue, **information-retrieval systems** must estimate the relevance of documents to a query and return only highly ranked documents as answers. However, relevance ranking is not an exact science. One commonly accepted approach is the use of **term frequency and inverse document frequency (TF-IDF)**. This method measures the relevance of a document to a term by taking into account the number of occurrences of the term in the document, adjusted for the number of terms in the document and the number of documents that contain the term.

$$TF(d, t) = \log \left(1 + \frac{n(d, t)}{n(d)} \right)$$

Figure 260 - Formula

This formula can be further refined to account for the placement of the term in the document, such as whether the term appears in the title or abstract.

$$IDF(t) = \frac{1}{n(t)}$$

Figure 261 - Formula

Additionally, when a query contains multiple keywords, weights are assigned to the terms using the inverse document frequency. This ensures that less frequent but more relevant terms are weighted more heavily in the ranking process. However, there are challenges to this approach as well.

For example, stop words such as "and" or "or" are ignored when indexing a document, as their inverse document frequency is extremely low.

$$r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$$

Figure 262 - Formula

Furthermore, similarity-based retrieval systems allow users to find documents that are "similar" to a given document, based on common terms or other factors.

$$\frac{\sum_{i=1}^n r(d, t_i)r(e, t_i)}{\sqrt{\sum_{i=1}^n r(d, t_i)^2}\sqrt{\sum_{i=1}^n r(e, t_i)^2}}$$

Figure 263 - Formula

Despite these challenges, information retrieval remains a critical component of modern computing. By utilizing sophisticated algorithms and cutting-edge technologies, researchers and engineers continue to refine the process of finding the most relevant documents in response to a query, helping to drive progress in a wide range of fields.

Relevance Using Hyperlinks

In the early days of web-search engines, ranking documents was limited to using relevance measures based on **TF-IDF**, with some limitations when dealing with large collections of documents, like the set of all web pages. Web pages often contain all the keywords in a query, while some pages with just a few occurrences of the query terms would not get a high **TF-IDF score**, researchers soon discovered that web pages had a critical piece of information that plain text documents did not have - *hyperlinks*.

Hyperlinks can be exploited to obtain a better ranking of web pages, with the relevance ranking of a page greatly influenced by the hyperlinks that point to it. The basic idea of popularity ranking, also called prestige ranking, is to find and rank popular pages higher than other pages containing the specified keywords. Popularity ranking combines traditional measures of relevance with the page's popularity, allowing for an overall measure of the page's relevance to the query.

Estimating a page's popularity is a challenge, but using hyperlinks to a page as a measure of its popularity has proven to be effective. Many web users maintain pages with links to their favorite sites, and many websites have links to related sites, which can also be used to infer the popularity of the linked sites. A web search engine can fetch web pages and analyze them to find links between pages.

$$P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^N (T[i, j] * P[i])$$

Figure 264 - Formula

One alternative to estimating the popularity of a page is to associate popularity with sites instead of pages. This way, all pages at a site receive the popularity of the site, and pages other than the root page of a popular site benefit from the site's popularity. However, the question of what constitutes a site arises, as there are many sites that host a large number of mostly unrelated pages. A simpler alternative is to allow the transfer of prestige from popular pages to pages to which they link.

[Stanford University](http://stanford.edu)

This notion of popularity is circular, with the popularity of a page defined by the popularity of other pages, and there may be cycles of links between pages. However, the popularity of pages can be defined by a system of simultaneous linear equations, which can be solved by matrix manipulation techniques.

It's interesting to note that the basic idea underlying popularity ranking is quite old and first appeared in a theory of social networking developed by sociologists in the 1950s. In the social networking context, the goal was to define the prestige of people. The use of a set of linear equations to define the prestige of pages is a novel approach to an old problem.

Synonyms, Homonyms, and Ontologies

As we endeavor to search for information on the vast expanse of the internet, we are faced with the challenge of finding relevant documents amidst a sea of data. The task is further complicated by the use of **synonyms** and **homonyms**, which can lead to the retrieval of documents that are irrelevant to our search.

To overcome this hurdle, we must turn to the realm of ontologies and concept-based querying. By understanding the concepts represented by the words in a document, we can match them to the **user's intended search**. The use of hierarchies further allows us to retrieve documents that are related to the user's query, even if the exact terms are not present.

Ontologies are structured hierarchies that define the relationships between concepts. The **WordNet system**, for example, provides a vast array of concepts and associated words. In addition to synonyms, it also defines relationships such as **is-a** and **part-of**, which further solidify the ontology. Such systems have immense potential to aid in information retrieval, as they allow for the standardization of terminology across languages and fields.

By employing the use of **ontologies** and **concept-based querying**, we can navigate the vast oceans of information with ease and precision. It is a remarkable feat of technology that allows us to harness the immense power of the internet, all in the pursuit of knowledge and understanding.

Indexing of Documents

In today's world, the efficient processing of queries in an information-retrieval system is of paramount importance, and an **effective index structure** plays a vital role in achieving this goal. A well-designed index can efficiently locate documents that contain a **specified keyword** by using an **inverted index**, which maps each keyword to a list of documents that contain that keyword.

To support relevance ranking based on **keyword proximity**, the index can also provide a list of locations within the document where the keyword appears. Additionally, the inverted lists may include with each document the term frequency of the term.

Given that these indices must be stored on disk, it is crucial to minimize the number of **I/O operations** required to retrieve each list. To do so, the system would attempt to keep each list in a set of consecutive disk pages, allowing the entire list to be retrieved with just one disk seek.

The and, or, and not operations in information retrieval systems are of great importance. The and operation finds documents that contain all of a specified set of keywords, the or operation retrieves documents containing at least one of the keywords, and the not operation finds documents that do not contain a specified keyword.

To rank documents based on term **frequency**, the index structure should maintain the number of times terms occur in each document. However, to reduce this effort, a compressed representation with only a few bits that approximates the term frequency may be used. The index should also store the document frequency of each term.

Sorting on popularity score is not always fully effective in avoiding long inverted list scans, since it ignores the contribution of the **TF-IDF scores**. An alternative is to break up the inverted list for each term into two parts, with the first part containing documents that have a high TF-IDF score for that term, while the second part contains all documents. Each part of the list can be sorted in order of popularity and document ID.

Designing an effective index structure is a critical factor in the success of information retrieval systems. By carefully implementing the and, or, and not operations and incorporating term frequency and document frequency into the index, we can efficiently and effectively retrieve relevant documents for our users.

Measuring Retrieval Effectiveness

In the field of information retrieval, the effectiveness of a system is often measured by two key metrics: **precision** and **recall**. Precision is the percentage of retrieved documents that are actually relevant to a given query, while recall measures the percentage of relevant documents that are successfully retrieved by the system.

However, accurately determining relevance is a complex and nuanced task that requires a deep understanding of natural language and the user's intent. To address this challenge, **researchers have created collections of documents** and **queries** that have been manually tagged as relevant or irrelevant. This allows for the evaluation of different ranking systems and the measurement of their average precision and recall across multiple queries.

Another challenge in information retrieval is the storage and retrieval of large numbers of documents. To conserve space, the sets of documents associated with a given keyword are often maintained in a compressed form. However, this can lead to inaccuracies in retrieval, with some relevant documents not being retrieved (*false drops*) or some irrelevant documents being retrieved (*false positives*). While false positives are generally more tolerable than false drops, both can significantly impact the overall effectiveness of a retrieval system.

Moreover, ranking strategies can also result in **false negatives** and **false positives**. False negatives occur when relevant documents are ranked low and thus not retrieved, while false positives occur when irrelevant documents are ranked high and thus retrieved. To address these issues, precision and recall can be measured as a function of the number of documents retrieved, and precision can also be measured as a function of recall. These measures can be graphed and used to evaluate the effectiveness of retrieval and ranking strategies across a suite of queries.

Crawling and Indexing the Web

As the world's collective knowledge continues to grow and evolve, so does the task of **organizing** and **indexing** it. In the digital age, web crawlers have emerged as a critical tool for navigating the vast expanse of the internet. These programs are designed to systematically explore the web by following hyperlinks between pages, collecting and processing data along the way.

The process begins with a set of **initial URLs**, which the crawler uses as a starting point. From there, it recursively follows links to other pages, adding new URLs to its to-be-crawled set as it goes. This process continues until all reachable pages have been retrieved and indexed.

Given the sheer size of the web, it's not possible to crawl it in its entirety within a reasonable timeframe. Search engines, therefore, only cover a portion of the web, and their **crawlers** may take weeks or months to complete a single crawl. To manage this workload, crawling is often distributed across multiple machines, each responsible for a subset of the web.

Once pages are retrieved during the crawling process, they are handed over to a prestige **computation** and **indexing system for further processing**. This system may be running on a separate machine, and like the crawling process, may be distributed across multiple machines for efficiency. Pages are usually cached to improve search engine performance, allowing users to access a cached copy of a page even if the original site is down.

To ensure optimal performance, search engines maintain separate copies of their index for queries and updates. These copies are periodically switched, with the old index being updated while the new copy is used for queries.

However, web crawlers do have limitations. Some sites, for example, don't make all their data available through hyperlinks, instead relying on search interfaces. These sites contain what's known as the deep web, and traditional web crawlers can't access this information. To extract data from the deep web, specialized crawlers must be used, which can guess what search terms or options to use to retrieve the desired data.

The challenges of crawling and indexing the web are ongoing, but with the continued development of new technologies and techniques, we can hope to make sense of the ever-expanding universe of information available to us.

Information Retrieval: Beyond Ranking of Pages

In the rapidly evolving landscape of search engines, providing users with relevant information beyond a simple ranked list of pages has become increasingly complex. As users seek to achieve more nuanced goals, such as understanding and answering questions based on a limited understanding of documents, search engines have adapted by exploring novel approaches to information retrieval.

One such approach involves creating structured information from **unstructured documents**, allowing for answering questions based on limited understanding. Another approach utilizes **natural language techniques** to find documents relevant to a query and **return relevant segments of the documents** as an answer.

As search engines have expanded beyond the realm of textual documents and into areas such as image and video search, the challenge of disambiguating ambiguous search terms has grown more complex. To address this issue, search engines have worked to provide a diverse set of results in terms of topic, ensuring that users are presented with relevant information regardless of the specific sense in which a word is used.

To further improve the search experience, specialized snippets have been developed to give users a more meaningful summary of the information they seek. For example, in the case of restaurant searches, a search engine can provide a snippet containing a restaurant's rating, phone number, and a link to a map, in addition to the restaurant's home page.

Information extraction, which involves converting information from a **textual** to a more **structured form**, has become an increasingly important area of focus in the search engine field. By automatically extracting relevant information from sources such as real estate advertisements or scholarly publications, search engines are able to provide users with a more accurate and comprehensive set of results.

Ultimately, the search engine field is constantly evolving, with new approaches and techniques emerging on a regular basis. As the demands of users continue to grow more complex, search engines will undoubtedly continue to innovate in order to provide the best possible search experience.

Directories and Categories

In the labyrinthine expanse of the internet, the quest for information can be an arduous undertaking. As such, a system for organizing and classifying vast amounts of data is necessary to facilitate navigation and access. This is where directories and categories come into play.

Traditionally, libraries employ a classification hierarchy to keep related books in close proximity to each other. This hierarchy is **hierarchical** and **finely structured**, with computer science books residing alongside mathematics books, both of which are science subcategories. At the lowermost level of the hierarchy, computer science books are broken down further into subcategories such as **algorithms**, **languages**, and **operating systems**. In contrast, information-retrieval systems have no need to physically store related documents in close proximity. Nevertheless, these systems must logically organize documents to enable efficient browsing. Therefore, such systems may use a classification hierarchy similar to libraries, with the added functionality of displaying related documents and descriptions when displaying a particular document.

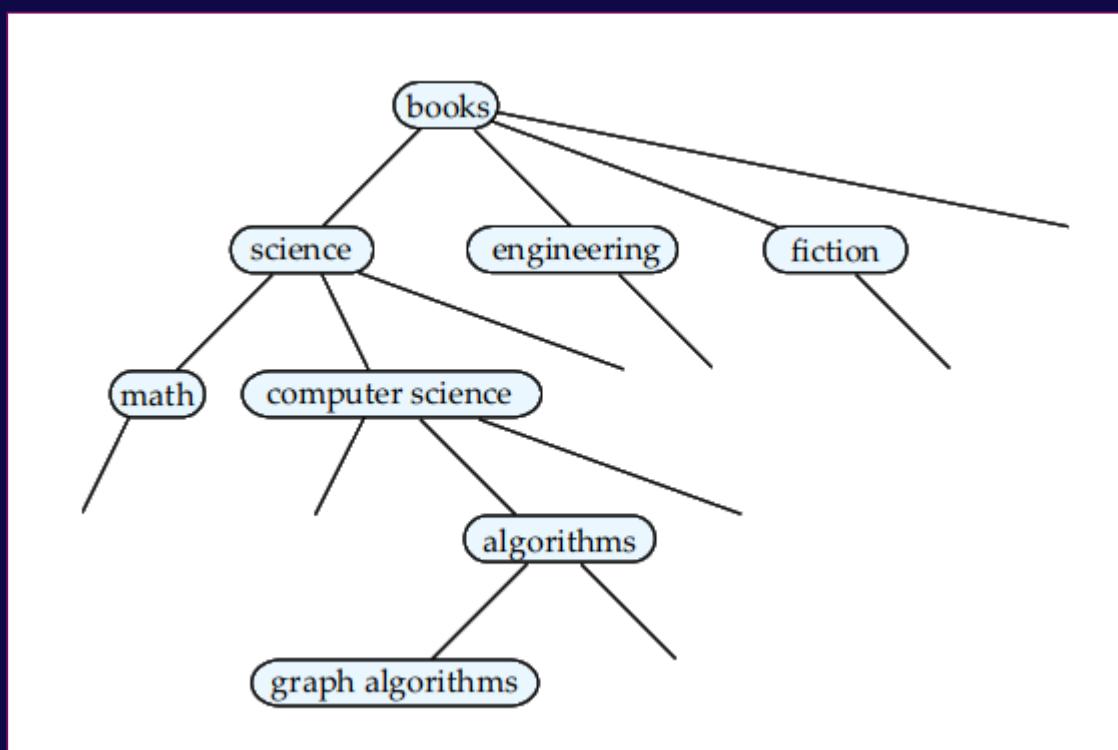


Figure 265 - A classification hierarchy for a library system

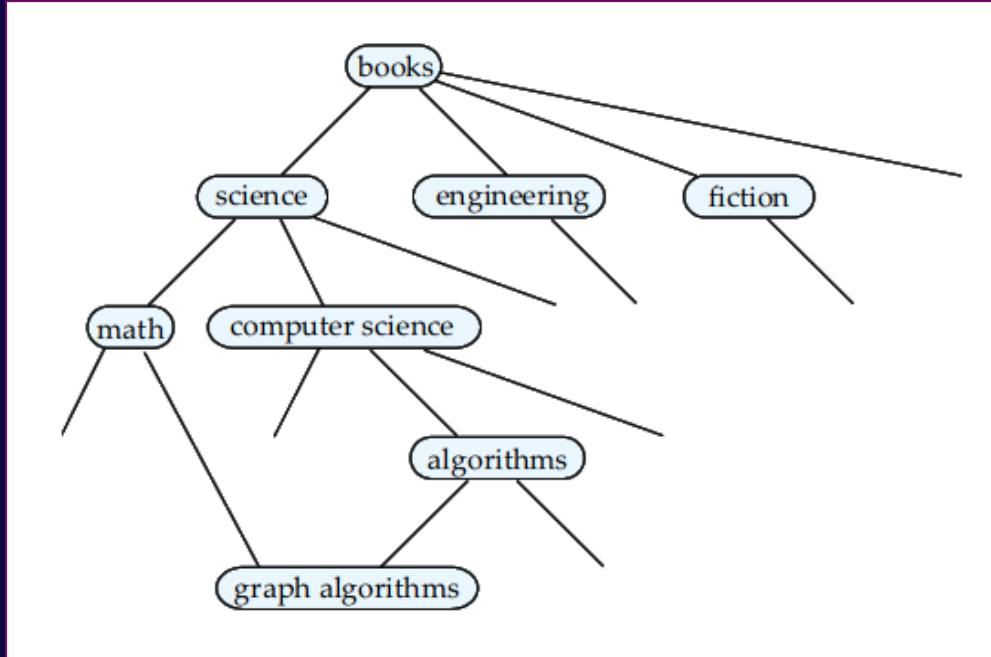


Figure 266 - A classification DAG for a library information-retrieval system

Unlike libraries, documents in an information-retrieval system can be classified into multiple categories. A document about mathematics for computer scientists can be classified under mathematics and computer science. This creates a **directed acyclic graph (DAG)** hierarchy structure, where documents and subcategories can occur under multiple areas.

Directories, which are **DAG structures**, store links to documents and other relevant information. Users begin at the root node and navigate through the hierarchy until reaching the desired topic, encountering related documents and subclasses in the process.

The task of organizing vast amounts of data on the internet is a daunting one. Establishing an appropriate directory hierarchy is the first challenge. This may be achieved through manual classification by librarians or by using automated techniques based on similarity metrics. Additionally, the relevance of a particular document to different nodes in the hierarchy is determined manually or through automated algorithms.

Directories and categories are indispensable tools for navigating the vast expanses of information available on the internet. While their creation and maintenance are challenging, they provide a powerful means of organizing and accessing information.

8 SPECIALTY DATABASES

Researchers have developed data models based on an object-oriented approach to deal with application domains that are restricted by the limitations of the relational data model. The **object-relational model** combines features of the **relational** and **object-oriented models**, providing a rich type system of object-oriented languages combined with relations as the basis for data storage. This model applies inheritance to relations, providing a smooth migration path from relational databases, which is attractive to vendors. Object-oriented databases support direct access to data from **object-oriented programming languages** without requiring a relational query language as the database interface. **XML language**, initially designed to add markup information to text documents, provides a way to represent data with nested structures and allows flexibility in structuring nontraditional data. It has become important due to its applications in data exchange. **XQuery XML query language** and **SQL/XML**, an extension of SQL, enable the creation of nested XML output and different ways of expressing queries on data represented in XML.

8.1 Object-Based Databases

As the use of database systems expanded beyond traditional data-processing tasks like banking and payroll management, limitations of the relational model began to hinder the development of more complex applications, such as computer-aided design and geographical information systems. The solution came in the form of object-based databases, which opened up the potential for dealing with complex data types in a more flexible manner.

Overview

The relational data model has long been the cornerstone of database systems, but as applications evolved to become more complex, the limitations of the model began to emerge. Programmers found themselves grappling with the inadequacy of the relational model's type system when dealing with data structures such as nested records and inheritance, which are easily handled by traditional programming languages. Enter object-based databases, which offer a richer type system, including object orientation and complex data types, making it possible to tackle complex application domains with ease.

However, the development of complex data types wasn't the only obstacle faced by database programmers. Accessing database data from programs written in programming languages like *Java* and *C++* proved to be a challenge, as the type system of the database didn't always match the type system of the programming language. To simplify data storage and retrieval, database products needed programming language constructs or extensions that permit direct access to data in the database, without the need for an intermediate language like SQL.

It's important to keep in mind that most support only a subset of the SQL features described here, with syntax often differing slightly from the standard, so users should refer to their database system's user manual to determine which features it supports.

Complex Data Types

In the fast-evolving world of database technology, demands have grown for ways to deal with more complex data types. Traditional database applications have been limited to conceptually simple data types, where basic data items are small and have atomic fields. However, this view can sometimes hide important details, such as addresses, which can be of interest to queries. The solution is to allow structured data types that permit more complex subparts.

<i>title</i>	<i>author_array</i>	<i>publisher</i> <i>(name, branch)</i>	<i>keyword_set</i>
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

Figure 267 - Non-1NF books relation, books

Furthermore, certain applications are better modeled as a set of objects or entities, rather than a set of records. For example, in a library application, multiple records may be required to represent a single book with multiple authors and a set of keywords. This requires the use of non-atomic domains, which can be efficiently represented using nested relations.

Although the use of complex data types such as sets and arrays can be beneficial in many applications, it should be used with caution to avoid data redundancy and ensure efficient query processing. As database technology continues to evolve, it is crucial to keep up with the latest trends and innovations to stay ahead of the curve.

Structured Types and Inheritance in SQL

In the early days of SQL, the type system was limited to a set of basic pre-defined types. However, the advent of *SQL:1999* brought about a significant expansion to the SQL type system, enabling structured types and inheritance.

Structured types allow for composite attributes from **E-R designs** to be directly represented within SQL. For instance, a structured type can be created to capture a composite attribute for a name or an address. The former could be defined as follows:

```
(firstname varchar(20),  
 lastname varchar(20))  
final;
```

With structured types, tables can be created that contain attributes of these types. For example, a table for personal information could include composite attributes for name and address as follows:

```
name Name,  
address Address,  
dateOfBirth date);
```

Using a “dot” notation, individual components of these composite attributes can be accessed, such as the first name component of the name attribute. Alternatively, a table can be created where rows correspond to a user-defined type, as demonstrated by the creation of the PersonType and the Person table.

```
create type PersonType as  
  (name Name, address Address, dateOfBirth date)  
  not final  
  method ageOnDate(onDate date)  
  returns interval year;  
  create instance method ageOnDate  
  (onDate date)  
  returns interval year  
  for PersonType  
begin  
  return onDate - self.dateOfBirth;  
end
```

The introduction of structured types and inheritance in *SQL:1999* significantly expanded the type system and enabled more expressive and flexible database designs.

Structured types in SQL offer a powerful mechanism for defining and manipulating complex data structures, complete with methods and constructors. By employing structured types, SQL provides a means to encapsulate functionality and data together, improving code organization and readability.

```
select name.lastname,
       ageOnDate(current date)
  from person;

  create function Name
    (firstname varchar(20), lastname varchar(20))

  returns Name

begin
  set self.firstname = firstname;
  set self.lastname = lastname;
end
```

Methods are an integral part of structured types, and allow for additional functionality to be defined on top of a type's attributes. Declared in the type definition, methods execute on instances of the type and can contain procedural statements that update the instance attributes. Methods can also be invoked on instances of a type to return computed values.

Constructor functions, which enable the creation of new instances of a type, are employed in *SQL:1999*. Constructor functions are named after their corresponding structured type, and can have one or more arguments with distinct types. The SQL standard also requires that every structured type has a default constructor that sets the attributes to their default values.

```
insert into Person
  values
    (new Name('John', 'Smith'),
     new Address('20 Main St', 'New York', '11001'), date '1960-8-22');

  create type Person
    (name varchar(20),
```

```
address varchar(20));  
  
create type Student  
under Person  
  
(degree varchar(20),  
department varchar(20));  
  
create type Teacher  
under Person  
  
(salary integer,  
department varchar(20));
```

Type inheritance allows for the creation of subtypes from supertypes, thereby enabling code reuse and organization. In SQL, subtypes can inherit both the attributes and methods of their supertypes, with the option to redefine inherited methods. In the event of multiple inheritance, where a type is a subtype of multiple types, conflicts can arise with duplicate attribute names. Renaming the attributes with the use of the “as” clause can resolve such conflicts.

```
create type TeachingAssistant  
under Student with (department as student dept),  
Teacher with (department as teacher dept);
```

Structured types in SQL offer a flexible and powerful mechanism for defining and manipulating complex data structures, with support for methods, constructors, and inheritance.

Table Inheritance

In the world of SQL, the concept of table inheritance can prove to be both powerful and confounding. At its core, table inheritance represents a way to establish relationships between tables by establishing subtables that inherit from a parent table. This enables us to model complex relationships between entities, similar to how specialization and generalization are modeled in the **entity-relationship (E-R) notation**.

```
create table people of Person;
create table students of Student
    under people;
create table teachers of Teacher
    under people;
```

Take, for instance, a scenario in which we want to create subtables for students and teachers. We begin by establishing a parent table called "*people*", defined as "*create table people of Person*". We can then create subtables for students and teachers under the *people* table, using "*create table students of Student under people*" and "*create table teachers of Teacher under people*", respectively.

```
delete from people where P;
create table teaching assistants
    of TeachingAssistant
under students, teachers;
```

Notably, every attribute present in the parent table is also present in the subtables. Additionally, when we declare subtables under a parent table, every tuple present in the subtables also becomes implicitly present in the parent table. However, only attributes that exist in the parent table can be accessed by a query.

To find tuples present in the parent table but not in its subtables, we can use the "*only*" keyword in our query. Furthermore, there are consistency requirements that must be satisfied when using subtables, including constraints around the correspondence between tuples in the subtable and parent table.

While SQL subtables offer significant benefits in terms of modeling relationships between tables, they are not without their limitations. For example, SQL does not support multiple inheritances, which can limit our ability to model certain scenarios effectively. However, with a little extra effort, we can create our own implementation of the subtable mechanism using existing SQL features.

Array and Multiset Types in SQL

Structured Query Language (SQL) is a versatile database management system that offers a range of collection types to support complex data modeling. Among these types are arrays and multisets, which were respectively introduced in *SQL:1999* and *SQL:2003*. While both types can store multiple values, they differ in their order and repetition constraints. A multiset can contain multiple instances of the same element and is unordered, whereas an array stores elements in a specific order.

For instance, in a database that records book information, we may need to associate a set of keywords and an ordered array of authors with each book. To accomplish this, we can define a custom type using the SQL syntax, such as "*Publisher*" with "*name*" and "*branch*" fields, and "*Book*" with "*title*," "*author*," "*pub date*," "*publisher*," and "*keyword*" fields. Here, the "*author*" field is an array of up to **10** author names, while the "*keyword*" field is a multiset of up to **20** character strings.

```
create type Publisher as
    (name varchar(20),
     branch varchar(20));

create type Book as
    (title varchar(20),
     author array varchar(20)
      array [10],
     pub date date,
     publisher Publisher,
     keyword set varchar(20) multiset);

create table books of Book;
```

To create a table of books with this schema, we can use the "*create table*" statement in SQL. We can then populate the table with values using the "*insert into*" statement, where we construct a tuple that matches the Book type. To access or modify the elements of an array, we can use the array index notation.

```
array['Silberschatz', 'Korth', 'Sudarshan']

multiset['computer', 'database', 'SQL']

('Compilers', array['Smith', 'Jones'], new
Publisher('McGraw-Hill', 'New York'), multiset['parsing',
'analysis'])

insert into books
```

```

values ('Compilers', array['Smith', 'Jones'],
       new Publisher('McGraw-Hill', 'New York'),
       multiset['parsing', 'analysis']);

```

SQL's support for array and multiset types provides a powerful tool for modeling complex data relationships and querying them efficiently. By leveraging the appropriate type for each attribute, we can maintain the integrity and meaning of the data and achieve a more expressive and concise database schema.

In the realm of database management systems, **handling collection-valued** attributes in queries can be a formidable challenge. In this regard, the ability to manipulate expressions that evaluate a collection is of paramount importance. These expressions can be used in various SQL clauses, such as the '*from*' clause, for instance. To demonstrate this, let us consider the '*books*' table that we introduced earlier.

Suppose we wish to retrieve all the books that contain the word "*database*" as one of their keywords. We can achieve this by using the following query:

```

from books
where 'database' in (unnest(keyword set));

```

It is worth noting that we have used the 'unnest(keyword set)' expression in a position that would have required a select-from-where subexpression in SQL without nested relations.

<i>title</i>	<i>author</i>	<i>pub_name</i>	<i>pub_branch</i>	<i>keyword</i>
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Figure 268 - Example

Alternatively, let's say we want to retrieve the first three authors of a particular book. We could write the following query:

```
from books  
where title = 'Database System Concepts';
```

Now, suppose we want to create a relation that contains pairs of the form "*title, author name*" for each book and each author of the book. In this scenario, we can use the following query:

```
from books as B,  
unnest(B.author array) as A(author);
```

Since the '*author array*' attribute of the '*books*' table is a collection-valued field, the '*unnest(B.author array)*' expression can be used in a '*from*' clause where a relation is expected. It is also worth noting that the tuple variable '*B*' is visible to this expression since it is defined earlier in the '*from*' clause.

It is important to note that when we unnest an array, we lose information about the ordering of elements in the array. We can use the '*unnest with ordinality*' clause to retain this information. The following query illustrates how we can use this clause to generate the '*authors*' relation from the '*books*' relation:

```
from books as B,  
unnest(B.author array) with ordinality as A(author, position);
```

In this case, the '*with ordinality*' clause generates an extra attribute that records the position of the element in the array. We can use a similar query, but without the '*with ordinality*' clause, to generate the '*keyword*' relation.

Moving on, let us consider the concept of nesting and unnesting. The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called unnesting. In the '*books*' table, there are two attributes, '*author array*' and '*keyword set*', that are collections, and two attributes, '*title*' and '*publisher*', that are not. If we wish to convert the '*books*' relation into a single flat relation, with no nested relations or structured *types* as attributes, we can use the following query:

```
as pub name, publisher.branch as pub branch,  
K.keyword  
from books as B,  
unnest(B.author array) as A(author),  
unnest (B.keyword set) as K(keyword);
```

In this query, the variable '*B*' in the '*from*' clause is declared to range over '*books*'. The variable '*A*' is declared to range over the authors in the '*author array*' for the book '*B*', and '*K*' is declared to range over the keywords in the '*keyword set*' of the book '*B*'.

Nesting is the reverse process of converting a **first normal form (1NF)** relation into a nested relation, achieved by extending grouping in SQL. The grouping operation in SQL generates a temporary multiset relation for each group, on which an aggregate function is applied to derive a single value. By utilizing the collect function, a nested relation can be created in place of a single value.

To illustrate, suppose we have a **1NF** relation called "*flat books*". The query select title, author, Publisher(pub name, pub branch) as publisher, collect(keyword) as keyword set from flat books group by title, author, publisher; nests the relation on the "*keyword*" attribute, returning a result as seen in Figure:

<i>title</i>	<i>author</i>	<i>publisher</i>	<i>keyword_set</i>
		(pub_name, pub_branch)	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Figure 269 - A partially nested version of the flat book's relation

To nest the "*author*" attribute into a multiset, the query select title, **collect(author)** as author set, Publisher(pub name, pub branch) as publisher, collect(keyword) as keyword set from flat books group by title, publisher; can be used.

An alternative approach to creating nested relations is to use subqueries in the select clause. This approach allows for an order by clause to be used in the subquery, providing control over the order of results. The system executes the nested subqueries for each tuple generated by the from and where clauses of the outer query. SQL:2003 includes several operators for multisets, such as a function that returns a duplicate-free version of a multiset and an aggregate operation that returns the intersection or union of all multisets in a group.

Unfortunately, the SQL standard does not provide a method for updating multiset attributes aside from assigning a new value. For instance, deleting a value from a multiset attribute would require setting it to (**A except all multiset[v]**).

Object-Identity and Reference Types in SQL

The world of databases can be a complex one, filled with intricate structures and relationships that require careful management. One such structure is the use of references in SQL, which provide the ability to refer to objects in a manner similar to object-oriented languages. In SQL, we can define a type with a field that is a reference to another type, such as a type Department with a field head that is a reference to the type Person.

```
create type Department  
  ( name varchar(20),  
    head ref(Person) scope people);  
  
create table departments of Department;  
  
create table departments of Department  
  (head with options scope people);  
  
create table people of Person  
  ref is person id system generated;
```

However, the use of references in SQL is not without its challenges. The scope of a reference is restricted to tuples of a table, which means that the referenced table must have an attribute that stores the identifier of the tuple. This attribute called the self-referential attribute, must be declared using the ref is a clause in the create table statement.

```
create table departments of Department  
  (head with options scope people);  
  
create table people of Person  
  ref is person id system generated;
```

To initialize a reference attribute, the identifier value of the referenced tuple must be obtained through a query. This can be achieved by first creating the tuple with a null reference and then setting the reference separately. Alternatively, users can generate identifiers for their own references, but this requires additional specifications in the table definitions.

```
insert into people  
  (person id, name, address)  
values ('01284567', 'John', '23 Coyote Run');  
  
insert into departments values ('CS', '01284567');
```

```
create type Person  
        (name varchar(20) primary key,  
         address varchar(20))  
        ref from(name);  
  
create table people of Person  
        ref is person id derived;
```

Despite these challenges, the use of references can simplify SQL queries and hide join operations. Dereferencing a reference using **the -> symbol** allows for easy access to the attributes of the referenced tuple. With careful management and understanding of the underlying structures, the use of references in SQL can provide a powerful tool for database management.

```
insert into departments  
        values ('CS', 'John');  
  
select head->name, head->address  
        from departments;  
  
select deref(head).name  
        from departments;
```

Implementing O-R Features

Object-relational database systems, while an extension of traditional relational database systems, require significant changes at multiple levels of the database system. In order to minimize modifications to the storage-system code, the elaborate data types supported by object-relational systems can be transformed into the simpler type system of relational databases.

The translation of certain features of the **E-R model** to relations offers insight into how this transformation can be achieved. For instance, multivalued attributes in the **E-R model** correspond to multiset-valued attributes in the object-relational model. Similarly, composite attributes roughly correspond to structured types. **ISA hierarchies** in the **E-R model** correspond to table inheritance in the object-relational model. Techniques previously used to convert **E-R model** features to tables, with some extensions, can be used to translate object-relational data to relational data at the storage level.

Subtables can be stored in an efficient manner without the replication of all inherited fields through either of two methods:

- Each table stores only the primary key and locally defined attributes, with inherited attributes derived by joining with the supertable;
- Or each table storing all inherited and locally defined attributes, with the tuple stored only in the table in which it is inserted and its presence inferred in each of the supertables.

However, certain types of entity representation may result in the replication of information, and it can be challenging to **translate foreign keys** referring to a supertable into constraints on the subtables. In order to implement such foreign keys efficiently, the supertable must be defined as a view, and the database system must support foreign keys on views.

Array and **multiset types** may be represented directly or using a normalized representation internally. Although normalized representations may be easier to implement, they tend to take up more space and require an additional join/grouping cost to collect data in an array or multiset.

ODBC and **JDBC** application program interfaces have been extended to retrieve and store structured types. **JDBC** provides a method **getObject()** that returns a *Java Struct object*, from which the components of the structured type can be extracted. It is also possible to associate a Java class with an **SQL structured type**, and **JDBC** will then convert between the types. Further details can be found in the **ODBC** or **JDBC** reference manuals.

Persistent Programming Languages

In the realm of software development, the concept of persistent programming languages has recently been gaining traction. These languages, which enable **direct manipulation of persistent data**, have been found to offer distinct advantages over traditional programming languages that only interact with non-persistent data, such as files. The former provides tighter integration with database languages and can be particularly effective for implementing user interfaces or facilitating communication with other computer systems.

Persistent programming languages, however, are not without their drawbacks. Given the complexity of these powerful languages, it can be relatively easy to make programming errors that compromise the integrity of the database. Furthermore, **automatic high-level optimization**, such as the reduction of disk **I/O**, is more challenging with persistent programming languages. Nevertheless, the benefits of these languages are considerable and warrant consideration.

The transformation of **object-oriented programming languages** into database programming languages is an area of particular interest. Several methods for enabling the persistence of objects have been proposed, including persistence by class, by creation, by marking, and by reachability. Each method has its strengths and weaknesses, and language developers must carefully consider which method best suits their purposes.

Language-independent issues, as well as those specific to **C++** and **Java**, must be taken into account when incorporating persistence into an existing programming language. Although several standards have been proposed, none have achieved universal acceptance. In the meantime, the interested reader is encouraged to explore the bibliographical references for further details on specific language extensions and implementation.

The concept of persistence has long been a topic of interest. Recently, persistent **C++** systems have garnered attention as a means of supporting persistence without altering the **C++** language itself. While the architecture of these systems may differ, they share common programming features, such as **utilizing object-oriented C++ features** to support persistence.

To make a class persistent, one approach is to declare a class called Persistent Object with attributes and methods to support persistence, and any class that should be persistent can be made a subclass of this class. Additionally, **C++** provides the ability to redefine standard function names and operators, which can be used to behave in the required manner when operating on persistent objects. This ability is called overloading.

Class libraries provide a means of **minimal C++** alterations and ease of implementation for supporting persistence. However, this method requires more time for programmers to write programs handling persistent objects, and it is challenging to specify integrity constraints on the schema and provide support for declarative querying. Some persistent **C++** implementations offer syntax extensions to mitigate these challenges.

When adding persistence support to **C++ (and other languages)**, certain aspects must be considered. For example, new data types must be defined to represent persistent pointers. **ODMG C++** standard uses a **template class d Ref< T >**, while **ObjectStore** uses a technique called "*hardware swizzling*" to address the problem of in-memory pointer sizes being too small for larger databases.

Another important aspect is the creation of persistent objects, which can be achieved by overloading the **C++** new operator to create persistent objects that are automatically added to a database.

Persistent **C++** systems also automatically create and maintain class extents for each class and provide a way to specify integrity constraints and enforce them by automatically creating and deleting pointers.

To support iteration over class members, an interface is required to iterate over members of a class extent. The iterator interface also allows selections to be specified, so only objects satisfying the selection predicate need to be fetched.

Persistent **C++** systems also provide support for transactions, allowing for starting a transaction, committing it, or rolling it back. However, detecting when an object has been updated poses a challenge. While some persistent extensions to **C++** require the programmer to explicitly specify that an object has been modified, other systems use memory-protection support provided by the *operating system/hardware* to detect writes to a block of memory and mark it as a dirty block that should be written later to disk.

Object-Relational Mapping

The integration of object-oriented data models and programming languages has been a challenging, yet vital pursuit. Two approaches have thus far been explored, but **object-relational mapping (ORM)** provides a compelling third option for integration.

ORM systems are constructed upon a traditional relational database foundation and allow programmers to establish a mapping between tuples in database relations and objects in the programming language. These objects, however, are transient in nature, lacking permanent identity. Nevertheless, a programmer can retrieve objects, or sets of objects, based on selection conditions on their attributes. Relevant data is then retrieved from the underlying database and objects are created based on the predetermined mapping.

Programmers can then update or delete objects and command the **ORM system** to save changes, resulting in corresponding updates, inserts, or deletions of tuples in the database. Hibernate, a widely used **ORM system**, provides object-relational mapping to Java. **ORM systems** strive to simplify the task of programmers by presenting them with an object model while retaining the benefits of utilizing a robust relational database underneath. Moreover, **ORM systems** can significantly enhance performance over direct database access when operating on objects cached in memory.

ORM systems also provide query languages for programmers to write queries directly on the object model, which are then translated into SQL queries on the underlying relational database. The resulting objects are then created from the SQL query results.

However, bulk database updates may incur significant overheads, and querying capabilities can be limited. Nonetheless, programmers may bypass the **ORM system** to **directly update the database** and **compose intricate queries directly in SQL**. Despite the drawbacks, the advantages of object-relational models are sufficient for numerous applications, and ORM systems have seen widespread adoption in recent years.

Object-Oriented versus Object-Relational

In our exploration of database systems, we have encountered three distinct approaches to integrating object-oriented data models with relational databases:

- **Object-oriented databases (OODBs)**
- **Object-relational databases (ORDBs)**
- **Object-relational mapping (ORM) systems**

While each approach has its unique strengths and weaknesses, they all aim to facilitate data modeling and querying, while providing different degrees of protection against programming errors and optimizing data storage and retrieval.

OODBs, for instance, are built around persistent programming languages and eliminate the need for data translation when manipulating data with a programming language. This results in low-overhead access to persistent data and high performance, making them ideal for applications such as **CAD databases**. However, they are more susceptible to data corruption due to programming errors, and they generally lack powerful querying capabilities.

ORDBs, on the other hand, are built on top of the relational model, making them more robust and better suited for complex data storage and querying, including multimedia data. The SQL language, which is declarative and has limited power compared to programming languages, provides a good degree of protection against programming errors and makes high-level optimizations relatively easy.

ORM systems, meanwhile, provide a layer on top of traditional relational databases that allow programmers to build applications using an object model. This provides the robustness of widely used relational database systems while offering the power of object models for application development. However, **ORM systems** suffer from overheads of data conversion between the object model and the relational model used to store data.

In general, each approach has its own strengths and is best suited for a particular type of application. Relational systems are great for simple data types and powerful query languages, **persistent programming language-based OODBs** are ideal for *complex data types and high performance*, *object-relational systems* are well-suited for *complex data types and powerful query languages*, and **ORM systems** offer complex data types integrated with programming languages and are designed as a layer on top of a relational database system.

It is important to note, however, that some database systems blur the boundaries between these approaches. **Object-oriented database systems** built around a persistent programming language can be implemented on top of a relational or object-relational database system, for instance. Such systems may provide lower performance than object-oriented database systems built directly on a storage system, but provide some of the stronger protection guarantees of relational systems.

8.2 XML

The **Extensible Markup Language (XML)** is a versatile tool for structuring data and has found its way into many applications beyond its original purpose in document management. With roots in the **Standard Generalized Markup Language (SGML)**, **XML** was designed to represent data and has proven invaluable for communication and integration between applications. However, its application in this context has raised many database-related concerns, including how to organize, manipulate, and query XML data. In this chapter, we delve into the intricacies of XML, exploring techniques for managing **XML data** with database systems and exchanging data formatted as XML documents. Through our analysis, we seek to provide readers with a comprehensive understanding of the role of XML in modern data management and the challenges that arise when integrating it with databases.

Motivation

XML, the extensible markup language, is a technology with deep roots in the history of document processing. At its core, **markup language refers to any element** within a document that is not intended to be part of the final printed output. For example, a writer may need to include notes on how to typeset a document, which must be distinguished from the actual content to avoid ending up in the printed material. In electronic document processing, markup language is used to describe what part of a document is content, what part is markup, and what the markup means.

Database System Concepts

The evolution of markup languages from specifying instructions on how to print parts of the document to specifying the function of the content mirrors the evolution of database systems from physical file processing to providing a **separate logical view**. With functional markup, text representing section headings or other elements is marked up to reflect their meaning, rather than simply how they should be printed.

While HTML, **SGML**, and **XML** are all markup languages that use tags enclosed in angle brackets, XML distinguishes itself from the others by not prescribing a set of tags to use. This feature enables **XML** to play a major role in data representation and exchange, while **HTML** is primarily used for document formatting.

```
<university>
  <department>
    <dept.name> Comp. Sci. </dept.name>
    <building> Taylor </building>
    <budget> 100000 </budget>
  </department>
  <department>
    <dept.name> Biology </dept.name>
    <building> Watson </building>
    <budget> 90000 </budget>
  </department>
  <course>
    <course.id> CS-101 </course.id>
    <title> Intro. to Computer Science </title>
    <dept.name> Comp. Sci </dept.name>
    <credits> 4 </credits>
  </course>
  <course>
    <course.id> BIO-301 </course.id>
    <title> Genetics </title>
    <dept.name> Biology </dept.name>
    <credits> 4 </credits>
  </course>
```

Figure 270 - XML representation of (part of) university information

```
<instructor>
    <IID> 10101 </IID>
    <name> Srinivasan </name>
    <dept.name> Comp. Sci. </dept.name>
    <salary> 65000 </salary>
</instructor>
<instructor>
    <IID> 83821 </IID>
    <name> Brandt </name>
    <dept.name> Comp. Sci. </dept.name>
    <salary> 92000 </salary>
</instructor>
<instructor>
    <IID> 76766 </IID>
    <name> Crick </name>
    <dept.name> Biology </dept.name>
    <salary> 72000 </salary>
</instructor>
<teaches>
    <IID> 10101 </IID>
    <course.id> CS-101 </course.id>
</teaches>
<teaches>
    <IID> 83821 </IID>
    <course.id> CS-101 </course.id>
</teaches>
<teaches>
    <IID> 76766 </IID>
    <course.id> BIO-301 </course.id>
</teaches>
</university>
```

Figure 271 - Continuation

The power of **XML** is exemplified by its ability to *represent complex, structured information in files, and exchange data between organizations*. The format of an XML document is not rigid and is self-documenting, as the presence of tags enables the meaning of the text to be readily understood.

Additionally, the format can evolve over time, without invalidating existing applications. **XML** allows nested structures, and the purchase order example provided in the Figure below highlights the benefits of this feature.

```
<purchase_order>
  <identifier> P-101 </identifier>
  <purchaser>
    <name> Cray Z. Coyote </name>
    <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
  </purchaser>
  <supplier>
    <name> Acme Supplies </name>
    <address> 1 Broadway, New York, NY, USA </address>
  </supplier>
  <itemlist>
    <item>
      <identifier> RS1 </identifier>
      <description> Atom powered rocket sled </description>
      <quantity> 2 </quantity>
      <price> 199.95 </price>
    </item>
    <item>
      <identifier> SG2 </identifier>
      <description> Superb glue </description>
      <quantity> 1 </quantity>
      <unit-of-measure> liter </unit-of-measure>
      <price> 29.95 </price>
    </item>
  </itemlist>
  <total.cost> 429.85 </total.cost>
  <payment.terms> Cash-on-delivery </payment.terms>
  <shipping.mode> 1-second-delivery </shipping.mode>
</purchaseorder>
```

Figure 272 - XML representation of a purchase order

While the **XML representation** of data may be less efficient than storing data in a relational database due to the repetition of tag names throughout the document, its flexibility and self-documenting nature make it a valuable tool for data representation and exchange.

Structure of XML Data

In the realm of data representation, the fundamental construct of an **XML document** is none other than the element. Simple in its essence, an element is comprised of matching start and end tags, encapsulating all the text that exists between them.

...

However, an **XML document** must have a single root element that envelops all other elements present within the document. Furthermore, for elements in an XML document to be properly structured, they must be nested in a correct manner, with every start tag having a matching end tag that is in the same parent element.

```
...
<course>
    This course is being offered for the first time in 2009.
    <course.id> BIO-399 </course.id>
    <title> Computational Biology </title>
    <dept.name> Biology </dept.name>
    <credits> 3 </credits>
</course>
...
```

Figure 278 - Mixture of text with subelements

```
<university-1>
  <department>
    <dept.name> Comp. Sci. </dept.name>
    <building> Taylor </building>
    <budget> 100000 </budget>
    <course>
      <course.id> CS-101 </course.id>
      <title> Intro. to Computer Science </title>
      <credits> 4 </credits>
    </course>
    <course>
      <course.id> CS-347 </course.id>
      <title> Database System Concepts </title>
      <credits> 3 </credits>
    </course>
  </department>
  <department>
    <dept.name> Biology </dept.name>
    <building> Watson </building>
    <budget> 90000 </budget>
    <course>
      <course.id> BIO-301 </course.id>
      <title> Genetics </title>
      <credits> 4 </credits>
    </course>
  </department>
  <instructor>
    <IID> 10101 </IID>
    <name> Srinivasan </name>
    <dept.name> Comp. Sci. </dept.name>
    <salary> 65000. </salary>
    <course.id> CS-101 </course.id>
  </instructor>
</university-1>
```

Figure 279 - Nested XML representation of university information

```

<university-2>
  <instructor>
    <ID> 10101 </ID>
    <name> Srinivasan </name>
    <dept.name> Comp. Sci.</dept.name>
    <salary> 65000 </salary>
    <teaches>
      <course>
        <course.id> CS-101 </course.id>
        <title> Intro. to Computer Science </title>
        <dept.name> Comp. Sci. </dept.name>
        <credits> 4 </credits>
      </course>
    </teaches>
  </instructor>

  <instructor>
    <ID> 83821 </ID>
    <name> Brandt </name>
    <dept.name> Comp. Sci.</dept.name>
    <salary> 92000 </salary>
    <teaches>
      <course>
        <course.id> CS-101 </course.id>
        <title> Intro. to Computer Science </title>
        <dept.name> Comp. Sci. </dept.name>
        <credits> 4 </credits>
      </course>
    </teaches>
  </instructor>
</university-2>

```

Figure 280 - Redundancy in nested XML representation

As is common with several other aspects of XML, the freedom to intermingle text with subelements within an element finds a more logical application in document processing rather than data processing contexts.

The ability to nest elements within one another offers an alternative means of representing information, as is demonstrated in the Figure:

```
...<course course.id= "CS-101">
    <title> Intro. to Computer Science </title>
    <dept.name> Comp. Sci. </dept.name>
    <credits> 4 </credits>
</course>
...
```

Figure 281 - Use of attributes

In addition to elements, XML specifications outline the concept of an attribute. Attributes are represented as name=value pairs before the closing ">" of a tag and can appear only once in a given tag, unlike subelements which may be repeated.

Although nested representations are frequently used in **XML data** interchange applications to **avoid joins**, they may lead to redundant data storage. In contrast, the normalized representation of XML data requires a join of purchase order records with a company address in relation to access address information.

XML documents are designed to be shared between applications, hence the introduction of a namespace mechanism. This feature enables organizations to specify globally unique names to be used as element tags in documents, thereby avoiding duplication with other XML documents.

```
<![[CDATA[<course> ...</course>]]>
<university xmlns:yale="http://www.yale.edu"> ...
    <yale:course>
        <yale:course.id> CS-101 </yale:course.id> <yale:title>
        Intro. to Computer Science </yale:title> <yale:dept name>
        Comp. Sci. </yale:dept name> <yale:credits> 4 </
        yale:credits>
    </yale:course> ...
</university>
```

Figure 282 - Unique tag names can be assigned by using namespaces

XML Document Schema

The schema for an **XML document** is an often-overlooked yet critical aspect of data processing. While databases use schemas to restrict data types and ensure data integrity, XML documents, by default, lack any schema. This freedom of structure may be suitable for **self-described data formats**, but it hampers the automatic processing of data and formatting of related data in XML.

```
<!DOCTYPE university [  
    <!ELEMENT university ( (department|course|instructor|teaches)+)>  
    <!ELEMENT department ( dept.name, building, budget)>  
    <!ELEMENT course ( course.id, title, dept.name, credits)>  
    <!ELEMENT instructor (IID, name, dept.name, salary)>  
    <!ELEMENT teaches (IID, course.id)>  
    <!ELEMENT dept.name( #PCDATA )>  
    <!ELEMENT building( #PCDATA )>  
    <!ELEMENT budget( #PCDATA )>  
    <!ELEMENT course.id ( #PCDATA )>  
    <!ELEMENT title ( #PCDATA )>  
    <!ELEMENT credits( #PCDATA )>  
    <!ELEMENT IID( #PCDATA )>  
    <!ELEMENT name( #PCDATA )>  
    <!ELEMENT salary( #PCDATA )>  
]>
```

Figure 283 - Example of a DTD

The **Document Type Definition (DTD)** is the first schema-definition language included as part of the XML standard. It constrains the appearance of subelements and attributes within an element rather than basic types like integers or strings. Each declaration of a DTD is formed as a regular expression for the subelements of an element.

For instance, a university element contains one or more course, department, or instructor elements, and the “|” operator denotes “*or*,” while the “+” operator denotes “*one or more*.”

```
<!ATTLIST course course id CDATA #REQUIRED>
```

```
<!DOCTYPE university-3 [  
    <!ELEMENT university ( (department|course|instructor)+)>  
    <!ELEMENT department ( building, budget )>  
    <!ATTLIST department  
        dept.name ID #REQUIRED >  
    <!ELEMENT course (title, credits )>  
    <!ATTLIST course  
        course.id ID #REQUIRED  
        dept.name IDREF #REQUIRED  
        instructors IDREFS #IMPLIED >  
    <!ELEMENT instructor ( name, salary )>  
    <!ATTLIST instructor  
        IID ID #REQUIRED >  
        dept.name IDREF #REQUIRED >  
    ... declarations for title, credits, building,  
        budget, name and salary ...  
]>
```

Figure 284 - DTD with ID and IDREFS attribute types

- Attributes of XML elements, in turn, have their allowable values declared in the **DTD**.
- Attributes can have a type declaration and a default declaration, with **#REQUIRED**, **#IMPLIED**, or a *default value*.
- The **ID attribute** provides a unique identifier for an element, while the **IDREF attribute** refers to an element by its **ID**. The **IDREFS** attribute allows a list of references, separated by spaces.

```

<university-3>
    <department dept.name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept.name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course.id="CS-101" dept.name="Comp. Sci"
            instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    <course course.id="BIO-301" dept.name="Biology"
            instructors="76766">
        <title> Genetics </title>
        <credits> 4 </credits>
    </course>
    <instructor IID="10101" dept.name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    <instructor IID="83821" dept.name="Comp. Sci.">
        <name> Brandt </name>
        <salary> 72000 </salary>
    </instructor>
    <instructor IID="76766" dept.name="Biology">
        <name> Crick </name>
        <salary> 72000 </salary>
    </instructor>
</university-3>

```

Figure 285 - XML data with ID and IDREF attributes

While the **DTD served** as the first **schema-definition language in XML**, its more recent replacement, **XML Schema**, provides additional features and is more expressive. However, both remain essential tools for constraining and typing information in XML documents.

As an effort to overcome the insufficiencies of the **DTD mechanism**, the **XML Schema** language emerged as a more sophisticated solution. This development brought forth an array of improvements in contrast to **DTDs**. The **XML Schema** introduces several built-in types such as string, integer, decimal, date, and boolean, while also allowing users to define their own types, ranging from simple types with added restrictions to complex types constructed with constructors like complexType and sequence.

The figures depict how the **DTD** in the figure can be represented by the XML Schema, thereby exemplifying the key features of **XML Schema**. Notably, schema definitions in XML Schema are specified in XML syntax themselves, using various tags defined by XML Schema. To avoid any conflicts with user-defined tags, the prefix “**xs:**” is used with the **XML Schema tag**, which is associated with the XML Schema namespace by the “**xmlns:xs**” specification in the root element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept.name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="course">
    <xs:element name="course.id" type="xs:string"/>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="dept.name" type="xs:string"/>
    <xs:element name="credits" type="xs:decimal"/>
</xs:element>
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept.name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 286 - XML Schema version of DTD

The root element “*university*” is specified with a type of “*UniversityType*,” which is declared later on. The types of the “*department*,” “*course*,” “*instructor*,” and “*teaches*” elements are defined by an element with the “**xs:element**” tag, whose body contains the type definition. The “*department*” type is defined as a complex type that consists of a sequence of elements “*dept name*,” “*building*,” and “*budget*.” Any type that has either attributes or nested sub-elements must be specified as a complex type.

```
<xs:attribute name = "dept name"/>
```

```
<xs:element name="teaches">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="IID" type="xs:string"/>
      <xs:element name="course.id" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
  <xs:sequence>
    <xs:element ref="department" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element ref="course" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element ref="instructor" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:element ref="teaches" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figure 287 - Continuation

Alternatively, the type of an element can be specified as a predefined type using the attribute “*type*.” XML Schema types such as “**xs:string**” and “**xs:decimal**” are used to constrain the types of data elements, such as “*dept name*” and “*credits*.” Finally, the example defines the type “*UniversityType*” as containing zero or more occurrences of each of the “*department*,” “*course*,” “*instructor*,” and “*teaches*” elements.

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept name"/>
</xs:key>

<xs: name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept name"/>
</xs:keyref>
```

Attributes are specified using the “**xs:attribute**” tag, which can be used to define the attribute as required or optional. Named complex types can be created using the “**xs:complexType**” element, and we can use the named type to specify the type of an element using the “*type*” attribute.

Furthermore, in addition to defining types, **XML Schema** allows for the specification of constraints. It permits the specification of keys and key references, similar to primary key and foreign key definitions in SQL. A **primary-key constraint** or **unique constraint** in SQL ensures that the attribute values do not recur within the relation. In the context of XML, we must specify a scope within which values are unique and form a key. The selector is a path expression that defines the scope for the constraint, and field declarations specify the elements or attributes that form the key.

To specify that “*dept name*” forms a key for department elements under the root “*university*” element, we add the following constraint specification to the schema definition:

```
<xs:key Copy code <xs:field xpath = "dept name"/>
```

Likewise, a **foreign-key constraint** from “*course*” to “*department*” can be defined in a similar fashion. The XML Schema’s capabilities and flexibility make it a valuable tool for defining complex data structures and managing data in a structured way.

Querying and Transformation

With the increasing use of XML for data exchange, mediation, and storage in various applications, the effective management of XML data has become increasingly vital. In particular, **querying** and **transforming XML data** are essential to extract relevant information from large datasets and convert data between different representations. While the output of a relational query is a relation, the output of an XML query can be an XML document, making querying and transformation an inseparable duo.

In this regard, **XPath** and **XQuery languages** take center stage. **XPath** is a language for path expressions and serves as a fundamental building block for **XQuery**. In contrast, **XQuery** is the go-to language for **querying XML data**, designed with inspiration from **SQL**, yet significantly different due to its ability to handle nested **XML data**. **XQuery** incorporates **XPath** expressions as well.

```
<name>Brandt</name>
```

A tree model of XML data is used in all these languages, with an XML document modeled as a tree where nodes correspond to elements and attributes. Each node, except the root element, has a parent node, which is an element, with the ordering of children of nodes in the tree modeling the order of elements and attributes in the **XML document**. The text content of an element is modeled as a **text-node child** of the element, with elements containing text broken up by intervening sub-elements having **multiple text-node children**.

```
/university-3/instructor  
university-3
```

XPath addresses parts of an **XML document** through path expressions, which can be viewed as an extension of the simple path expressions used in object-oriented and object-relational databases. A path expression is a sequence of location steps separated by “/”, and the result of a path expression is a set of nodes. The **initial ‘/’** indicates the root of the document, which is an abstract root “*above*” the document tag . As a path expression is evaluated, the result of the path at any point consists of an ordered set of nodes from the document, with the result of a path expression being the set of nodes after the last step of path expression evaluation.

The complexity of **XML queries** can be mitigated by utilizing the powerful and versatile XPath language. **With XPath**, accessing attribute values is a breeze, as demonstrated by the syntax of '**/university-3/course/@course id**', which returns a set of all values of course id attributes of course elements.

Selection predicates, contained in square brackets, allow for filtering of results, such as in '**/university-3/course[credits >= 4]**', which returns course elements with a credit value greater than or equal to 4. **XPath** also provides several functions, including **count()** and **id()**, which offer even more functionality for complex queries.

```
/university-3/course[credits >= 4]
/university-3/course[credits >= 4]//@course id
/university-2/instructor[count(./teaches/course) > 2]
/university-3/course/id(@dept name)
/university-3/course/id(@instructors)
/university-3/course[@dept name="Comp. Sci"] |
/university-3/course[@dept name="Biology"]

doc("university.xml")/university/department
```

The union operator, denoted by '|', allows for the combination of query results, while the '**//**' operator can search for elements regardless of their position within the schema. The ability to access data without full knowledge of the schema is a powerful feature of XPath, making it a valuable tool for any developer or database manager. The functions **doc()** and **collection()** provide even greater flexibility, allowing for querying specific documents or collections of documents. With **XPath**, the potential for data retrieval is boundless.

As the World Wide Web continues to evolve, the need for structured data representation and querying has become increasingly apparent. The **World Wide Web Consortium (W3C)** recognized this need and developed *XQuery*, a powerful standard query language for XML. Since its release as a W3C recommendation in 2007, *XQuery* has become an indispensable tool for developers and database administrators alike.

```
for $x in /university-3/course
    let $courseId := $x/@course id
    where $x/credits > 3
        return { $courseId }
```

One of the most distinctive features of **XQuery** is its **FLWOR expressions**. Modeled after SQL queries, these expressions allow developers to easily specify for, let, where, order by, and return clauses, and construct queries with ease. The **Cartesian product** of possible variable values is produced when more than one variable is specified, much like the SQL from clause. Additionally, the let clause allows for the assignment of **XPath expression** results to variable names for simpler representation.

```
for $x in /university-3/course[credits > 3]
    return { $x/@course id }
```

While **XQuery queries** may contain all five clauses, they are by no means necessary for query construction. In fact, queries may only contain for and return clauses, as seen in the simple **XQuery query** above. The let clause may help simplify more complex queries, while the where clause and order by clause are useful for performing additional tests and sorting output, respectively.

XQuery's powerful ability to construct elements using the element and attribute constructors is another key feature.

The element and attribute constructors make it possible to construct elements with a course *ID*, *department name*, *title*, and *credits*, all with just a few lines of code.

```
return element course
{
    attribute course id {$x/@course id},
    attribute dept name {$x/dept name},
    element title {$x/title},
    element credits {$x/credits} }
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
where $c/course id= $t/course id
    and $t/IID = $i/IID
return { $c $i
}
for $c in /university/course,
    $i in /university/instructor,
    $t in /university/teaches
[ $c/course id= $t/course id
    and $t/IID = $i/IID]
return { $c $i }
```

XQuery's ability to join elements is another valuable feature, and it is specified in much the same way as in SQL. **Path expressions** in **XQuery** are identical to those in **XPath2.0** and may return a single value or element, or a sequence of values or elements. In short, XQuery is a powerful language with a multitude of features that make it an indispensable tool for querying **XML data**.

In the realm of **XQuery**, nested queries and sorting of results are essential techniques for generating complex **XML** structures and organizing data output. By nesting **FLWOR** expressions in the return clause, one can create element nestings that do not appear in the source document, as demonstrated in the Figure below. Additionally, **XQuery** provides a variety of aggregate functions like **sum()** and **count()**, as well as the **distinct-values() function**, which returns a sequence without duplicates.

```
<university-1>
{
  for $d in /university/department
  return
    <department>
      { $d/* }
      { for $c in /university/course[dept.name = $d/dept.name]
        return $c }
    </department>
}
{
  for $i in /university/instructor
  return
    <instructor>
      { $i/* }
      { for $c in /university/teaches[IID = $i/IID]
        return $c/course.id }
    </instructor>
}
</university-1>
```

Figure 288 - Creating nested structures in Xquery

```

for $d in /university/department
    return
<department-total-salary>
<dept name>
    { $d/dept name }
</dept name>
<total salary>
{
fn:sum(
    for $i in /university/instructor[dept name = $d/dept name]
    return $i/salary
)
}
</total salary>
</department-total-salary>

```

Although **XQuery lacks** a group by construct, aggregate queries can still be written by utilizing the aggregate functions on path or **FLWOR** expressions nested within the return clause. Take, for example, the query showcased in the university **XML schema**, which calculates the total salary of all instructors in each department. By using the order by clause, results can also be sorted in **XQuery**, as demonstrated in the instructor element example.

```

or $i in /university/instructor
    order by $i/name
    return <instructor>
    { $i/* }
</instructor>
< university-1 >
{
    for $d in /university/department order by $d/dept name
    return

```

```

<department>
{ $d/* }

{ for $c in /university/course[dept name = $d/dept name]
order by $c/course id
return <course> { $c/* }
</course>
</department>
}

</university-1>

```

Furthermore, **XQuery** provides a plethora of built-in functions, ranging from numeric functions to string matching and manipulation functions. It also supports user-defined functions, which can be pre-defined to be associated with **XML Schema** or **XQuery local functions**, as exemplified in the custom function that returns a list of courses offered by a department based on a given instructor identifier.

```

declare function local:dept
courses($iid as xs:string) as element(course)*
{
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept name = $i/dept name]
    return $c
}

```

XQuery is a powerful tool for generating complex XML structures and organizing data output, providing a wide range of functions and techniques to handle complex queries.

XQuery, the **XML query language**, offers a rich set of features that allow for expressive and powerful queries on XML data. One of its key strengths lies in its support for specifying the types of function arguments and return values, which can be leveraged to ensure correctness and improve readability. However, these types of specifications are optional and can be omitted if desired.

```

for $i in /university/instructor[name = "Srinivasan"],
return local:inst dept courses($i/IID)

```

XQuery uses the type system of **XML Schema**, allowing for flexible and fine-grained type declarations. For instance, the type element can match elements with any tag, while the element(course) matches only elements with the tag course. Additionally, types can be suffixed with an asterisk to indicate a sequence of values of that type. For example, the function dept courses specify its return value as a sequence of course elements.

```
some $e in path satisfies P
```

XQuery also offers automatic type conversion, which allows for more natural and concise queries. For instance, if a numeric value represented as a string is compared to a numeric type, **XQuery** will automatically perform the necessary type conversion. Similarly, when an element is passed to a function that expects a string value, **XQuery** will convert the element to a string by concatenating all the text values nested within it.

```
for $d in
    /university/department
where every $i in
    /university/instructor[dept name=$d/dept name]
        satisfies $i/salary > 50000
return $d
and fn:exists
    (/university/instructor[dept name=$d/dept name])
```

Other advanced features of **XQuery** include if-then-else constructs, existential and universal quantification, and the ability to submit queries to an **XML database system** using the **XQJ API**. These features allow for even greater expressiveness and flexibility in **querying XML data**.

Application Program Interfaces to XML

In the realm of XML data manipulation, two **standard APIs** have emerged as popular programming models for parsing and creating in-memory representations of **XML documents**. However, it is important to note that while these APIs excel in handling individual XML documents, they may not be ideal for querying large collections of **XML data**. For such purposes, declarative querying mechanisms like **XPath** and **XQuery** may prove to be more suitable.

The first API, based on the **Document Object Model (DOM)**, presents **XML content** as a tree structure, where each element is represented by a node called **DOMNode**. This API offers a range of interfaces and methods in the *Java API for DOM* that enable navigation of the *DOM tree*, beginning with the root node. Developers can access sub-elements of an element using **getElementsByTagName(name)**, which returns a list of all child elements with a specified tag name, with each member of the list accessible using the **item(i)** method that returns the *i*th element in the list.

In addition, attribute values of an element can be accessed using the method **getAttribute(name)**, and the text value of an element is represented as a **Text node**, which is a child of the element node. **DOM** provides various functions for updating the document by adding and deleting attribute and element children of a node, setting node values, and more.

The second commonly used programming interface, the **Simple API for XML (SAX)**, is built on the concept of event handlers and is designed to provide a common interface between parsers and applications. This API allows developers to create handler functions for each parsing event, which are then called with parameters describing the event as it occurs. The handler functions to carry out tasks such as constructing a tree representing the **XML data**, adding nodes to a partially constructed tree, and setting the parent of a node as the current node for attaching child nodes.

While **DOM** is suitable for accessing **XML data** stored in databases and building an **XML** database with **DOM** as its primary interface for accessing and modifying data, it lacks support for any form of declarative querying. On the other hand, **SAX** requires more programming effort than **DOM** but avoids the overhead of creating a **DOM** tree in situations where the application needs to create its own data representation.

Storage of XML Data

In the realm of database management, storing **XML data** has become a crucial task for many applications. There are various methods available to store **XML data**, including document storage in file systems or through special-purpose databases. Alternatively, **XML data** can be converted to a relational form and stored in a relational database, which is a widely used method.

```
nodes(id, parent id, type, label, value)
      department(id, dept name, building, budget)
          course(parent id, course id, dept name, title, credits)
```

However, converting **XML data** to relational form can be challenging, particularly when the data is not generated from a relational schema. Moreover, elements that recur and nested elements can further complicate the storage of **XML data** in relational form. In such cases, alternative approaches, such as storing **XML data** as a string or using a tree representation, can be adopted.

```
<university>
    <department>
        <row>
            <dept.name> Comp. Sci. </dept.name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </row>
        <row>
            <dept.name> Biology </dept.name>
            <building> Watson </building>
            <budget> 90000 </budget>
        </row>
    </department>
    <course>
        <row>
            <course.id> CS-101 </course.id>
            <title> Intro. to Computer Science </title>
            <dept.name> Comp. Sci </dept.name>
            <credits> 4 </credits>
        </row>
        <row>
            <course.id> BIO-301 </course.id>
            <title> Genetics </title>
            <dept.name> Biology </dept.name>
            <credits> 4 </credits>
        </row>
    </course>
</university>
```

Figure 289 - SQL/XML representation of (part of) university information

While the former approach has some limitations, such as the lack of direct querying capabilities and the need to scan all tuples, the latter approach offers a more organized way of storing XML data as a tree and utilizing a relational database. By storing each element and attribute with its identifier in the “*nodes*” relation, arbitrary **XML data** can be modeled as a tree and efficiently stored.

```
select xmlelement (name "course",
    xmlattributes (course id as course id, dept name as dept
    name),
    xmlelement (name "title", title),
    xmlelement (name "credits", credits))
from course

select xmlelement (name "department",
    dept name,
    xmlagg (xmlforest(course id)
    order by course id))
from course
group by dept name
```

However, implementing XML data storage and querying on top of the relational abstraction is a complex task that requires supporting features such as transactions, security, and data access from clients. Hence, using an existing database system is recommended to avoid building a full-featured database system from scratch.

XML Applications

In a world where structured data storage is paramount, **XML** is emerging as the go-to solution for storing and communicating data and accessing web services. As traditional textual representations fall short in capturing the nuances of complex, multi-valued data, **XML-based formats** offer a more efficient and reliable solution.

From user preferences to editable document representation, XML is now widely used to represent data with complex structures. Document representation standards such as **Open Document Format (ODF)** and **Office Open XML (OOXML)** are now based on XML, allowing for easier and more efficient data storage and exchange.

The benefits of XML are not only limited to data storage and representation but also extend to standardized data exchange formats across a range of specialized applications. In industries such as shipping and the chemical industry, **XML-based standards** have been developed to represent complex data such as molecular structure and shipment records, respectively. Even online marketplaces can benefit from XML, with RosettaNet standards defining **XML schemas** and **semantics for representing data**, as well as standards for message exchange.

In addition to data storage and exchange, XML has proven to be invaluable in web services, where limited forms of information need to be provided through predefined interfaces. XML-based procedures allow providers to communicate input and output information through the **HTTP protocol**, with the **Simple Object Access Protocol (SOAP)** providing a standard for invoking procedures and representing the procedure input and output.

9 ADVANCED TOPICS

This chapter focuses on application development and discusses various techniques to improve application performance, including performance tuning and standard benchmarks. It also covers issues related to application testing and migration and provides an overview of existing database-language standards. Followed by a chapter that describes **spatial** and **temporal data types**, **multimedia data**, and the challenges associated with storing them in databases. It also addresses mobile computing systems and related database issues. In addition, covers advanced transaction-processing techniques, including **transaction-processing monitors** and **transactional workflows**. It also explores main-memory database systems, real-time transaction systems, and long-duration transactions.

9.1 Advanced Application Development

In the realm of advanced application development, ensuring that an application runs at optimal speed is of utmost importance. In previous chapters, we learned how to design and build an application, but the challenge does not end there. It is not uncommon for an application to run slower than expected or to handle fewer transactions than required, which can lead to user dissatisfaction or even render the application unusable. This is where performance tuning comes in, which involves identifying and eliminating bottlenecks and upgrading hardware to achieve faster processing.

To assess the performance of a database system, standardized benchmarks are used. These sets of tasks allow developers to estimate the **hardware** and **software requirements** of an application even before it is built. However, testing an application during development is equally essential. This involves generating database states and test inputs, and verifying that the outputs match the expected outputs. In this chapter, we delve into the intricacies of application testing.

Legacy systems, though outdated and based on older-generation technology, are still prevalent in many organizations and run mission-critical applications. Interfacing with and migrating away from these systems can pose significant challenges, and we discuss these issues in detail. Furthermore, with the rise of the Internet, communication between applications is more crucial than ever, making standards an essential aspect of application development. We explore the various standards proposed for database-application development.

Performance Tuning

As we venture into the realm of database systems, we are met with the perennial challenge of tuning performance. This crucial task involves a multitude of intricate adjustments and design choices that are implemented to optimize the performance of a system for a specific application.

The design choices affecting the performance of a database system are diverse, ranging from fundamental aspects like schema and transaction design to technical parameters such as buffer sizes, and even hardware considerations, such as the number of disks.

```
select sum(salary)
  from instructor
 where dept name=?
```

One key strategy for improving performance in SQL queries executed by an application program is through enhancing set orientation. Often, a query is executed repeatedly with different values for a parameter, leading to unnecessary communication overheads and processing overheads at the server.

```
select dept name,
       sum(salary)  from instructor
      group by dept name;
```

Consider a program that sifts through various departments, invoking an embedded SQL query to find the total salary of all instructors in the department. If the instructor relation does not have a clustered index on the dept name, each query will entail a scan of the relation. Even if there is an index, a **random I/O operation** will be necessary for each dept name value.

```
PreparedStatement pStmt = conn.prepareStatement(
    "insert into instructor values(?, ?, ?, ?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setInt(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.addBatch();
pStmt.setString(1, "88878");
pStmt.setString(2, "Thierry");
pStmt.setInt(3, "Physics");
pStmt.setInt(4, 100000);
pStmt.addBatch(); pStmt.executeBatch();
```

Figure 290 - Batch update in JDBC

Instead, a single SQL query can be utilized to find the total salary expenses of each department. By combining multiple SQL queries into one, the execution costs can be greatly reduced, especially for larger databases with a significant number of departments.

Another technique for improving set orientation is through nested subquery decorrelation. Advanced optimizers can transform even poorly written queries and execute them efficiently. However, complex queries with nested subqueries remain a challenge for many optimizers.

```
update department set budget = budget +  
    (select amount  
     from funds received  
     where funds received.dept name = department.dept name)  
     where exists(  
         select *  
         from funds received  
         where funds received.dept name = department.dept name);
```

For bulk loads and updates, database systems provide a bulk import utility that allows for the reading of data from a file and efficient performance of integrity constraint checks and index maintenance. A bulk export utility is also available for **common input** and **output file formats** such as comma-separated values or tab-separated values formats.

Thus, the task of tuning performance in database systems demands careful consideration of numerous aspects, with strategic design choices and a keen understanding of the underlying mechanisms of the system being key.

As we delve deeper into the intricacies of database management systems, we uncover the location of bottlenecks - *the crux of a system's performance*. The pursuit of performance optimization is the pursuit of bottleneck elimination, achieved through an improvement in the performance of the components causing them. By focusing on the weak links in a system, we can see significant improvements in overall speed, as opposed to investing resources in already optimized regions.

```
merge into department as A
    using (select *
            from funds received) as F
        on (A.dept name =F.dept name)
when matched then
    update set budget = budget + F.amount;
when not matched then
    insert values
        (F.dept name, null, F.budget)
```

However, database systems prove more complex than their program counterparts, as they can be modeled as queueing systems. With each transaction requesting various services, from disk reads to **CPU cycles**, each service has a corresponding queue, with waiting times often exponential in relation to their utilization.

As such, a system should aim to maintain low utilization to avoid lengthy queues and delays.

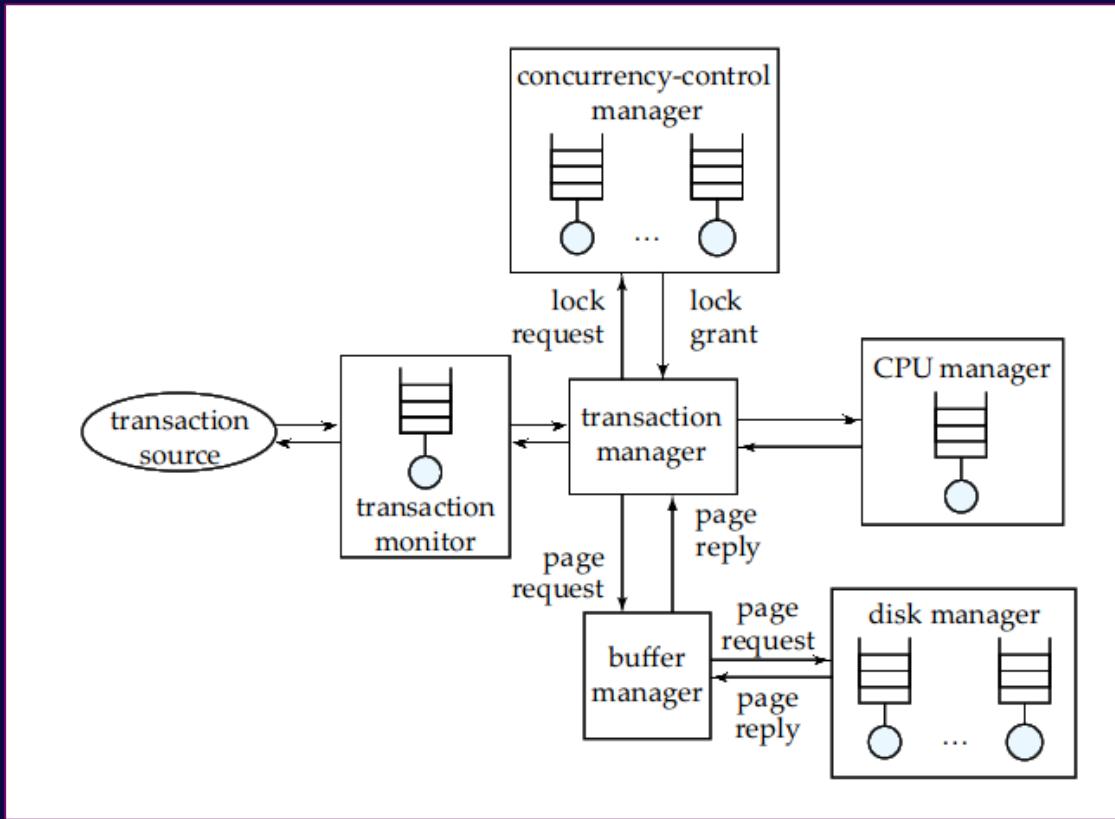


Figure 291 - Queues in a database system

To achieve optimal performance, database administrators can tune a system at three levels - *hardware*, *database management*, and *application*. Each level offers various options for improvement, from optimizing hardware components to adjusting database management parameters to improving application design. As we continue to advance in our understanding of database systems, we must strive for balance, seeking to eliminate bottlenecks and utilize all components to their fullest potential.

Amidst the intricate world of database tuning lies the question of when to store data in memory, and when to persist it on disk. It is a delicate balancing act, one that requires a keen understanding of system design and the underlying hardware. The answer lies in the frequency of data access, a crucial metric that determines the optimal placement of data.

(price per megabyte of memory) / (pages per megabyte of memory)

$$n * \frac{\text{price per disk drive}}{\text{access per second per disk}} = \frac{\text{price per megabyte of memory}}{\text{pages per megabyte of memory}}$$

Figure 292 - Formula

This is where the **5-minute rule** comes into play, a heuristic that advises storing frequently accessed data in memory, and relegating infrequently accessed data to disk. The rule stipulates that if a page is used more frequently than once in **5 minutes**, it should be cached in memory. However, with current technological advancements, the rule has evolved, and now recommends caching all pages that are accessed at least once in **2 hours on average**.

The rule also takes into account the amount of data read or written per **I/O operation**, as well as the type of access (*random or sequential*). For instance, the 1-minute rule applies to sequentially accessed data, advising that data should be cached in memory if they are used at least once in **1 minute**. Interestingly, this rule has remained relatively constant over the years, even as disk transfer rates have increased greatly.

With the advent of flash memory, system designers now have the option to store frequently used data in flash storage, instead of on disk. The **flash-as-buffer** approach, which uses flash storage as a persistent buffer, is another viable option. However, these approaches require changes to the database system, and may not be suitable for all applications.

In addition to placement optimization, choosing the right **RAID implementation** is also crucial to system performance. The decision to use **RAID 1** or **RAID 5** depends on the frequency of data updates and random writes. For many applications, **RAID 1** is a better option, as it requires fewer **I/O** operations per second.

Ultimately, the art of database tuning requires a holistic approach, one that considers not only the schema, but also the hardware and the needs of the application. Only by carefully balancing all these factors can one hope to achieve optimal system performance.

In the world of database management, automated tuning of physical design and concurrent transactions is a matter of utmost importance. Commercial database systems have evolved to provide tools that assist database administrators in the **index** and **materialized** view selection and other design-related tasks. These tools examine the workload and suggest views to be materialized, allowing the database administrator to prioritize queries to be optimized. **Microsoft's Database Tuning Assistant** is a leading example of such a tool, allowing users to estimate the effect of materializing a view on the total workload cost.

Generating a workload is a crucial step in the process, and it is usually accomplished by recording all the queries and updates executed during a set period. Compression techniques are then used to represent the workload using a small number of queries and updates. The most expensive queries are chosen to be addressed first, with greedy heuristics used to estimate the benefits of materializing different indices or views.

Concurrent execution of different types of transactions can lead to poor performance due to contention on locks. This issue can be addressed through snapshot isolation, which allows queries to be executed on a snapshot of the data while updates can proceed concurrently. If snapshot isolation is not available, weaker levels of consistency such as **read-committed isolation** can be employed. However, this can lead to issues with dirty reads and phantom reads.

Physical design and **concurrent transaction tuning** are crucial for the optimal performance of a database management system. It is essential to have reliable tools and techniques to optimize the workload and minimize contention on locks, ensuring that the system runs efficiently and effectively.

Performance Benchmarks

The quest for faster and more efficient database servers has been the holy grail of software systems, with vendors striving to outdo one another in a bid to offer the best performance. However, as software systems become more complex, different vendors' implementations vary, leading to significant performance variations across different tasks. As a result, measuring a system's performance through a single task is insufficient. Instead, suites of standardized tasks, called performance benchmarks, are used to quantify a system's performance accurately.

The need to measure performance accurately requires careful computation of the performance numbers from multiple tasks, as simple performance measures may be misleading. For instance, if two systems have different transaction types, computing their average throughput per transaction type and averaging the results to gauge system performance is misleading. To obtain accurate results, it is necessary to take the time to completion for the workload, rather than the average throughput for each transaction type.

$$\frac{n}{\frac{1}{t_1} + \frac{1}{t_2} + \dots + \frac{1}{t_n}}$$

Figure 293 - Formula

Online transaction processing (OLTP) and decision support, including **online analytical processing (OLAP)**, are two broad classes of applications handled by database systems. These two classes of tasks have different requirements. Hence, the architecture of some database systems has been tuned to transaction processing, while others have been tuned to decision support. Other vendors try to strike a balance between the two tasks. However, applications usually have a mixture of transaction-processing and decision-support requirements. Therefore, determining the best database system for an application depends on the mix of the two requirements.

To address these challenges, the **Transaction Processing Performance Council (TPC)** has defined a series of benchmark standards for database systems. The **TPC benchmarks** are defined in great detail, including the set of relations, the sizes of the tuples, and the number of tuples in the relations. The performance metric is throughput, expressed as **transactions per second (TPS)**. Furthermore, the **TPC benchmark measures performance** in terms of price per **TPS**, as cost is of great importance for business applications. However, obtaining **TPC** benchmark numbers for systems requires an external audit that ensures that the system faithfully follows the definition of the benchmark, including full support for the **ACID properties** of transactions.

Amidst the fast-paced world of technology, the **TPC-H benchmark stands out** as a refinement of its predecessor, the **TPC-D benchmark**. Its schema may remain the same, but **TPC-H boasts** of **22** intricate queries, of which **16** are derived from **TPC-D**. This benchmark embodies the essence of *ad hoc querying*, where the queries are not known beforehand, hence preventing the creation of appropriate materialized views in advance. Materialized views and other redundant information are strictly prohibited, with indices permitted solely on primary and foreign keys.

In a bid to gauge performance, **TPC-H** employs two tests.

- The power test involves running the queries and updates one at a time sequentially, with the measure of queries per hour derived from dividing **3600 seconds** by the geometric mean of the execution times of the queries (in seconds).
- The throughput test, on the other hand, runs multiple streams in parallel, with each stream executing all **22** queries, accompanied by a parallel update stream. The number of queries per hour is computed by using the total time for the entire run.

Notably, the composite query per hour metric provides an overall measure, obtained by computing the square root of the product of the power and throughput metrics. In determining a composite price/performance metric, the system price is divided by the composite metric.

Beyond **TPC-H** lies the **TPC-W Web commerce benchmark**, which models websites that feature static and dynamic content. Caching of dynamic content is allowed, given its effectiveness in speeding up webpages. This benchmark centers around an electronic bookstore, with different scale factors factored in. **Web interactions per second (WIPS)** and **price per WIPS** serve as the primary performance metrics, highlighting the effectiveness of this benchmark.

Although **TPC-W** is no longer in use, these benchmarks continue to shape the tech landscape, providing crucial insights into the world of technology and data analytics.

Other Issues in Application Development

In the realm of application development, two prominent issues have emerged: **testing** and **migration**. Testing, a critical aspect of application development, is a continuous process that involves designing a comprehensive test suite that encompasses all test cases. This becomes particularly challenging when one is dealing with database applications, as **subtle bugs** and **errors** can often go unnoticed. Thus, to ensure that the test suite is effective, it is imperative to create test databases that can catch commonly occurring errors.

In the context of migration, the legacy systems pose a significant challenge. These systems, which may be several decades old and employ outdated technologies, may still contain valuable data and support critical applications. However, replacing these systems with newer ones is an onerous task that requires porting vast amounts of data and retraining a large number of staff. To avoid such a massive undertaking, organizations may attempt to interoperate the legacy systems with newer ones. One such approach is to build a wrapper layer on top of the legacy system that can make it appear to be a relational database, thereby enabling interoperability between relational databases and legacy databases.

The challenges of application development continue to grow and evolve, and it is crucial for organizations to stay abreast of these changes to remain competitive in the ever-changing landscape of technology.

9.2 Spatial and Temporal Data and Mobility

Throughout the storied history of databases, the data types typically stored in these repositories were relatively straightforward, with earlier versions of SQL reflecting this limited scope. However, as time progressed, a mounting demand for the handling of more intricate data types, including temporal, spatial, and multimedia data, has emerged.

Compounding these evolving data requirements is the proliferation of mobile computing devices, from humble laptops and handheld organizers to smartphones equipped with built-in computers and a plethora of wearable technologies increasingly used in commercial settings. This trend has given rise to an array of issues, further complicating the storage and management of data.

Within the context of this ever-changing landscape, this chapter delves into several data types and other salient database issues germane to these cutting-edge applications.

Motivation

As the world becomes increasingly digital, so too does the need for more complex data types in databases. The history of databases reflects the evolution of technology, with earlier versions of SQL offering limited support for data types. However, the growing demand for handling spatial, temporal, and multimedia data types has ushered in a new era of database requirements.

Temporal data, which stores information about past states, is critical for many applications, but can be a cumbersome task to incorporate into a schema design. The solution lies in database support for temporal data.

Spatial data, including **geographic** and **computer-aided-design data**, have traditionally been stored as files in a file system. However, the increasing complexity and volume of data, coupled with a growing number of users, have necessitated the use of database systems. These systems offer the ability to store and query large amounts of data efficiently, along with other features such as atomic **updates** and **concurrency control**, which are essential for spatial-data applications.

Multimedia data, such as *image*, *video*, and *audio data*, require database systems with specialized features due to their continuous-media nature. Retrieval at a steady, predetermined rate is crucial for displaying video and audio data, making them challenging to store and manage.

Mobile databases, which cater to the needs of mobile computing systems, such as laptops and high-end cell phones, have unique requirements. These systems must be able to operate while disconnected from the network, necessitating special memory management techniques.

Time in Databases

In the realm of database systems, time plays a crucial role in modeling the states of real-world phenomena. While most databases capture the current state of the real world, they often lose crucial information about past states, which may be critical for various applications. This is where temporal databases come into the picture.

Temporal databases store information about states of the real world across time, which helps to provide comprehensive insights into various phenomena, including patients' medical histories, factory sensor readings, and other such data that change with time.

ID	name	dept.name	salary	from	to
10101	Srinivasan	Comp. Sci.	61000	2007/1/1	2007/12/31
10101	Srinivasan	Comp. Sci.	65000	2008/1/1	2008/12/31
12121	Wu	Finance	82000	2005/1/1	2006/12/31
12121	Wu	Finance	87000	2007/1/1	2007/12/31
12121	Wu	Finance	90000	2008/1/1	2008/12/31
98345	Kim	Elec. Eng.	80000	2005/1/1	2008/12/31

Figure 294 - A temporal instructor relation

However, when it comes to the issue of time in database systems, we must differentiate between the time measured by the system and the time observed in the real world. This leads us to the concept of "**valid time**" and "**transaction time**," which define the set of time intervals during which a fact is true in the real world and the time interval during which the fact is current within the database system, respectively.

A temporal relation refers to a database relation where each tuple has an associated time when it is true, and the time may be either valid time or transaction time. A temporal relation that stores both valid time and transaction time is a bitemporal relation.

To represent intervals, we have the "**Interval**" structured type, which contains the '**from**' and '**to**' fields. While some tuples may have a "*" in the '**to**' time column, indicating that the tuple is true at the current time, the times are stored internally in a more compact form for ease of representation.

SQL standard defines various types like date, time, timestamp, time with timezone, and timestamp with timezone to represent time. The SQL interval type allows us to refer to a period of time without specifying a particular time when this period starts.

In terms of temporal query languages, a temporal selection involves the time attributes, while a temporal projection inherits the times of tuples from the original relation. A temporal join is an operation where the time of a tuple in the result is the intersection of the times of the tuples from which it is derived.

The various temporal predicates like precedes, overlaps, and contains can be applied to intervals to provide valuable insights into various real-world phenomena. Overall, temporal databases play a crucial role in helping us understand and model the real world across time, enabling us to analyze various data and make informed decisions.

Spatial and Geographic Data

As the world grows more complex, so too do our data needs. One area where this complexity is most apparent is in the storage, indexing, and querying of spatial and geographic data. These types of data require specialized structures that can handle their unique properties, such as their spatial relationships and topological characteristics.

The importance of spatial data support in databases cannot be overstated. Consider the challenge of storing a set of polygons and querying the database to find all polygons that intersect a given polygon. Standard index structures, like **B-trees** or **hash indices**, are insufficient for this task. To efficiently process this type of query, special-purpose index structures like **R-trees** are required.

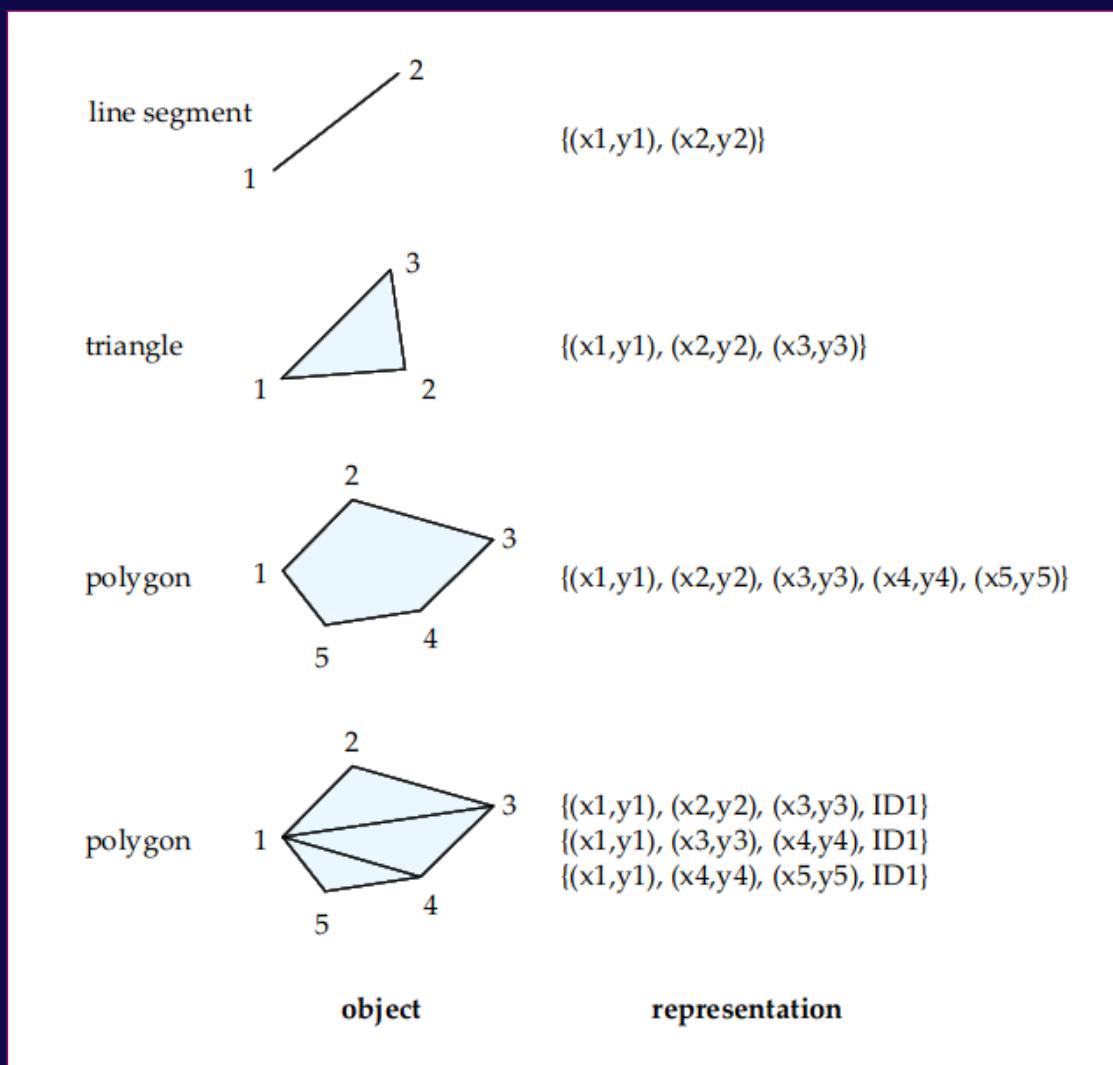


Figure 295 - Representation of geometric constructs

There are two types of spatial data that are particularly important: **computer-aided-design (CAD)** data and geographic data. **CAD data** provides spatial information about how objects are constructed, while geographic data includes road maps, land-usage maps, topographic elevation maps, and political maps, to name a few.

Geographic information systems are specialized databases designed for storing geographic data. Many database systems, such as **IBM DB2 Spatial Extender**, the **Informix Spatial Datablade**, and **Oracle Spatial**, have added support for geographic data.

Geometric information can be represented in several different ways, but a normalized representation is ideal. A line segment, for example, can be represented by the coordinates of its endpoints, while a polyline consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints. Polygons can be represented by listing their vertices in order or by dividing them into a set of triangles. Circles and ellipses can be represented by corresponding types or by approximating them with polygons.

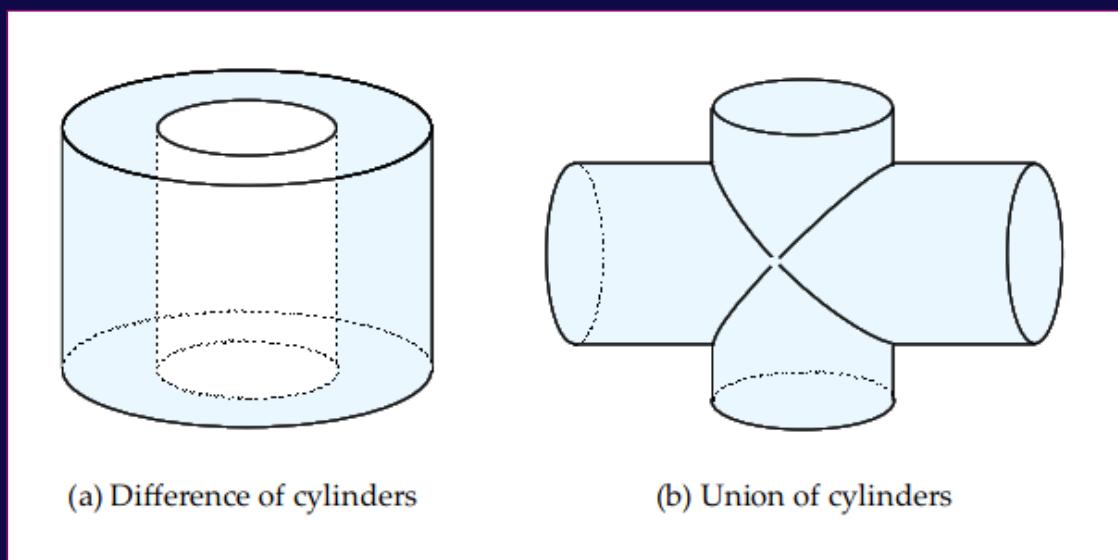


Figure 296 - Complex three-dimensional objects

Design databases pose unique challenges, particularly for large-scale integrated circuits or the design of entire airplanes. **Object-oriented databases** represent components of the design as objects, and the connections between the objects indicate how the design is structured. The objects stored in a design database are generally geometric objects, from simple **two-dimensional objects** like points, lines, and polygons to complex three-dimensional objects formed by union, intersection, and difference operations. **Three-dimensional** surfaces may also be represented by wireframe models, which essentially model the surface as a set of simpler objects, such as line segments, triangles, and rectangles.

Geographic information systems typically incorporate both raster and vector data for the effective display of information. A layered display of information is common, with each layer of information superimposed upon the previous layer in bottom-to-top order. Interestingly, even vector data may be converted to raster form to ensure compatibility with certain web browsers, while preventing unauthorized extraction of data.

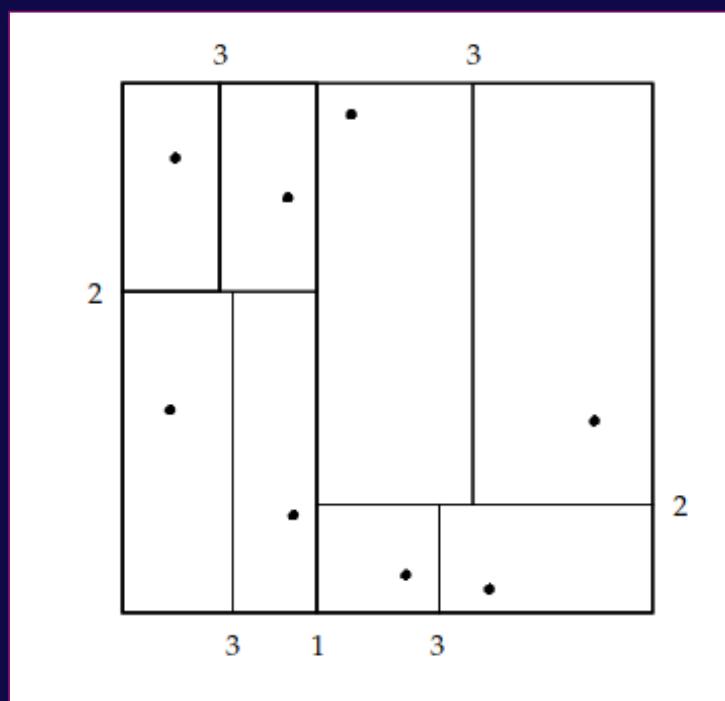


Figure 297 - Division of space by a k-d tree

Queries on **geospatial data** may be classified according to their spatial or non-spatial requirements. Spatial queries involve objects or regions in close proximity to a given point or region or objects that intersect with one another. Join algorithms play a crucial role in the spatial querying process, with several techniques proposed for the efficient computation of spatial joins on vector data.

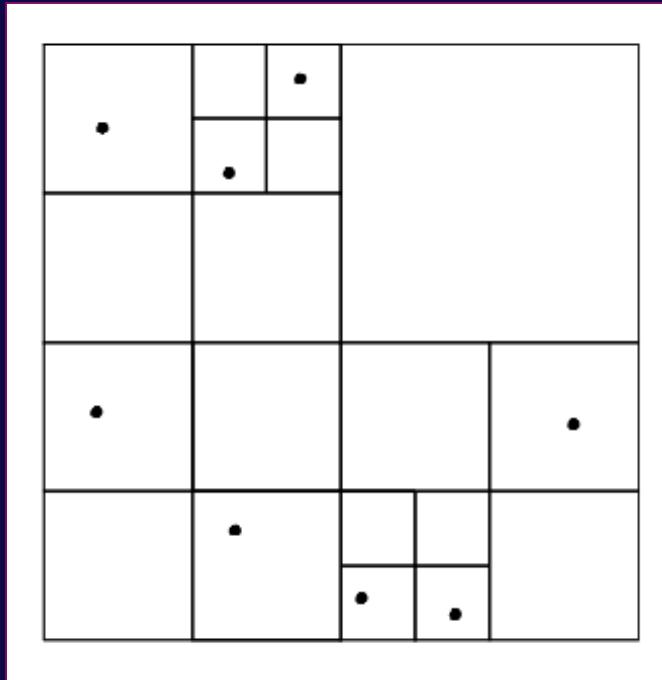


Figure 298 - Division of space by a quadtree

Since spatial data are inherently **graphical**, **graphical query languages** provide a **natural** means for querying such data. Results of such queries are typically displayed graphically, with users able to invoke a wide range of operations on the interface. Efficient indexing of spatial data is another crucial aspect of geospatial data analysis, with traditional index structures such as hash indices and **B-trees** being unsuitable for **multi-dimensional data**. **K-d trees** are a popular alternative indexing technique for efficiently searching multi-dimensional data.

The **R-tree** stands tall as a powerful tool for **efficient search**, **insert**, and **delete** operations. Its distinctive feature lies in the fact that sibling nodes may overlap, unlike other popular structures like **B+-trees**, **k-d trees**, and **quadtrees**, whose ranges do not overlap.

This creates a unique challenge for object searches and queries for objects that intersect a given object, requiring a thorough exploration of multiple paths.

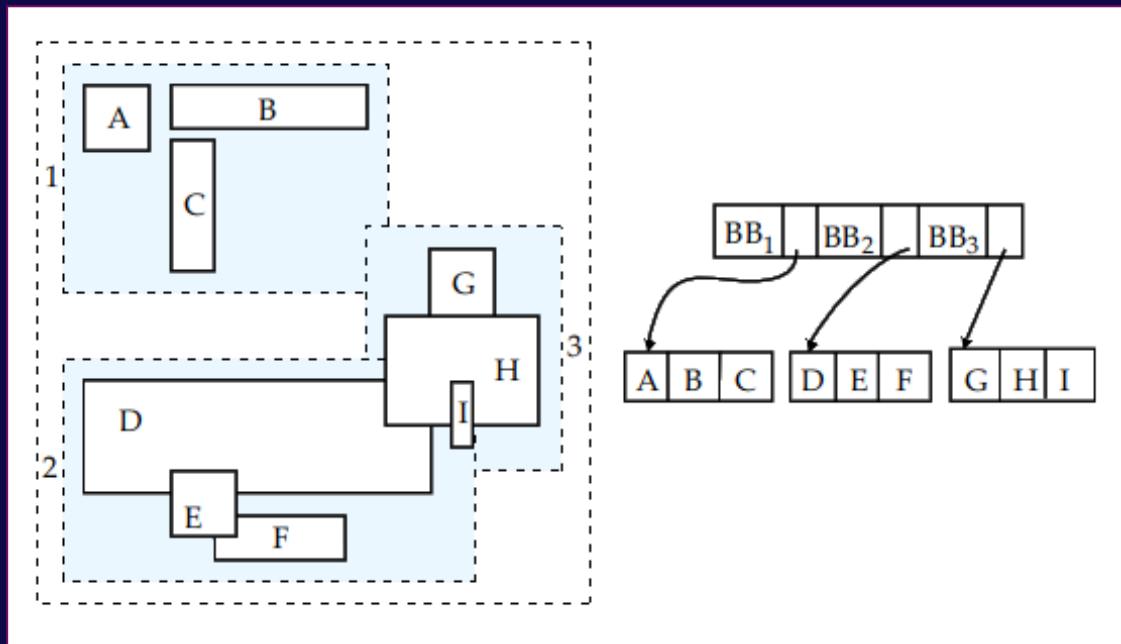


Figure 299 - An R-tree

Inserting an object into an **R-tree** requires a leaf node to hold the object. While picking a leaf node that has enough space and contains the bounding box of the object is ideal, this may not always be feasible, and searching for such a node could be very expensive. As such, the **R-tree algorithm** chooses a heuristic approach, selecting an arbitrary child node with a bounding box containing the bounding box of the object for continuing the traversal. This ensures that the tree remains balanced and the bounding boxes of internal and leaf nodes remain consistent.

The insertion procedure is slightly different from that of **B+-trees**, relying on **heuristics to split nodes**. The quadratic split heuristic, for instance, selects a pair of entries with the maximum wasted space and iteratively adds the remaining entries to one of the two sets in a manner that minimizes the total area of their bounding boxes or their overlap.

Deletion in R-trees is performed akin to **B+-tree deletion**, either borrowing entries from sibling nodes or merging them when a node becomes under-filled. An alternative approach is to redistribute all entries of under-filled nodes to sibling nodes to improve clustering.

Multimedia Databases

In the rapidly evolving world of data storage and management, multimedia databases present a unique challenge. As the popularity of multimedia data, including images, audio, and video, continues to rise, traditional file systems may no longer suffice for managing large collections of multimedia objects.

Database features such as **transactional updates**, **querying facilities**, and **indexing** becomes increasingly important when managing large collections of multimedia objects. However, storing multimedia objects outside the database can complicate matters and lead to inconsistencies, making it desirable to store the data themselves in the database.

To successfully store multimedia data in a database, several issues must be addressed. The database must support large objects, as multimedia data such as videos can occupy up to several gigabytes of storage. Furthermore, retrieving some types of data, such as audio and video, requires a guaranteed steady rate of data delivery.

Multimedia databases also require similarity-based retrieval for many applications, necessitating the creation of special index structures for this purpose. Finally, multimedia data must be stored and transmitted in compressed form due to the large number of bytes required to represent multimedia data.

Multimedia databases represent a unique challenge in the world of data storage and management. To successfully manage large collections of multimedia objects, databases must support large objects, guarantee steady data delivery rates, provide similarity-based retrieval capabilities, and store and transmit multimedia data in compressed form.

Mobility and Personal Databases

In the ever-evolving technological landscape, mobile computing has emerged as a game-changer, challenging traditional notions of centralized control and administration in database applications. The ubiquity of mobile devices, ranging from laptops and notebooks to cell phones with computer-like capabilities, has transformed the way data is accessed and stored.

Fuelled by a **low-cost wireless digital communication infrastructure**, based on wireless local-area networks, cellular digital packet networks, and other cutting-edge technologies, mobile computing has found myriad applications across industries. For instance, business travelers can work and access data while on the move, delivery services can track packages, and emergency response services can access vital information in real-time.

However, this technological shift has given rise to unique challenges that demand novel solutions. One such challenge is the need for energy efficiency, as battery power is a scarce resource for most mobile devices. To mitigate this issue, mobile devices spend most of their time in sleep mode, waking up intermittently to check for incoming or outgoing data.

Moreover, in this decentralized ecosystem, where **users administer their machines**, data consistency becomes a concern. As users employ multiple devices, they require up-to-date access to their data, regardless of the device they are using at a given time. Cloud computing, a variant of which is often used to address this issue, has emerged as a preferred solution.

In light of these challenges, new techniques are being developed to tackle mobility and personal computing issues. This includes the development of a model of mobile computing, consisting of mobile hosts, wired networks of computers, and mobile support stations. The emergence of Bluetooth, wireless LANs, and packet-based cellular telephony networks, including **2.5G, 3G, 4G** and **5G**, has added another dimension to the database application domain, generating immense databases that require real-time access.

As mobile computing continues to evolve, it is essential to stay abreast of the latest developments and adapt to the rapidly changing technological landscape. Only then can we harness the full potential of mobile computing to revolutionize the way we access, store, and process data.

The version-vector scheme, a novel solution to the persistent problem of dealing with failures in distributed file systems, has garnered widespread attention due to its versatile applications in modern computing. With the proliferation of mobile computing and the increasing prevalence of disconnected systems, the version-vector scheme has proven invaluable in handling the complexities of distributed file systems. Additionally, its usefulness extends to groupware systems, where hosts are intermittently connected, requiring the exchange of updated documents.

In replicated databases, the version-vector scheme is particularly effective when applied to individual tuples, such as calendar entries or contacts. By utilizing this scheme, the reconciliation of inconsistent copies of data can be handled seamlessly, without any risk of conflict. However, the version-vector scheme falls short in addressing the most challenging aspect of shared data updates, namely the reconciliation of inconsistent copies. While many applications can automatically perform reconciliation by executing update operations on remote computers, this solution may not work if update operations do not commute.

Despite its many strengths, the version-vector scheme requires significant communication between a reconnecting mobile host and its support station, which can delay consistency checks until data are needed, potentially increasing overall database inconsistency. Furthermore, the practicality of transaction-processing techniques for distributed systems is limited by the potential for disconnection and the cost of wireless communication.

Therefore, in many cases, it is preferable to let users prepare transactions on mobile hosts but require that they submit transactions to a server for execution. Transactions that span multiple computers and include a mobile host face long-term blocking during transaction commit, unless disconnections are rare or predictable. These limitations and challenges highlight the need for continued research in this critical area of modern computing.

9.3 Advanced Transaction Processing

The chapter of this seminal work delves into the cutting-edge world of advanced transaction processing, taking the reader on a journey beyond the fundamental concepts introduced in earlier chapters. Here, the authors present an array of sophisticated techniques designed to ensure that transactions retain the revered ACID properties even in complex and failure-prone environments, where concurrency is a constant consideration.

Among the topics explored in this chapter are **transaction-processing monitors**, transactional workflows, and the challenges of conducting transactions in the rapidly evolving world of elec

onic commerce. The authors also delve into the intricacies of **main-memory databases**, **real-time databases**, **long-duration transactions**, and **nested transactions**. These are no small feats to accomplish, and the authors do so with the finesse of seasoned experts, guiding the reader through this complex and vital terrain with the utmost care and attention to detail.

Through their masterful presentation of these advanced techniques, the authors demonstrate their deep expertise and commitment to advancing the field of transaction processing. This chapter is a tour de force, and a must-read for anyone seeking to understand the intricacies of ensuring transactional integrity in today's fast-paced and ever-changing digital landscape.

Transaction-Processing Monitors

In the fast-paced world of transaction processing, one architecture has stood the test of time: the transaction-processing monitor (*TP monitor*). First developed in the 1970s and 1980s to support a large number of remote terminals, TP monitors have evolved to become the core support system for distributed transaction processing.

At the forefront of this technology is **IBM's CICS TP monitor**, one of the earliest and most widely used TP monitors. Other TP monitors like Oracle Tuxedo and Microsoft Transaction Server have also found a place in the market. However, with the rise of web application server architectures, TP monitors have faced some tough competition. These servers support many of the features of TP monitors and are often referred to as "*TP lite*."

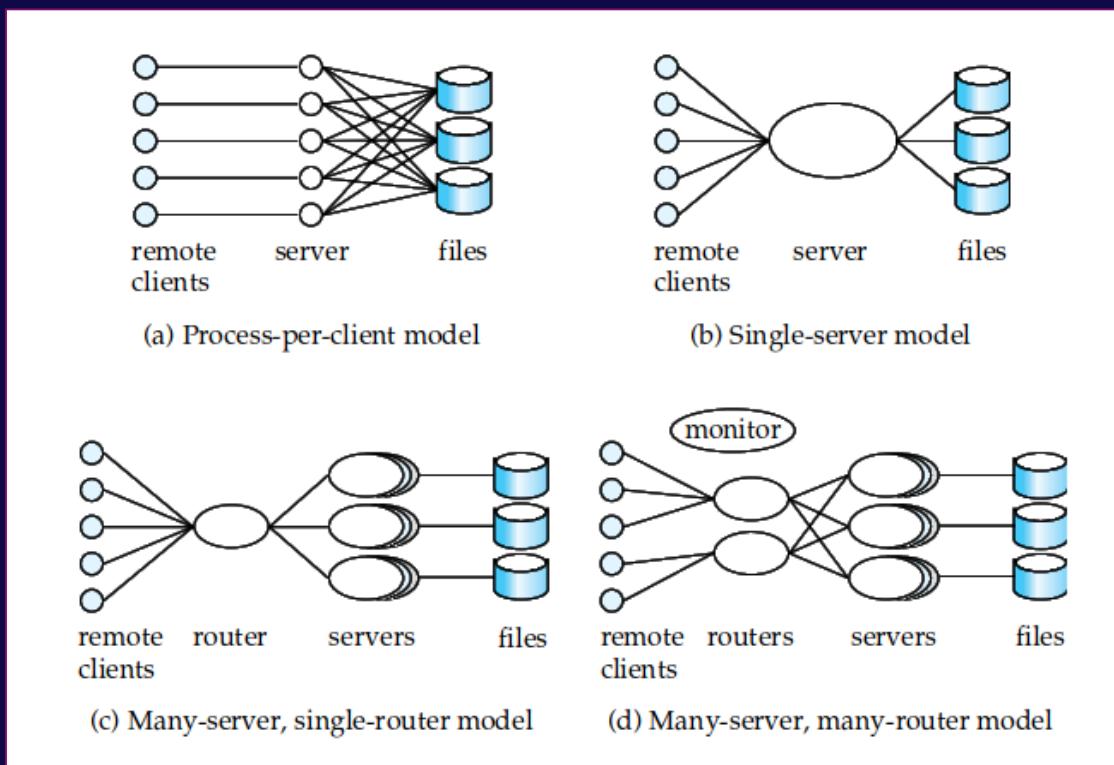


Figure 300 - TP-monitor architectures

Despite this, the concepts underlying TP monitors remain the same. **Large-scale transaction-processing systems** rely on a **client-server architecture**, which can either have a server process for each client or a single-server process to which all remote clients connect. The latter model presents some unique benefits, including low overhead and a multithreaded server process that can execute its own low-overhead multitasking.

However, the **single-server model** also presents its own set of problems, especially when multiple applications access the same database. One solution to this is the **many-server, single-router model**, which runs **multiple application-server** processes that access a common database and lets the clients communicate with the application through a single communication process that routes requests.

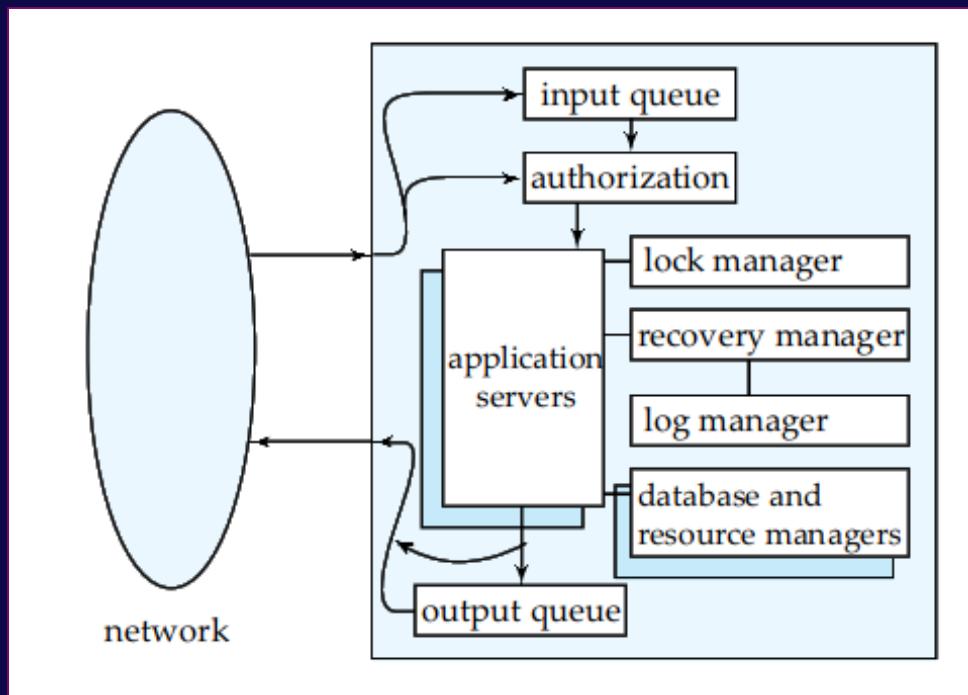


Figure 301 - TP-monitor components

A more general architecture, the many-server, many-router model, has multiple processes to communicate with clients. The client communication processes interact with one or more router processes, which route their requests to the appropriate server. Very high-performance web-server systems also adopt such an architecture.

Despite its challenges, the transaction-processing industry has continued to grow and innovate. And with the rise of distributed data and the need for parallel processing, TP monitors and their many variations will continue to play a critical role in the world of transaction processing.

TP monitors treat each subsystem as a resource manager, offering transactional access to a set of resources. The interface between the TP monitor and the resource manager is defined by a set of transaction primitives, such as begin a transaction, committing the transaction, aborting the transaction, and preparing to commit the transaction (for a two-phase commit). In addition, the resource manager must also provide other services such as data supply to the application. The X/Open Distributed Transaction Processing standard defines the resource-manager interface, which many database systems support and can act as resource managers for.

TP monitors can also act as coordinators of two-phase commit for transactions accessing services provided by them, such as persistent messaging and durable queues, and for database systems. This ensures that all actions are carried out or none at all, regardless of any failures. In complex client-server systems with multiple servers and a large number of clients, TP monitors administer system checkpoints and shutdowns, provide security and authentication of clients, administer server pools, and control the scope of failures.

TP monitors can also be used to hide database failures in replicated systems, such as remote backup systems. Transaction requests are sent to the TP monitor, which relays them to one of the database replicas, and if one site fails, the TP monitor can transparently route messages to a backup site, masking the failure of the first site.

Finally, TP monitor systems provide a transactional Remote Procedure Call (RPC) interface to their services. This enables the enclosing of a series of RPC calls within a transaction, allowing updates to be performed within the scope of the transaction and rolled back if there is any failure. The use of TP monitors in this way is a critical part of modern large-scale distributed systems.

Transactional Workflows

In an age where the automation of workflows has become the norm, the importance of transactional workflows cannot be overstated. As a result, organizations are increasingly finding themselves requiring software systems that can coordinate multiple tasks executed by different processing entities. These entities can be either human or automated, with the execution of each task depending on the successful completion of the preceding task.

The complexity of these workflows is not only a result of the coordination of multiple entities but also because of the existence of **multiple independently managed information-processing systems**. This is further complicated by the fact that these systems were developed separately to automate different functions. Workflows that involve interactions among several such systems, each performing a task, as well as interactions with humans, require transactional workflows that use and extend the concepts of transactions to the context of workflows.

Workflow application	Typical task	Typical processing entity
electronic-mail routing	electronic-mail message	mailers
loan processing	form processing	humans, application software
purchase-order processing	form processing	humans, application software, DBMSs

Figure 302 - TP-monitor components

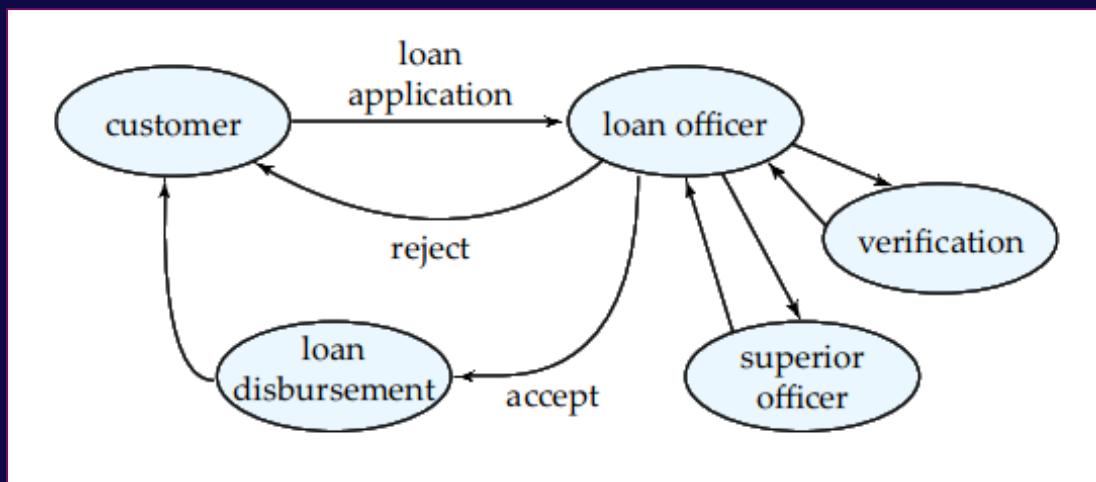


Figure 304 - Workflow in loan processing

To automate a workflow, two activities must be addressed:

- Workflow specification
- Workflow execution

Workflow specification involves detailing the tasks that must be carried out and defining the execution requirements. On the other hand, **workflow execution** involves ensuring computation correctness, data integrity, and durability, providing the safeguards of traditional database systems.

In recent years, a number of workflow systems have been developed to address these challenges. However, the properties of workflow systems are studied at a relatively abstract level, without going into the details of any particular system. The abstraction of the workflow system is such that the internal aspects of a task do not need to be modeled for the purpose of specification and management of a workflow.

Thus, in today's world, the **automation of workflows is critical**, and the use of transactional workflows that extend the concepts of transactions to the context of workflows is imperative. With multiple systems involved, each performing a task, as well as interactions with humans, the management of workflows requires the utmost care and attention to ensure that data integrity and computation correctness are maintained. The execution of complex tasks often involves the **coordination of multiple**, interdependent activities known as workflows. The proper management and recovery of these workflows are of utmost importance to ensure their atomicity and reliability.

To achieve this, recovery procedures must be implemented to guarantee that a workflow will ultimately reach an acceptable termination state, regardless of any failures that may occur within its processing components. This includes the ability to recover the **execution-environment context** and restore the state information of the scheduler, as well as to manage message queues to ensure proper task execution.

Workflows are commonly **hand-coded within application systems**, but the advent of workflow-management systems has brought a new level of simplicity and reliability to their construction and execution. These systems allow workflows to be specified in a **high-level manner** and **executed** accordingly, with a growing number of commercial options available. In today's interconnected world, workflows that cross organizational boundaries are becoming increasingly prevalent, making it vital for these systems to be able to interoperate to minimize human intervention. This is where business process management comes in, enabling the orchestration of process logic to control workflows by invoking services provided by multiple applications within a **service-oriented architecture (SOA)**.

The use of standards such as the **Web Services Business Process Execution Language (WS-BPEL)**, **Business Process Modeling Notation (BPMN)**, and **XML Process Definition Language (XPDL)** provides a common framework for specifying and executing workflows based on web services and business processes. Leading examples of business process management systems based on the SOA architecture include *Microsoft's BizTalk Server*, *IBM's WebSphere Business Integration Server Foundation*, and *BEA's WebLogic Process Edition*, among others. As the world becomes increasingly interconnected and complex, the proper management and execution of workflows will only continue to grow in importance.

E-Commerce

In today's fast-paced digital world, **e-commerce** has revolutionized the way we conduct business. Through the use of electronic means, primarily the Internet, various commerce-related activities can be carried out with great ease and convenience. These include **presale activities, negotiations on price and quality of service, payment for the sale, delivery of the product or service, and customer support and post-sale service.**

The backbone of e-commerce lies in the extensive use of databases to support these activities. With customized **e-catalogs**, customers can easily browse and search for products that cater to their specific needs. This requires not only an intuitive hierarchical organization of products but also keyword search facilities to facilitate the search process. Furthermore, e-catalogs can be personalized based on the **customer's past buying history, providing them with tailored offerings and special discounts.**

When there are multiple buyers and sellers for a product, a marketplace such as a stock exchange helps to negotiate the price to be paid for the product. The challenges in handling marketplaces lie in ensuring the secure authentication of bidders, recording secure buy or sell bids in a database, and quickly communicating the bids to all relevant parties.

After selecting products and determining the price, the order has to be settled, which involves payment for the goods and the delivery of the goods. However, **payment settlement** electronically has its own set of challenges, including credit card fraud and trusting the seller to bill only for the agreed-on item. Therefore, it is imperative to ensure secure payment settlement through various means such as third-party payment gateways and digital wallets.

E-commerce has opened up a world of opportunities and convenience for consumers and businesses alike. By utilizing databases and customized e-catalogs, securely handling marketplaces, and ensuring secure payment settlement, e-commerce has transformed the traditional commerce landscape and will continue to do so in the future.

Main-Memory Databases

The demand for high-speed transaction processing has become a pressing need for many organizations. To meet this demand, databases must be able to **handle hundreds or even thousands of transactions per second**. However, relying solely on high-performance hardware and parallelism is insufficient to achieve very low response times, as disk **I/O** remains a bottleneck.

The issue with disk **I/O** lies in the fact that **each I/O operation** requires approximately **10 milliseconds** to complete, a number that has not decreased at a rate comparable to the increase in processor speeds. This not only increases the time required to access a data item but also limits the number of accesses per second. To address this issue, the size of the database buffer can be increased to make the database system less disk-bound. The recent advancements in main-memory technology have allowed for the construction of large main memories at a relatively **low cost, enabling faster processing of transactions by keeping data memory resident**. However, disk-related limitations still persist, as log records must be written to stable storage before a transaction is committed.

To overcome this issue, one can create a stable log buffer in the main memory using **non-volatile RAM**, which would reduce commit time and minimize overhead. Additionally, the group-commit technique can be implemented to reduce the number of output operations per committed transaction. Although group commits result in a slight delay in the commit of transactions that perform updates, this delay can be made quite small and is acceptable for many applications.

As data sizes continue to grow at a rapid pace, more applications are expected to have data that fit into main memory. This provides opportunities for optimizations, such as designing internal data structures in main-memory databases to reduce space requirements and minimizing space overhead for **query-processing techniques**. Furthermore, recovery algorithms can be optimized, as pages rarely need to be written out to make space for other pages. While the use of high-performance hardware and parallelism can improve transaction processing, the issue of disk **I/O bottleneck** must be addressed to achieve low response times. Through the use of large main memories and techniques such as stable log buffers and group commits, database systems can provide faster processing of transactions while minimizing overhead and optimizing recovery algorithms.

Real-Time Transaction Systems

Real-time transaction systems pose a unique challenge in the world of database management. While traditional integrity constraints focus solely on maintaining the accuracy and consistency of data stored within the database, real-time systems must also adhere to strict deadlines. Whether it be managing a plant, controlling traffic, or creating a schedule, missed deadlines can lead to catastrophic consequences, making it imperative that these constraints are taken seriously.

There are three types of deadlines to consider: **hard**, **firm**, and **soft**. Hard deadlines are non-negotiable and must be met to avoid severe consequences such as system crashes. Firm deadlines, on the other hand, have zero value if missed after the deadline has passed. Soft deadlines represent a sliding scale, where the value of a task diminishes as the degree of lateness increases. These deadlines can be both tricky to manage and highly consequential in nature.

Transaction management in real-time systems requires a delicate balance. If a transaction must wait, it may cause the system to miss a deadline, leading to dire outcomes. However, preempting a transaction holding a lock can also be risky, as the **time lost due to rollback** and **restart** could cause the transaction to miss its deadline anyway. Main-memory databases are often used to manage real-time constraints, as disk accesses can take orders of magnitude more time than main-memory references, making execution time difficult to estimate accurately.

Even with main-memory databases, variances in execution time arise from lock waits and transaction aborts, among other factors. Extensive research has been devoted to concurrency control for real-time databases, with optimistic concurrency protocols performing exceptionally well. These protocols result in fewer missed deadlines than even the extended locking protocols.

Designing a **real-time system** involves striking a delicate balance between processing power and deadline management, all while navigating the variability inherent in transaction management. While the challenges are many, the rewards for the successful implementation of such systems are immeasurable.

Long-Duration Transactions

In a world where database systems involve human interaction, the traditional transaction concept faces serious challenges. These long-duration transactions exhibit properties such as exposure of uncommitted data, subtasks, recoverability, and performance, making it impractical to enforce the requirement for only serializable schedules. Each of the **concurrency-control protocols**, such as **two-phase locking**, **graph-based protocols**, **timestamp-based protocols**, and **validation protocols**, has adverse effects on these long-duration transactions, leading to long waiting times, transaction aborts, or both.

T_1	T_2
<code>read(A)</code> $A := A - 50$ <code>write(A)</code>	<code>read(B)</code> $B := B - 10$ <code>write(B)</code>
<code>read(B)</code> $B := B + 50$ <code>write(B)</code>	<code>read(A)</code> $A := A + 10$ <code>write(A)</code>

Figure 305 - A non-conflict-serializable schedule

The **enforcement of transaction atomicity** is a critical issue that increases the probability of long-duration waits or creates a possibility of cascading rollback, particularly for long-duration transactions. While snapshot isolation and optimistic concurrency control without read validation protocol can provide a partial solution to these issues, conflicting updates are more likely with long-duration transactions, resulting in additional complexity.

The challenges posed by **long-duration transactions** in database systems with human interaction require a rethinking of the traditional transaction concept and concurrency-control protocols to accommodate the unique properties of these transactions. Only then can we ensure **fast response times**, **predictability**, and **user satisfaction** while **maintaining transaction atomicity** and **consistency**.

A nuanced set of techniques has emerged to mitigate the risks of long-duration transactions. These techniques, characterized by multilevel **transactions** and **compensating transactions**, offer a potent solution to complex, high-concurrency systems.

Multilevel transactions, known colloquially as sagas, enable the release of locks upon subtransaction completion. This results in the creation of higher-level operations that enhance concurrency. The intricacy of this approach is exemplified through a simple example of increment and decrement operations in transactions **T1** and **T2**. These subtransactions may be executed in any order, generating the correct result.

However, the exposure of uncommitted data that occurs through **multilevel transactions** may create cascading rollbacks. This is where compensating transactions enter the fray. After subtransaction completion and lock release, a compensating transaction may be executed to undo the effect of a subtransaction.

The implementation of such techniques is not without its difficulties, however. For long-duration transactions to survive system crashes, a comprehensive restoration process is required, including the logging of all internal system data such as lock tables and transaction timestamps. Despite these **implementation challenges**, **multilevel** and **compensating transactions** offer a robust solution to the complexities of high-concurrency systems.

10 CASE STUDIES

The upcoming section shall explicate the ways in which disparate database systems integrate the plethora of concepts that have been elaborated upon in the preceding chapters of this book. We embark on this journey by delving into the depths of PostgreSQL - **a widely used open-source database system.**

In the subsequent chapters, we endeavor to investigate three of the most commonly employed commercial database systems - *IBM DB2*, *Oracle*, and *Microsoft SQL Server*.

Each of these chapters sheds light on the distinctive features of the corresponding database systems, including their tools, SQL variations and extensions, and system architecture. These features encompass a wide array of characteristics such as storage organization, query processing, concurrency control and recovery, and replication.

It is essential to note that the aforementioned chapters only provide a glimpse into the key aspects of the database products under consideration, and thus, must not be considered as an exhaustive coverage of the products. As the products are subject to periodic updates and upgrades, the details provided in these chapters may become obsolete. Hence, when working with a particular product version, it is imperative to refer to the user manuals for specific details.

Furthermore, it is important to bear in mind that the chapters in this section utilize industrial jargon rather than academic terminology. As such, the readers shall encounter terms such as '*table*' instead of '*relation*', '*row*' instead of '*tuple*', and '*column*' instead of '*attribute*'.

10.1 PostgreSQL

PostgreSQL, the open-source object-relational database management system, has been making waves in the tech world for its robust features and versatility. Developed as a descendant of the **POSTGRES** system by *Professor Michael Stonebraker* at the *University of California, Berkeley*, **PostgreSQL** is named after the pioneering relational database system Ingres, which was also developed under Stonebraker's guidance. With support for many aspects of *SQL:2003*, **PostgreSQL** offers a range of advanced features, including complex queries, foreign keys, triggers, views, transactional integrity, full-text searching, and limited data replication. Furthermore, users can extend **PostgreSQL** with new data types, functions, operators, or index methods, and it supports a wide variety of programming languages, including *C*, *C++*, *Java*, *Perl*, *Tcl*, and *Python*, as well as database interfaces such as **JDBC** and **ODBC**.

PostgreSQL has earned its place as one of the two most widely used open-source relational database systems, alongside **MySQL**. One of the most notable aspects of **PostgreSQL** is its permissive **BSD license**, which allows anyone to use, modify, and distribute its code and documentation for any purpose without a fee. As such, it has become a popular choice for businesses and developers who value flexibility and cost-effectiveness.

Introduction

In the annals of database management systems, few have undergone as many transformations as **PostgreSQL**. Developed from the original **POSTGRES** system by *Professor Michael Stonebraker* at *UC Berkeley* in the late *1980s*, **PostgreSQL** has evolved into a fully-fledged open-source object-relational database management system, supporting numerous features such as complex queries, foreign keys, triggers, views, and transactional integrity.

Over the years, **PostgreSQL** has gone through several major releases, with the initial version introduced to users in *1989*. The developers of the system have focused on enhancing portability and performance, with the addition of an SQL language interpreter in *1994* marking a significant milestone. The system has since undergone several iterations, with the latest version boasting native support for **Microsoft Windows**.

As a testament to its capabilities, **PostgreSQL** is widely used in research and production applications, including the PostGIS system for geographic information, and is taught as an educational tool in universities. With contributions from a community of over **1000** developers, **PostgreSQL** continues to evolve and improve.

We provide a comprehensive overview of **PostgreSQL**, delving into the system's user interfaces, languages, data structures, and concurrency-control mechanisms. Whether you're a beginner or an expert, this chapter is an excellent resource for understanding one of the most robust open-source relational database systems available.

User Interfaces

PostgreSQL, the popular open-source database management system, offers an array of user interfaces for administration and programming tasks. Among them are command-line tools for administering the database, with the interactive terminal client `psql` serving as the main interface. **Psql** provides advanced features such as variable substitution and SQL interpolation while leveraging the **popular GNU readline library for command-line editing**. Additionally, PostgreSQL supports a Tcl/Tk shell for scripting purposes, which requires the `pgtcl` library to be loaded.

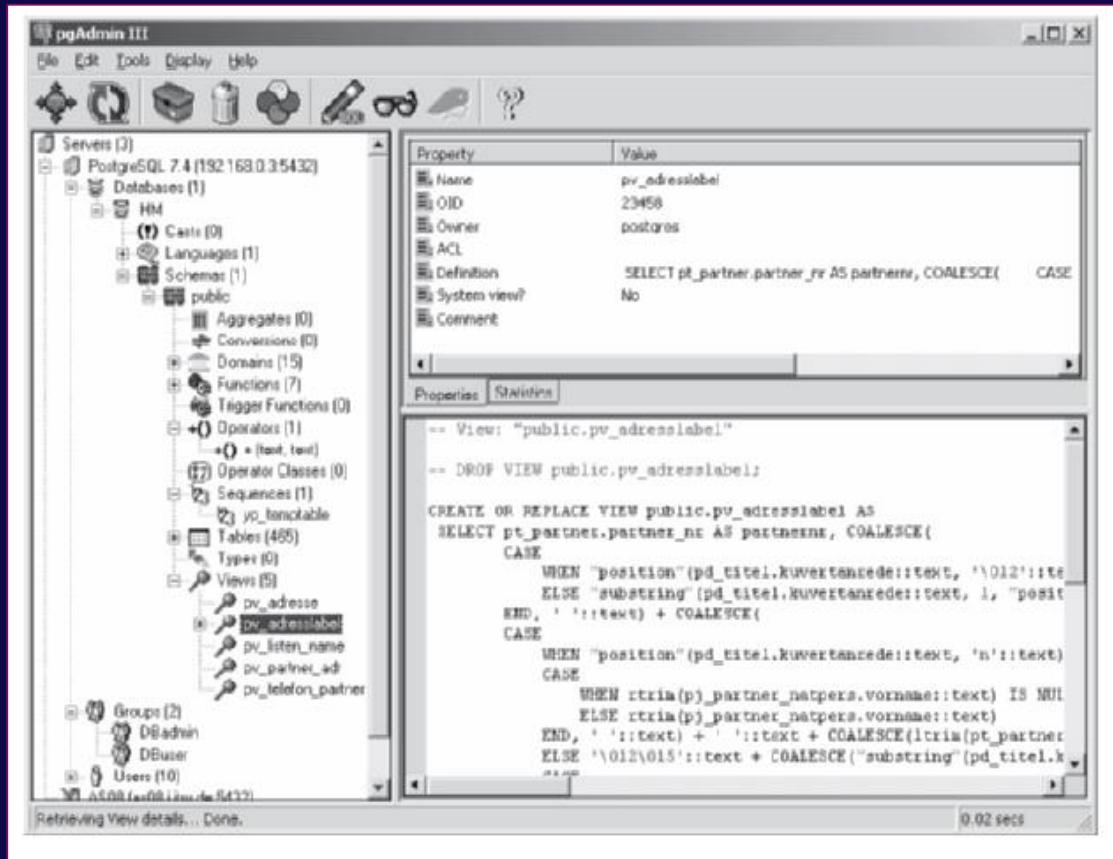


Figure 306 - pgAdmin III: An open-source database administration GUI

On the graphical front, **PostgreSQL** lacks any built-in tools for administration or design, but a plethora of open-source and commercial **GUI tools** exist for users to choose from. Some popular options for administration include **pgAccess** and **pgAdmin**, while **TORA** and **Data Architect** are commonly used for database design. For those looking for form-design and report-generation tools, commercial and open-source options include **Rekall**, **GNU Report Generator**, and **GNU Enterprise**.

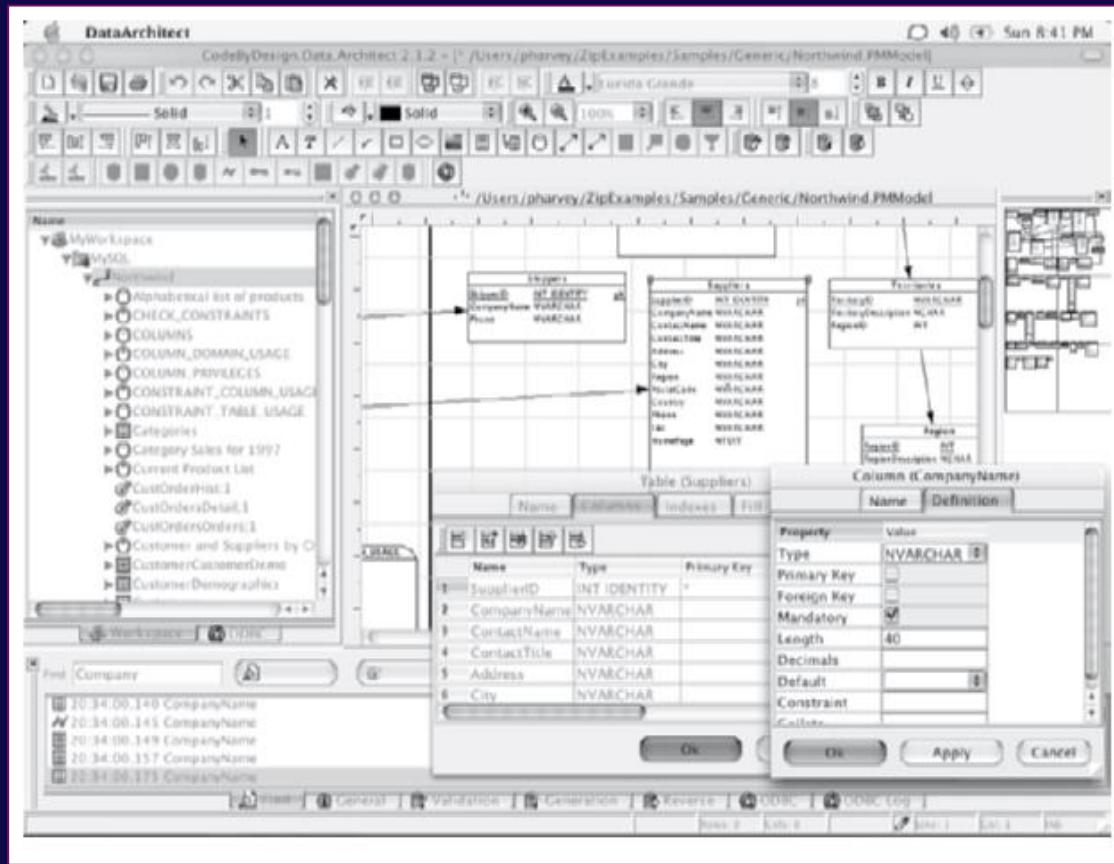


Figure 307 - Data Architect: A multiplatform database design GUI

As for programming interfaces, **PostgreSQL** provides a host of options, with native interfaces for ODBC and JDBC, and bindings for a wide range of programming languages such as *C*, *C++*, *PHP*, *Perl*, *Tcl/Tk*, *ECPG*, *Python*, and *Ruby*. The **C API** for **PostgreSQL** is provided by the *libpq library*, which also serves as the underlying engine for most programming-language bindings. *Libpq* supports **synchronous** and **asynchronous** execution of **SQL commands** and prepared statements, with a reentrant and thread-safe interface that allows for flexible configuration of connection parameters.

SQL Variations and Extensions

In the world of database management systems, **PostgreSQL** reigns supreme as a flexible and versatile option. This powerful tool boasts support for numerous **SQL** features, including some object-relational features that are not found in other systems. Furthermore, **PostgreSQL** offers support for a variety of non-standard types, including geometric data types, enumerated types, and full-text searching capabilities.

At the heart of **PostgreSQL's type system** lies its base types, which are implemented in a low-level language such as C. These types encapsulate both state and operations and can represent scalar values or variable-length arrays. Additionally, composite types are automatically generated whenever a table is created, and domains can be defined by coupling a base type with a constraint.

```
create rule rule name as on
    { select | insert | update | delete }
    to table [ where rule qualification ]
    do [ instead ] { nothing | command | ( command ; command ... ) }

create view myview as select * from mytab;
create table myview (same column list as mytab);
create rule return as on select to myview do instead
    select * from mytab;
```

One of **PostgreSQL's strengths** lies in its support for *non-standard types*, which are included in the standard distribution. Geometric data types are used in geographic information systems to represent two-dimensional spatial objects, while full-text searching is performed using the tsvector and tsquery types.

Furthermore, **PostgreSQL offers** data types to store network addresses, making it a viable option for network-management applications.

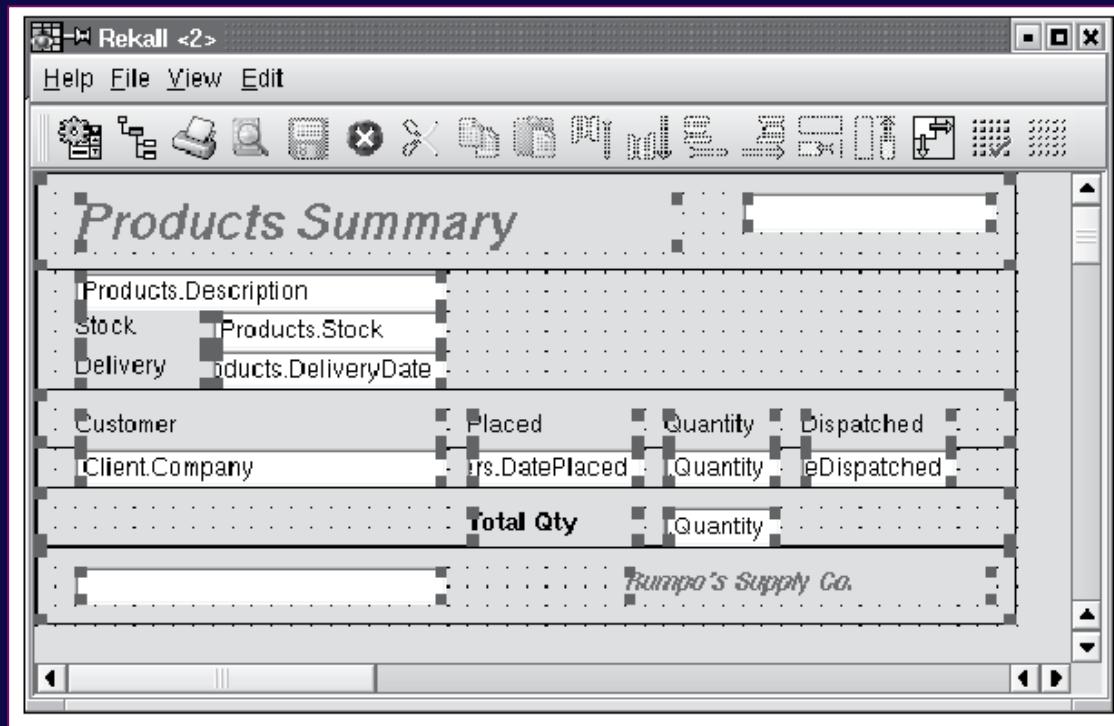


Figure 308 - Rekall: Report-design GUI

PostgreSQL offers a wide range of types to meet the needs of a variety of application domains. Its flexibility and versatility have earned it a place at the forefront of the database management system landscape.

In the realm of **PostgreSQL database management**, queries on views undergo a transformation before execution to queries on the underlying tables they represent. The recommended method for creating views is through the create view syntax, which not only offers greater concision but also ensures the prevention of recursive view creation - a potential result of carelessly declared rules that can lead to **runtime errors**.

Although rules lack the ability to explicitly define update actions on views, they do offer a means to audibly log increases in instructor salaries through the creation of an update rule.

Similarly, **insert/update rules** enable the approval of salary increases stored in a table, granting permission for the simultaneous update of salaries across an entire instructor table.

```
create rule salary audit as on update to instructor
    where new.salary <> old.salary
        do insert into salary audit
            values (current timestamp, current user,
                    new.name, old.salary, new.salary);

create rule approved increases insert
    as on insert to approved increases
        do instead
            update instructor
                set salary = salary + new.increase
                where name = new.name;
```

While the functionality of rules and per-row triggers overlap, **PostgreSQL's rule system** is able to implement most *triggers*, *foreign keys*, and other constraints that remain outside the scope of rules. Triggers possess the added benefit of generating error messages that signal constraint violations, something a rule cannot do. Conversely, rules provide an update or delete action for views, while triggers cannot be used with view relations, as the trigger would never be called.

```
insert into approved increases select * from salary increases;
```

An important difference between triggers and views lies in their execution. While triggers execute iteratively for every affected row, rules manipulate the query tree before query planning, rendering them more efficient than triggers in situations where a statement affects numerous rows.

```
create type city t as
    (name varchar(80), state char(2));
create type status t as enum
    ('alpha', 'beta', 'release');
```

Moreover, **PostgreSQL** is designed with extensibility in mind, with much of its system catalogs storing information on data types, functions, access methods, and more. This storage of additional information enables **PostgreSQL** to incorporate user-written code into the server via the dynamic loading of shared objects.

The contrib module of the **PostgreSQL distribution** contains a vast array of user functions, base types, and index extensions, making it easier for users to extend the system's capabilities. Users can even define composite and enumeration types or create new base types, further demonstrating **PostgreSQL's flexibility** and **expandability**.

In the realm of PostgreSQL, the order of listed names bears a significant weight when comparing values of an enumerated type. This fact can be put to good use in a statement that fetches the names of products that have passed the alpha stage, such as "**select name from products where status > 'alpha'**". If one desires to add base types to PostgreSQL, the process is relatively straightforward, as evidenced in the *complex.sql* and *complex.c* files featured in the **PostgreSQL tutorials**. The initial step involves declaring the base type in C, similar to the "*Complex*" example below:

```
double x;  
double y;  
} Complex;
```

After that, one must establish functions that will read and write values of the new type in text format. Subsequently, the new type can be registered with the "*create type complex*" statement, which designates details such as *internallength*, *input*, *output*, and *alignment*. If the text **I/O** functions have been registered as "*complex in*" and "*complex out*," then one may opt to define **binary I/O functions** for more efficient data dumping. Ultimately, extension types are just like existing base types of PostgreSQL, except that they are dynamically loaded and linked to the server. Fortunately, indices may be readily extended to handle new base types.

PostgreSQL provides users with the capacity to create functions that are stored and executed on the server. Function overloading is also supported, allowing functions to be declared using the same name but with arguments of varying types. For those interested in adding functions written in C, PostgreSQL has an application programmer interface. The coding conventions for user-defined C functions, which are dynamically loaded, and internal functions that are statically linked to the server, are essentially identical. Therefore, the standard internal function library serves as an excellent source of coding examples for user-defined C functions.

Once the shared library containing the function has been established, a declaration such as the following registers it on the server:

```
returns cstring as 'shared object filename' language C immutable strict;
```

In this case, the entry point to the shared object file is assumed to be the same as the **SQL function name** (*here, complex out*) unless otherwise specified. The *C code* for the text output function of complex values is straightforward due to the application program interface, which hides most of **PostgreSQL's internal details**.

In PostgreSQL, the creation of new aggregate functions and the extension of existing index structures is made simple by the use of **user-defined functions** and **operator classes**.

To define a new aggregate function, two functions must be declared: the first function declares the complex output, and the subsequent functions implement the output. The code utilizes **PostgreSQL-specific constructs**, such as the `palloc` function, which allows dynamic memory allocation controlled by PostgreSQL's memory manager. The addition of a new aggregate function requires the definition of the state transition function and, optionally, a final function to compute the return value.

```
create aggregate sum
(
    sfunc = complex add,
    basetype = complex,
    stype = complex,
    initcond = '(0,0)' );
```

PostgreSQL's support for various index methods includes B-tree and hash indices, as well as **unique-to-PostgreSQL index methods**, such as the **Generalized Search Tree (GiST)** and the **Generalized Inverted Index (GIN)**. Indexing for two-dimensional spatial objects is also available with the R-tree index, which is implemented using a GiST index behind the scenes.

Adding index extensions for a specific type requires the definition of an operator class encapsulating index-method strategies and support routines.

```
create operator
class complex abs ops
    default for type complex using btree as
        operator 1 < (complex, complex),
        operator 2 <= (complex, complex),
        operator 3 = (complex, complex),
        operator 4 >= (complex, complex),
        operator 5 > (complex, complex),
    function 1 complex abs
        cmp(complex, complex);
```

Stored functions and procedures in PostgreSQL can be written in a number of procedural languages, including **PL/pgSQL**, **PL/Tcl**, **PL/Perl**, and **PL/Python**. The server programming interface provides a range of functions to manipulate PostgreSQL data, and supports the **C programming language**.

Transaction Management in PostgreSQL

In the realm of transaction management, PostgreSQL employs a hybrid approach utilizing both snapshot isolation and two-phase locking protocols. The choice of the protocol employed is dependent on the type of statement being executed. For data manipulation language (DML) statements, PostgreSQL uses the snapshot isolation scheme, commonly referred to as the multi-version concurrency control (MVCC) scheme. Meanwhile, concurrency control for data definition language (DDL) statements is based on the standard two-phase locking approach.

<i>Isolated level</i>	<i>Dirty Read</i>	<i>Non repeatable Read</i>	<i>Phantom Read</i>
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeated Read	No	No	Maybe
Serializable	No	No	No

Figure 309 - Definition of the four standard SQL isolation levels

PostgreSQL offers different isolation levels, including three weak consistency levels, in addition to the serializable level of consistency. The weak consistency levels allow for a higher degree of concurrency for applications that do not require strong guarantees of serializability. These consistency levels are defined in terms of three phenomena that violate serializability: dirty read, nonrepeatable read, and phantom read. **PostgreSQL** supports two of the four isolation levels specified in the **SQL standard**, namely read committed and serializable.

However, the implementation of the serializable isolation level uses snapshot isolation, which does not ensure true serializability.

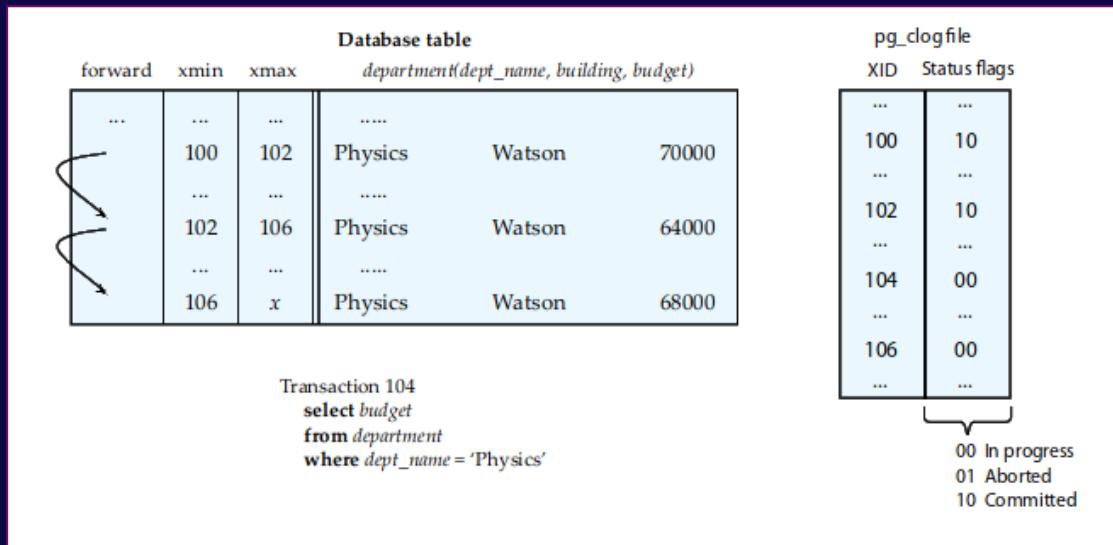


Figure 310 - The PostgreSQL data structures used for MVCC

The **MVCC scheme** implemented in PostgreSQL maintains different versions of each row that correspond to instances of the row at different points in time. This enables a transaction to see a consistent snapshot of the data by selecting the most recent version of each row committed before taking the snapshot. With the **MVCC protocol**, readers never block writers, and vice versa, since readers access the most recent version of a row that is part of the transaction's snapshot, while writers create their own separate copy of the row to be updated. The only conflict that can cause a transaction to be blocked arises if two writers try to update the same row.

<i>dept.name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 331 - The department relation

At the heart of **PostgreSQL's MVCC** is the concept of tuple visibility, which defines which of the potentially many versions of a row in a table is valid within the context of a given statement or transaction. A tuple is visible for a transaction if the tuple was created by a transaction that committed before the transaction took its snapshot, and updates to the tuple (if any) were executed by a transaction that is either aborted, started running after the transaction took its snapshot, or was active when the transaction took its snapshot.

As we delve deeper into the inner workings of **PostgreSQL's MVCC protocol**, a fascinating and intricate dance between transactions and tuples emerges. This protocol operates on the principle of tuple visibility, with each tuple being associated with a **creation-transaction ID** and an **expire-transaction ID**.

To be visible, a tuple must satisfy several conditions. Firstly, its creation-transaction ID must be a committed transaction according to the pg clog file. Secondly, it must be less than the cutoff transaction **ID xmax** recorded by SnapshotData, and finally, it must not be one of the active transactions stored in SnapshotData.

Conversely, an expire-transaction ID, if it exists, must be an aborted transaction according to the **pg clog file**, or greater than or equal to the cutoff transaction ID xmax recorded by SnapshotData, or one of the active transactions stored in SnapshotData.

<i>Lock name</i>	<i>Conflicts with</i>	<i>Acquired by</i>
ACCESS SHARE	ACCESS EXCLUSIVE	<code>select</code> query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	<code>select for update</code> query <code>select for share</code> query
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	<code>update</code> <code>delete</code> <code>insert</code> queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	<code>vacuum</code> <code>analyze</code> <code>create index concurrently</code>
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	<code>create index</code>
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	---
EXCLUSIVE	All except ACCESS SHARE	---
ACCESS EXCLUSIVE	All modes	<code>drop table</code> <code>alter table</code> <code>vacuum full</code>

Figure 334 - Table-level lock modes

To illustrate the concept of tuple visibility, let us consider a hypothetical database depicted in the Figure above.

The behavior of **PostgreSQL MVCC** varies depending on whether the SQL statement in question is an insert, select, update, or delete statement. In the case of an insert statement, a new tuple is created based on the data in the statement, and the tuple header is initialized with the creation ID before being inserted into the table. If no integrity constraints are violated, no further interaction with the **concurrency-control protocol** is necessary.

When executing **select**, **update**, or **delete statements**, the interaction with the **MVCC protocol** depends on the isolation level specified by the application. If the isolation level is read committed, the system creates a new **SnapshotData data structure** and identifies target tuples visible with respect to the SnapshotData, and matching the search criteria of the statement. In the case of an **update** or **delete statement**, an additional step is necessary before the actual operation can proceed.

This is because the visibility of a tuple only ensures that the tuple was created by a committed transaction before the statement started. However, it may have been updated or deleted by another concurrent transaction since the query's start. If the **expire-transaction ID** corresponds to a transaction that is still in progress, the system must wait for the completion of this transaction first. If the transaction aborts, the operation can proceed and perform the actual modification. If the transaction commits, the search criteria must be re-evaluated, and the row can only be modified if it still meets the criteria.

PostgreSQL MVCC provides the read-committed isolation level for update and deletes statements, but this can result in non-repeatable reads and phantom reads, as queries within a transaction may see different snapshots of the database. To provide the serializable isolation level, **PostgreSQL MVCC** uses the snapshot at the start of the transaction to determine tuple visibility. Additionally, updates and deletes are processed differently in serializable mode compared to read-committed mode, with transactions waiting for concurrent transactions to complete before proceeding.

Storage and Indexing

In its approach to data layout and storage, PostgreSQL prioritizes simplicity of implementation and ease of administration. To achieve these goals, the database relies on “cooked” file systems instead of managing the physical layout of data on raw disk partitions. PostgreSQL maintains a list of directories in the file hierarchy, known as tablespaces, and each installation is initialized with a default tablespace. Additional tablespaces may be added at any time, and when *creating a table, index, or database*, the user may specify any existing tablespace for storage.

- Creating multiple tablespaces is particularly useful if they reside on different physical devices, allowing faster devices to be dedicated to data in higher demand. This design potentially leads to performance limitations, as PostgreSQL clashes with the file system, resulting in double buffering.
- Performance can also be limited by the fact that PostgreSQL stores data in *8-KB blocks*, which may not match the kernel’s block size. While it’s possible to change the PostgreSQL block size, this may have undesired consequences, such as limiting the database’s ability to store large tuples efficiently or being wasteful when a small region of a file is accessed.

Modern enterprises increasingly use external storage systems, such as **network-attached storage** and **storage-area networks**, instead of disks attached to servers. PostgreSQL may directly leverage these technologies because of its reliance on cooked file systems. Despite the performance limitations, many PostgreSQL developers believe that for the vast majority of applications and the **database's audience**, the ease of administration and management, as well as simplicity of implementation, justify the tradeoff.

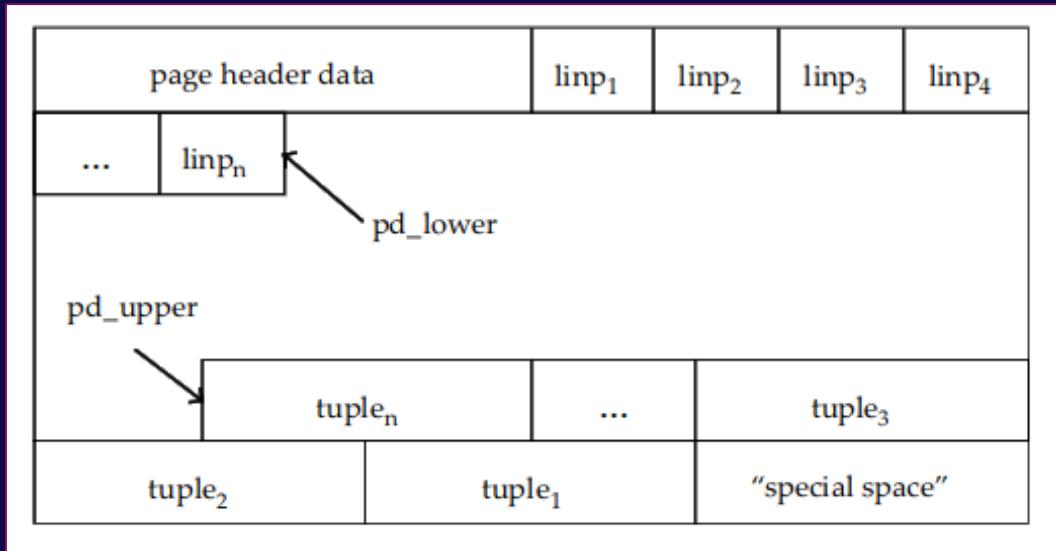


Figure 335 - Slotted-page format for PostgreSQL tables

Tables are the primary unit of storage in PostgreSQL and are stored in heap files using a slotted-page format. Each page contains a header followed by an array of line pointers, and a line pointer holds the offset and length of a specific tuple in the page. A record in a heap file is identified by its **tuple ID**, consisting of a **block ID** and a **slot ID**, allowing access to the tuple.

Although PostgreSQL permits tuples in a page to be **deleted** or **updated**, neither operation actually deletes or replaces old versions of rows immediately. Expired tuples may be physically deleted by later commands, causing holes to be formed in a page, and the indirection of accessing tuples through the line pointer array allows the reuse of such holes.

A tuple's length is limited by the length of a data page, making it difficult to store very long tuples. PostgreSQL attempts to "*toast*" individual large attributes by compressing the value. If this doesn't shrink the tuple enough to fit on the page, the data in the toasted attribute is replaced with a reference to a copy stored outside the page.

PostgreSQL indices are data structures that provide a dynamic mapping from search predicates to sequences of tuple IDs from a specific table. The returned tuples are intended to match the search predicate, and PostgreSQL supports several index types, including B-tree and hash indices. While the default index type is a **B+-tree index** based on **Lehman** and **Yao's B-link trees**, PostgreSQL's hash indices are an implementation of linear hashing, useful only for simple equality operations. Despite their limitations, hash indices in PostgreSQL have been shown to have a lookup performance no better than **B-trees**.

Query Processing and Optimization

In the intricate world of PostgreSQL database management, every query is a carefully crafted and meticulously optimized masterpiece. From the moment a query is received, it is subject to a series of transformations that result in a query plan, the execution of which will ultimately determine the success of the operation.

The first step in this process is **query rewrite**, which involves firing rules that have been defined explicitly by users or implicitly through the definition of views. As the query is repeatedly rewritten and checked against rules until a fixed point is reached, the system seamlessly handles complex statements containing select **clauses**, **update**, **delete**, and insert statements with the finesse of a skilled artisan.

Once the query has been rewritten, it is subject to the planning and optimization phase. The PostgreSQL optimizer employs a cost-based approach to generate an access plan with a minimal estimated cost. From the innermost subquery, the optimizer generates a plan for each query block, using a standard planner or a genetic query optimizer for more complex queries.

The result of the optimization phase is a query plan that is a tree of relational operators, each representing a specific operation on one or more sets of tuples. The crucial cost model depends on accurate statistical estimates of the number of tuples to be processed at each operator, which are maintained by the system.

Finally, the query plan is executed by the executor module, which follows the iterator model with a set of four functions implemented for each operator. Access methods, join methods, and sort and aggregation operators work in harmony to bring the query to fruition with a deftness that would make even the most accomplished maestro envious.

In the world of PostgreSQL, every query is a symphony of precision and artistry, executed with the utmost care and attention to detail.

System Architecture

PostgreSQL, the renowned open-source database management system, boasts a distinctive system architecture, one that operates on a **process-per-transaction model**. The backbone of this architecture is the postmaster, a central coordinating process that oversees the entire PostgreSQL site. It initializes and shuts down the server and is also responsible for handling connection requests from new clients. Upon receiving a connection request, the postmaster assigns each new client to a back-end server process, which takes on the onus of executing queries on behalf of the client and returning the results to it.

To connect to the PostgreSQL server and submit queries, client applications can utilize the several database application programmer interfaces supported by PostgreSQL. These include *libpq*, **JDBC**, **ODBC**, **Perl DBD**, and others provided as client-side libraries. For instance, the command-line `psql` program, part of the standard PostgreSQL distribution, is an example of a client application.

The **postmaster** is responsible for processing initial client connections, and it listens continuously for new connections on a known port. After performing essential steps like user authentication, the postmaster spawns a **new back-end server** process to handle the new client, following which the client interacts solely with the back-end server process, submitting queries and receiving query results. This is the fundamental premise of the process-per-connection model that PostgreSQL is built upon.

The back-end server process is responsible for executing client queries by performing the necessary query-execution steps, including parsing, optimization, and execution. Each back-end server process can handle multiple connections, thereby enabling efficient query processing for multiple clients.

The system architecture of PostgreSQL is depicted in Figure below:

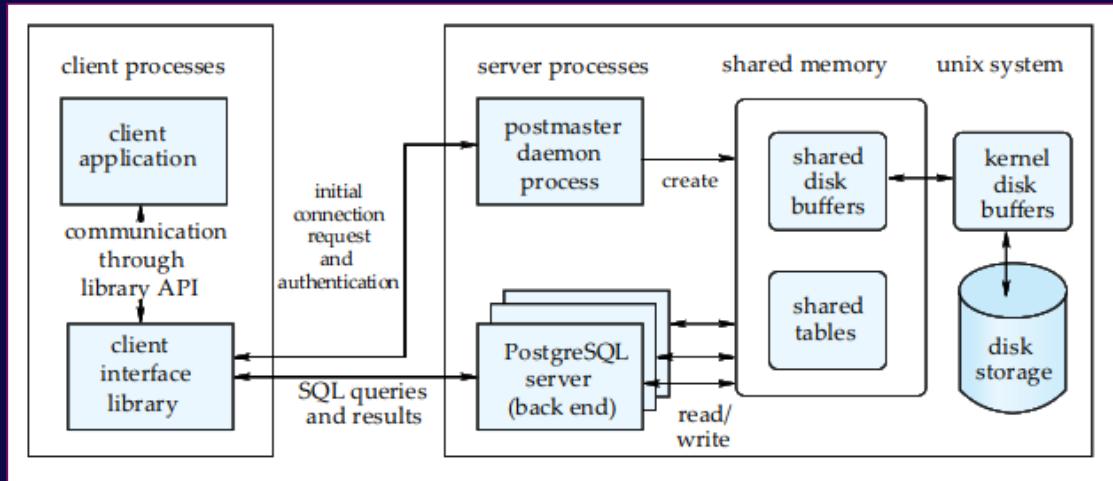


Figure 336 - The PostgreSQL system architecture

10.2 Oracle

In 1977, three software pioneers, *Larry Ellison, Bob Miner, and Ed Oates* founded *Software Development Laboratories*, which was later renamed **Oracle**. At that time, there were no commercial relational database products, and Oracle set out to build a relational database management system to fill that void. It quickly became a pioneer in the **RDBMS market** and has maintained its position as a market leader ever since.

Oracle's product and service offerings have expanded over the years, beyond just the relational database server, to include middleware and applications. Notably, the company has made strategic acquisitions, ranging from small companies to large, publicly traded ones, such as Peoplesoft, Siebel, Hyperion, and BEA. This has broadened Oracle's portfolio of enterprise software products, making it one of the most comprehensive in the industry.

This chapter is focused on Oracle's flagship database product, **Oracle11g**, and its closely related products. Oracle continuously develops new versions of its products, so all product descriptions are subject to change. However, the feature set described here is based on the first release of **Oracle11g**. Oracle's long-standing position in the market and its commitment to innovation and expansion continue to shape the technology landscape today.

Database Design and Querying Tools

Oracle's suite of software products called Oracle Fusion Middleware offers a variety of tools for database design, querying, report generation, and data analysis, including **OLAP**. The portfolio of tools encompasses both traditional software using Oracle's PL/SQL programming language and newer ones based on *Java/J2EE* technologies, supporting open standards such as *SOAP*, *XML*, *BPEL*, and *UML*.

For database and application design, the **Oracle Application Development Framework (ADF)** is an **end-to-end J2EE-based development framework**, while Oracle Designer is a database design tool. Oracle Warehouse Builder is a tool for data warehousing, including schema design, data mapping and transformations, data load processing, and metadata management.

For querying, report generation, and data analysis, Oracle offers a comprehensive suite of tools called **Oracle Business Intelligence Suite (OBI)**, which includes a Business Intelligence server and tools for ad hoc querying, dashboard generation, reporting, and alerting. The component for ad hoc querying, Oracle BI Answers, allows users to build a query with a point-and-click interface that generates physical queries from logical queries. Results can be presented in various formats and saved for later modification.

Oracle's tools offer a robust set of features for all aspects of database management, making it a preferred choice for businesses of all sizes.

SQL Variations and Extensions

As I perused the vast expanse of Oracle's capabilities, my eyes fixated upon a section detailing the multitude of SQL variations and extensions supported by the tech giant. Oracle boasts full or partial support for all core *SQL:2003* features, save for features-and-conformance views, and an array of language constructs, some of which conform to *Optional Features of SQL Foundation:2003*, while others are unique to Oracle in syntax or functionality.

Of particular note is Oracle's robust support for **object-relational constructs**, including object types, collection types, object tables, table functions, object views, methods, and user-defined aggregate functions. Additionally, Oracle **XML DB** offers in-database storage for XML data, with a broad set of XML functionality, including **XML Schema** and **XQuery**.

Oracle's procedural languages, *PL/SQL* and *Java*, are supported, with Java integration through a Java virtual machine inside the database engine. Oracle provides a package to encapsulate related procedures, functions, and variables into single units, along with support for **SQLJ** (*SQL embedded in Java*) and **JDBC**, and a tool to generate Java class definitions corresponding to user-defined database types.

Oracle also supports the creation of dimensions as metadata objects to enable query processing against databases designed based on dimensional modeling techniques, as well as offering support for analytical database processing through a variety of SQL constructs and native multidimensional storage inside the relational database server. The use of these structures in conjunction with a relational database allows for the storage of data in both relational and multidimensional formats, while still taking advantage of Oracle's comprehensive suite of features in areas such as backup and recovery, security, and administration tools.

Oracle's expansive support for SQL variations and extensions, along with its robust object-relational features, **XML DB functionality**, procedural languages, and support for dimensional modeling and **OLAP**, makes it a powerhouse in the world of database management systems, offering unparalleled capabilities for users across a wide range of industries and use cases.

Storage and Indexing

In the world of Oracle database management, information is stored in files and accessed through instances, which consist of a shared memory area and a set of processes that interact with the data. At the heart of this **system lies the control file**, a small but essential piece of metadata that enables the instance to operate.

Each database is comprised of **logical storage** units known as table spaces, which, in turn, consist of physical structures called data files. These files can be part of a **file system** or **raw devices**, depending on the configuration of the system. Oracle databases typically include several table spaces, such as the system and auxiliary sysaux table spaces, which contain data dictionary tables and storage for triggers and stored procedures.

To optimize performance, user data is often stored in separate table spaces from system data. In addition, table spaces are used as a means of moving data between databases, a process that can be significantly expedited through Oracle's transportable table spaces feature.

Space within table spaces is divided into **units called segments**, each containing data for a specific data structure. These segments include data segments, index segments, temporary segments, and undo segments, which contain important undo information for transaction management and recovery.

Oracle provides detailed storage parameters that allow for precise control over how space is allocated and managed, including the size of new extents and the percentage of space utilization at which a database block is considered full.

Oracle supports a variety of table types, including *heap-organized tables*, *nested tables*, and *temporary tables*, each with its own unique characteristics and advantages. Additionally, clustering offers an alternative method of file organization for table data, enabling rows from different tables to be stored together in the same block based on common columns.

Oracle, the venerable database management system, offers its users a wealth of indexing options, each with its own unique set of advantages and characteristics. From the ubiquitous **B-tree index** to the space-efficient bitmap index and the **versatile function-based index**, Oracle provides a powerful toolkit to optimize database performance.

Perhaps the most commonly used index type in Oracle is **the B-tree index**, which can be created on one or multiple columns. Interestingly, in the world of Oracle (*as well as other database systems*), a **B-tree index** is equivalent to a **B+-tree index**. Each index entry consists of the values of the indexed columns and the row identifier. If the prefix of the entry is repeated across multiple entries, Oracle has the option to compress the prefix to save space. This prefix compression can lead to substantial space savings and improved performance.

The **bitmap index**, on the other hand, uses a bitmap representation for **index entries**, leading to significant space and disk **I/O savings**, particularly for columns with a moderate number of distinct values. The bitmap index in Oracle uses a B-tree structure to store entries, but with a different format. The bitmap represents the space of all possible rows in the table between the start and end row-ids, with each bit in the bitmap representing one possible row. If the column value of that row matches the index entry, the bit is set to **1**, otherwise, it is set to **0**. The compression algorithm used in Oracle is a variation of **Byte-Aligned Bitmap Compression**, which stores a section of the bitmap as verbatim bitmaps if the distance between two consecutive **1s** is small enough. However, if the distance between two **1s** is large enough, a run-length of **0s** is stored.

```
upper(name) = 'VAN GOGH'
```

Notably, bitmap indices allow multiple indices on the same table to be combined in the same access path, significantly improving performance. Oracle can convert row-ids to the compressed bitmap representation, making it possible to use a regular **B-tree index** anywhere in a **Boolean tree** of bitmap operation simply by putting a row-id-to-bitmap operator on top of the index access in the execution plan.

```
select *
  from employees
 where contains(resume,'LINUX');
```

Finally, Oracle provides a function-based index that allows indices to be created on expressions that involve one or more columns, such as **col1 + col2 * 5**. This feature enables case-insensitive searches and other advanced querying options, offering unparalleled flexibility for database optimization.

In the ever-evolving world of data warehousing, efficient partitioning, and materialized views have emerged as powerful tools to accelerate query processing and enhance database performance. In a seminal work on the topic, Oracle details several **types of partitioning strategies**, including range, hash, list, composite, and reference partitioning, each tailored to different data structures and performance requirements.

Range partitioning, for instance, is particularly well-suited for date columns, enabling a rolling window of historical data to be loaded efficiently into a warehouse. As new data is added, it is loaded into a separate table with the same column definition as the **partitioned table**, checked for consistency, cleansed, and indexed. The system can then make the separate table a new partition of the partitioned table through a simple metadata change, without affecting existing data in any way. Queries can be optimized by restricting data access to relevant partitions, eliminating the need for time-consuming table scans.

Hash partitioning, in contrast, is useful when it is important to distribute rows evenly among partitions or when partition-wise joins are critical for query performance. List partitioning is ideal when partitioning column data have a relatively small set of discrete values, while composite partitioning enables subpartitioning of **tables by range, list, or hash**.

Materialized views, another crucial tool in data warehousing, allow the results of an SQL query to be stored in a table and used for later query processing. Oracle maintains the materialized result, updating it whenever referenced tables are updated. This technology is widely used for replication in distributed and mobile environments, as well as in data warehousing to speed up query processing.

Automatic query rewrites in Oracle take advantage of useful materialized views when resolving a query, changing the query to use the materialized view instead of the original tables. Oracle's dimension metadata object specifies hierarchical relationships in tables, enabling the use of materialized views for wider classes of queries.

Partitioning and materialized views are indispensable tools for boosting query performance and enhancing database efficiency, and Oracle's advanced strategies in these areas represent a major breakthrough in data warehousing technology.

Query Processing and Optimization

In the realm of database management systems, Oracle stands out with its diverse range of processing techniques designed to optimize query performance. From full table scans to index scans and fast full scans, Oracle offers a variety of methods for accessing data, each with its own unique set of advantages. Additionally, Oracle provides the ability to combine information from multiple indices, allowing for **multiple where-clause conditions** to be used efficiently in computing the result set.

But it's not just about accessing data - Oracle also offers several types of joins in its execution engine, including inner joins, outer joins, semijoins, and antijoins. Each type of join is evaluated using one of three methods: hash join, sort-merge join, or nested-loop join.

In terms of query optimization, Oracle takes a multi-step approach. One key step involves performing various query transformations and rewrites, including view merging, complex view merging, subquery flattening, materialized view rewrite, and star transformation. Another step involves access path selection to determine access paths, join methods, and join order.

```
fact table.fki in  
(select pk from dimension tablei  
where <conditions on dimension tablei >)
```

However, not all transformations are beneficial, which is why Oracle employs a cost-based approach to query transformations and access path selection. For each transformation that is attempted, access path selection is performed to generate a cost estimate, and the transformation is accepted or rejected based on the cost for the resulting execution plan.

Oracle's query processing engine is a powerful tool for optimizing database queries. With its diverse range of processing techniques, multi-step approach to query optimization, and cost-based query transformations, Oracle provides a robust solution for managing complex data sets with speed and efficiency.

As Oracle continues to lead the way in the world of database management, their commitment to improving performance and efficiency remains unwavering. One of the latest features is the Partition Pruning method, which enables the optimizer to efficiently match conditions in the where clause of a query with the partitioning criteria for a table. This allows only relevant partitions to be accessed, leading to significant speedups in performance.

Furthermore, the **SQL Tuning Advisor** provides valuable insights into high-load SQL statements by analyzing statistics, profiling, and making access path and SQL structure recommendations. This feature is especially useful for applications that generate the same set of SQL statements repeatedly, allowing for more efficient execution plans to be generated.

The SQL Plan Management feature is another crucial tool for maintaining a set of trusted execution plans for a workload. This mitigates the risk of performance degradation and application breakages that may result from changes in database behavior. By capturing, selecting, and evolving SQL plan baselines, Oracle ensures that only verified, efficient plans are used in production.

Last but not least, the Parallel Execution feature is designed to divide the workload of a single SQL statement between multiple processes, leading to faster query results. With these innovative features, Oracle continues to revolutionize the field of database management, setting new standards for performance, efficiency, and reliability.

Concurrency Control and Recovery

In the world of database management systems, Oracle stands tall as a behemoth. Providing a rich array of features, including concurrency control and recovery techniques, **Oracle** is a **go-to for many organizations**. With its multi-version concurrency control mechanism, based on the snapshot isolation protocol, Oracle offers a unique approach to read-only queries. Unlike other systems, Oracle gives read-only queries a read-consistent snapshot, which provides a view of the database as it existed at a specific point in time.

Oracle's scheme allows for long-running queries to run on a system with a large amount of transactional activity. Unlike database systems that use **read locks**, **Oracle's concurrency model permits read and write operations** to happen concurrently without any blockages. This feature allows for a high degree of concurrency, even with long-running queries, such as reporting queries.

Oracle's flashback feature also allows a user to perform operations on data that existed at a point in time, providing a mechanism to deal with user errors. Additionally, the flashback archive feature creates an internal history version of a table, allowing for the tracking of changes to a table beyond what would be possible through normal undo retention.

Oracle supports two **ANSI/ISO isolation levels**, read committed and serializable, with row-level locking, which prevents inconsistencies due to **DML activity**, and table locks that prevent inconsistencies due to **DDL activity**. With its rich feature set and robust approach, Oracle continues to be a leader in the world of database management systems.

System Architecture

In the realm of database applications, the system architecture is of paramount importance to ensure the optimal execution of SQL statements. At the heart of this architecture lies an operating system process that executes code within the database server, and Oracle provides two configurations to manage this process: the **dedicated server** and the **shared server**.

For the former, memory structures are of utmost significance and are broadly divided into three categories: **software code areas**, **system global areas (SGA)**, and **program global areas (PGA)**. Each process in the dedicated server is allocated a **PGA** to hold its local data and control information, including memory for sorting and hashing operations, which play a crucial role in the evaluation of SQL statements. Deciding how much memory should be allocated for each operation is a complex task that requires balancing the available memory and performance requirements of different operations.

The **SGA**, on the other hand, is a memory area for structures that are shared among users and comprises several significant structures such as the buffer cache, redo log buffer, and shared pool. The buffer cache stores frequently accessed data blocks in memory to reduce the need for **physical disk I/O**, while the shared pool stores the sharable parts of the data structures representing **SQL statements**, including the text of the statement. This feature enables multiple users to execute the same **SQL statement** simultaneously, resulting in reduced memory consumption and compilation time.

The dedicated server process structures are of two types: server processes that execute **SQL statements** and **background processes** that perform various administrative and performance-related tasks. These processes play a crucial role in the smooth functioning of the server, and Oracle generates about two dozen different types of background processes to manage this. These processes include the database **writer**, **log writer**, **checkpoint**, **system monitor**, **process monitor**, **recoverer**, and **archiver**, among others.

The dedicated server architecture of Oracle's database application is a complex system comprising multiple memory and process structures. It ensures optimal performance by dynamically allocating memory to active operations, reducing **physical disk I/O**, and minimizing memory consumption and compilation time. The multitude of background processes, each performing specific tasks, guarantees the smooth functioning of the server and enables efficient management of user requests.

Replication, Distribution, and External Data

Oracle's support for replication, distribution, and external data management is a boon for businesses seeking efficient and effective database management solutions. With its support for several types of replication, Oracle enables the replication of data from master sites to other sites using materialized views.

These views can be **read-only** or updatable, allowing for a wider range of view definitions. Oracle also supports multiple master sites for the same data, and updates can be propagated either asynchronously or synchronously. In cases of asynchronous replication, updates are sent in batches and may require conflict resolution. For synchronous replication, updates are propagated immediately to all other sites.

Oracle's support for distributed databases is another asset, **allowing for queries** and **transactions spanning multiple databases** on different systems. The use of gateways enables remote systems to include **non-Oracle databases**, with built-in capability to optimize queries including tables at different sites. Transactions spanning multiple sites are enabled through a **built-in two-phase-commit protocol**.

```
create table table as  
    select ... from < external table >  
    where ...
```

Oracle's support for external data sources is also noteworthy, with mechanisms for fast parallel loads of large amounts of data from **external files**, and **external data sources**, such as **flat files**, can be referenced in the from clause of a query as if they were regular tables.

The external table feature is intended for extraction, transformation, and loading operations in a data warehousing environment, and data can be loaded into the data warehouse from a flat file using SQL statements that allow for transformations and filtering.

Additionally, Oracle provides an export utility for unloading data and metadata into dump files that can be moved to another system and loaded into another Oracle database using the corresponding import utility.

Database Administration Tools

In the realm of database administration, Oracle has been at the forefront of providing an array of tools and features for system management and application development.

The Oracle team has been relentless in their pursuit of manageability, striving to reduce the complexity of all aspects of creating and administering an Oracle database. Their efforts have spanned a broad spectrum of areas, including: *database creation, tuning, space management, storage management, backup and recovery, memory management, performance diagnostics, and workload management*.

At the core of Oracle's database systems management toolkit lies the **Oracle Enterprise Manager (OEM)**, which serves as the mainstay for administering an Oracle database. With its intuitive graphical user interface, OEM streamlines tasks such as configuration, performance monitoring, resource management, security management, and access to various advisors. Moreover, OEM provides integrated management of Oracle's applications and middleware software stack.

Another cornerstone of Oracle's manageability effort is the **Automatic Workload Repository (AWR)**, which captures and records a variety of information related to workloads and resource consumption at regular intervals. By monitoring the characteristics of a workload over time, Oracle can diagnose deviations from normal behavior, such as a significant **performance degradation of a query, lock contention**, and **CPU bottlenecks**. AWR's wealth of recorded data serves as the foundation for various advisors that provide analyses of the system's performance, as well as advice on how to improve it. Oracle's advisors cover a wide range of areas, including SQL tuning, creation of access structures such as indices and materialized views, memory sizing, segment defragmentation, and undo sizing.

Database resource management is a critical aspect of database administration, and Oracle has not neglected it. The **Database Resource Management** feature allows database administrators to divide users into resource consumer groups with different priorities and properties, thereby enabling control over how processing power is allocated among users or groups of users. Through this feature, administrators can set limits for the degree of parallelism for parallel execution and the time limits for how long an SQL statement is allowed to run.

Additionally, the resource manager can limit the number of active user sessions for each resource consumer group and control other resources such as undo space.

Data Mining

Oracle Data Mining is a powerful tool that enables the process of data mining directly within the database, eliminating the need to move large data sets to other engines for analysis. This is a significant advantage as it **saves time**, **reduces costs**, and **enables real-time analysis** of new data as it enters the database. With algorithms for both supervised and unsupervised learning, Oracle Data Mining provides a range of functions for classification, regression, attribute importance, anomaly detection, clustering, association rules, and feature extraction. Furthermore, Oracle offers a variety of statistical functions within the database, **covering linear regression**, **correlation**, **hypothesis testing**, **distribution fitting**, and **Pareto analysis**.

To access the data mining functionality, Oracle provides two interfaces: one based on Java and the other based on **Oracle's procedural language PL/SQL**. Once a model has been built on an Oracle database, it can be effortlessly deployed on other Oracle databases. Oracle Data Mining is an exceptional tool that empowers users to extract valuable insights from data without the need for cumbersome and costly data transfers.

10.3 IBM DB2 Universal Database

IBM's DB2 Universal Database family is a collection of premier database servers and suites of associated products that facilitate business intelligence, information integration, and content management.

This powerful tool is available on various hardware and operating-system platforms, ranging from handheld devices to mainframes, massively parallel processors (MPP), and large symmetric multiprocessors (SMP) servers. IBM's DB2 Universal Database server supports *Unix variants like Linux, IBM AIX, Solaris, and HP-UX*, along with *Microsoft Windows, IBM MVS, IBM VM, IBM OS/400*, and several others. Moreover, the DB2 Everyplace edition offers support for operating systems such as PalmOS and Windows CE, while the DB2 Express-C edition is available free of charge.

Applications can migrate smoothly between *low-end platforms* and *high-end servers* due to the portability of the DB2 interfaces and services. Apart from the core database engine, the DB2 family includes several other products that provide essential tooling, administration, replication, distributed data access, pervasive data access, OLAP, and many other features.

Overview

The history of **IBM's DB2 database engine** can be traced back to the **System R project** at the *IBM San Jose Research Laboratory*. Since the release of the first DB2 product in 1984, IBM has continually enhanced its functionality and performance, incorporating advanced features such as transaction processing, query optimization, and active-database support.

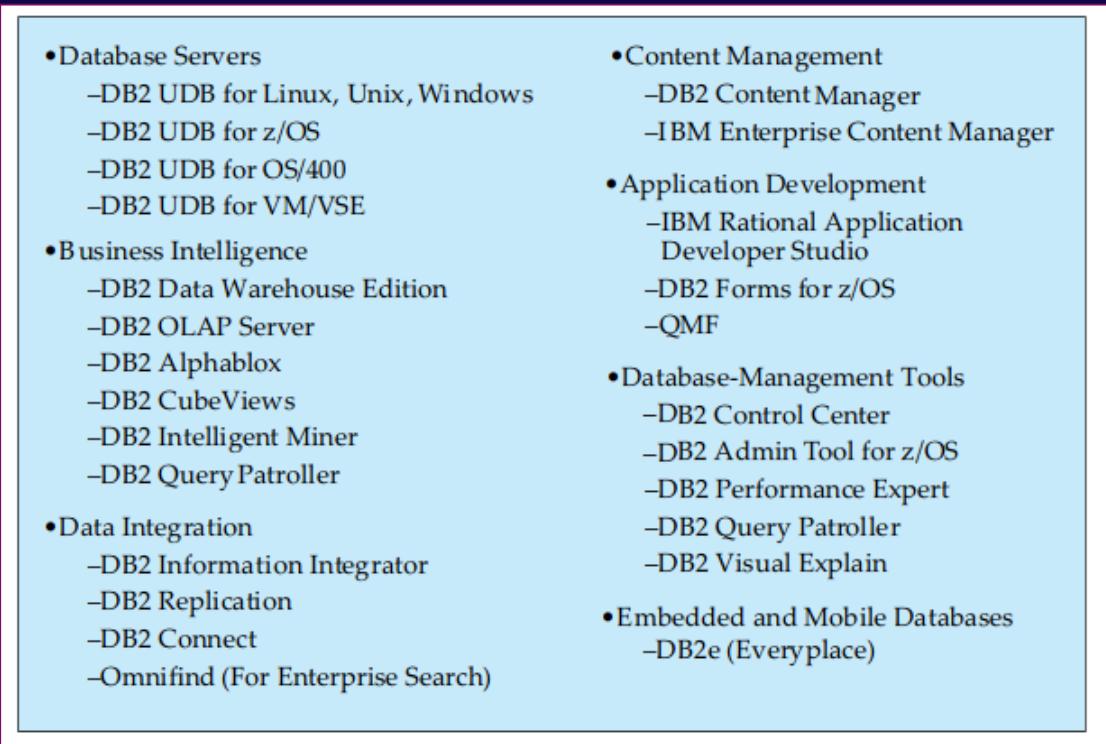


Figure 337 - The DB2 family of products

With support for a wide range of server and operating-system platforms, the DB2 database engine consists of four code base types, each with slightly different features. In this chapter, the focus is on the **DB2 Universal Database (UDB) engine**, which supports *Linux*, *Unix*, and *Windows*. The latest version of DB2 UDB for *Linux*, *Unix*, and *Windows* as of 2009 is *version 9.7*, which includes several new features such as native support for **XML in shared-nothing environments**, compression for tables and indexes, automatic storage management, and improved support for procedural languages.

Database-Design Tools

The **DB2 Universal Database** boasts a wide range of features to support efficient and effective database design. With a plethora of industry-standard database design and CASE tools at their disposal, users can take advantage of DB2's ability to generate DB2-specific DDL syntax through tools such as ERWin and Rational Rose.

Moreover, the **DB2 Control Center GUI tool** offers a comprehensive suite of design- and administration-related tools, allowing users to view, define and create objects and SQL queries, as well as query results. The entire DB2 family of products offers support for these tools, as well as plug-in modules for application development in IBM Rational *Application Developer* and Microsoft *Visual Studio products*.

With support for logical and physical database features such as constraints, triggers, tablespaces, bufferpools, and partitioning, DB2 offers users the flexibility and customization necessary for efficient database design and administration. Additionally, tools such as db2look enable users to obtain a full set of DDL statements for a database schema, making testing and replication a breeze.

SQL Variations and Extensions

The art of database management has reached new heights with DB2, providing users with a rich array of SQL features for all aspects of database processing.

Among the many salient attributes of DB2 are the support for **XML object-relational** and **application-integration features**, which are the focus of this section. The *XML* functions in DB2 are diverse and provide extensive *XML* manipulation capabilities, making them an integral part of *SQL*. A typical example is the construction of an *XML* document from relational tables, as exemplified in the figure below:

```
select xmlelement(name 'PO',
    xmlattributes(poid, orderdate),
    (select xmlagg(xmlelement(name 'item',
        xmlattributes(itemid, qty, shipdate),
        (select xmlelement(name 'itemdesc',
            xmlattributes(name, price))
        from product
        where product.itemid = lineitem.itemid)))
    from lineitem
    where lineitem.poid = orders.poid))
from orders
where orders.poid= 349;
```

Figure 338 - DB2 SQL XML query

DB2 also supports **user-defined data types (UDTs)**, both distinct and structured.

These types can be based on the built-in data types of DB2, with additional or alternative semantics defined by the user. For instance, one can define a distinct data type for the US dollar, and then create a field with the new data type in a table. Structured data types, on the other hand, are complex objects consisting of two or more attributes.

They can be used to create typed tables, nested attributes inside columns of tables, and even type hierarchies.

```
<PO poid = "349" orderdate = "2004-10-01">
  <item itemid="1", qty="10", shipdate="2004-10-03">
    <itemdesc name = "IBM ThinkPad T41", Price = "1000.00 USD"/>
    </item>
</PO>
```

Figure 339 - Purchase order in XML for id=349

Moreover, the latest version of DB2, *version 9*, has added new features that are worthy of note. The native storage of XML data as an *XML* type and native support for the XQuery language are some of these new features.

Additionally, specialized storage, indexing, query processing, and optimization techniques have been introduced to facilitate the efficient processing of *XML* data and queries. APIs have also been extended to deal with *XML* data and *XQuery*.

```
create distinct type us dollar as decimal(9,2);

select product from us sales
  where price > us dollar(1000);

create type department t as
  (deptname varchar(32),
   depthead varchar(32),
   faculty count integer)

mode db2/sql;

create type point t as
  (x coord float,
   y coord float)

mode db2/sql;

create table dept of department t;
```

DB2 stands out as a remarkable database management system, with a rich array of SQL features that are useful for various aspects of database processing. The XML object-relational and application-integration features, UDTs, and other advanced features of DB2 provide users with diverse and powerful tools for managing their databases.

In the ever-evolving world of database management, user-defined functions, and methods have become crucial feature for users. DB2, a popular database management system, allows its users to define their own functions and methods, which can then be seamlessly integrated into SQL statements and queries. These functions can generate scalars or tables as their result, and users can create them in various programming languages like *C*, and *Java*, or scripts like **REXX** or **PERL**.

```
create function db2gse.GsegeFilterDist (
    operation integer, g1XMin double, g1XMax double,
    g1YMin double, g1YMax double, dist double,
    g2XMin double, g2XMax double, g2YMin double,
    g2YMax double)
returns integer
specific db2gse.GsegeFilterDist
external name 'db2gsefn!gsegeFilterDist'
language C
parameter style db2 sql
deterministic
not fenced
threadsafe
called on null input
no sql
no external action
no scratchpad
no final call
allow parallel
no dbinfo;
```

Figure 340 - Definition of a UDF

In addition to the flexibility of user-defined functions, DB2 also supports **large objects (LOBs)**, which allow for the manipulation of data such as text, images, and video that are typically quite large in size.

DB2 provides three different types of LOBs: *binary large objects (blobs)*, *single-byte character large objects (clob)*s, and *double-byte character large objects (dbclob)*s.

```
create index extension db2gse.spatial.index(
    gS1 double, gS2 double, gS3 double)
from source key(geometry db2gse.ST.Geometry)
generate key using
    db2gse.GseGridIdxKeyGen(geometry..srid,
    geometry..xMin, geometry..xMax,
    geometry..yMin, geometry..yMax,
    gS1, gS2, gS3)

with target key(srsId integer,
    lvl integer, gX integer, gY integer, xMin double,
    xMax double, yMin double, yMax double)
search methods <conditions> <actions>
```

Figure 341 - Spatial index extension in DB2

DB2 also offers indexing extensions, which enable users to create keys from structured data types using the create index extension statement. This feature provides users with the ability to create indices based on specific attributes. Furthermore, users can take advantage of the rich set of constraint-checking features available in DB2 for enforcing object semantics such as uniqueness, validity, and inheritance.

```
select trn id, amount, date
    from transactions where cust id =  order by date
fetch first 1 row only;
```

DB2 can integrate web services as a producer or consumer, which allows users to create web services that can invoke DB2 using SQL statements. The resultant web service call is processed by an embedded web service engine in DB2, and the appropriate **SOAP** response is generated. This feature enables users to create and interact with web services seamlessly.

```
select ticker id, GetQuote(ticker id)
    from portfolio;
```

All these features make DB2 a powerful database management system that provides users with a range of tools to manipulate and manage their data effectively.

Storage and Indexing

In the world of relational databases, DB2 stands tall with its robust storage and indexing architecture. In this carefully crafted design, the storage and indexing system comprises multiple layers, including the **file system or disk-management layer**, **buffer pool management services**, **data objects** like tables, indices, and LOBs, and a range of other features like concurrency and recovery managers.

At the core of this architecture lies the storage abstraction layer, which allows efficient management of logical database tables in a **multinode** and **multidisk environment**. By defining node groups, DB2 offers unparalleled flexibility in allocating table partitions across different nodes in a system. Large tables can be partitioned across all nodes, while small tables can reside on a single node, ensuring optimum performance.

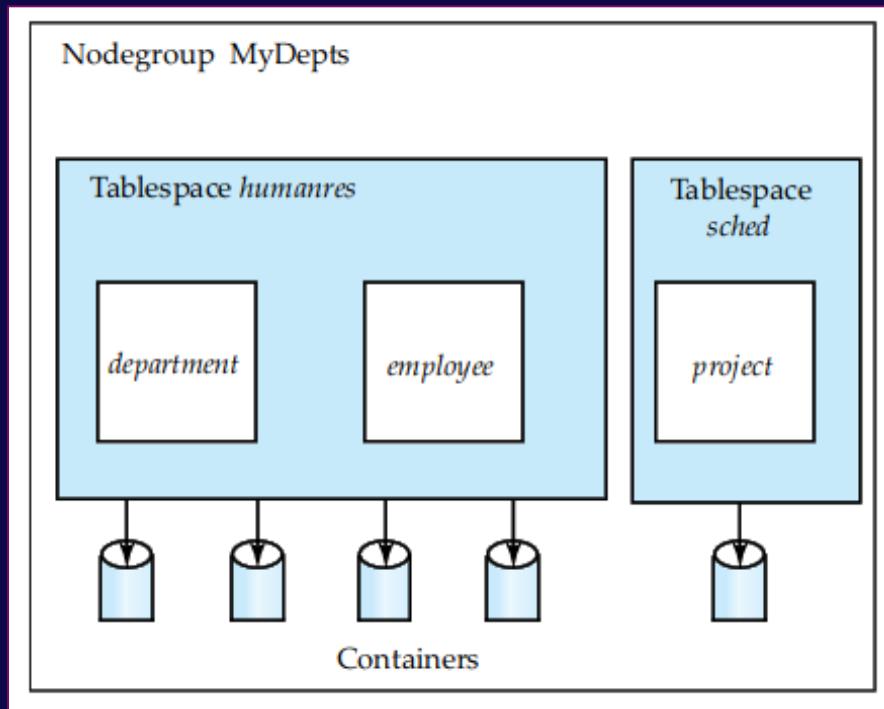


Figure 342 - Tablespaces and containers in DB2

DB2 uses tablespaces to organize tables within a node. Each tablespace consists of one or more containers that reference directories, devices, or files. The administrator can choose to create **either system-managed or DBMS-managed tablespaces**, offering complete control over space management.

The buffer pool is another critical component of the **DB2 storage** and **indexing architecture**, allowing the maintenance of memory copies of objects in a shared data area. DB2 enables buffer pools to be defined using SQL statements, and version **8** allows buffer pools to grow or shrink online automatically, without the need to quiesce the database activity.

```
create bufferpool .....
```

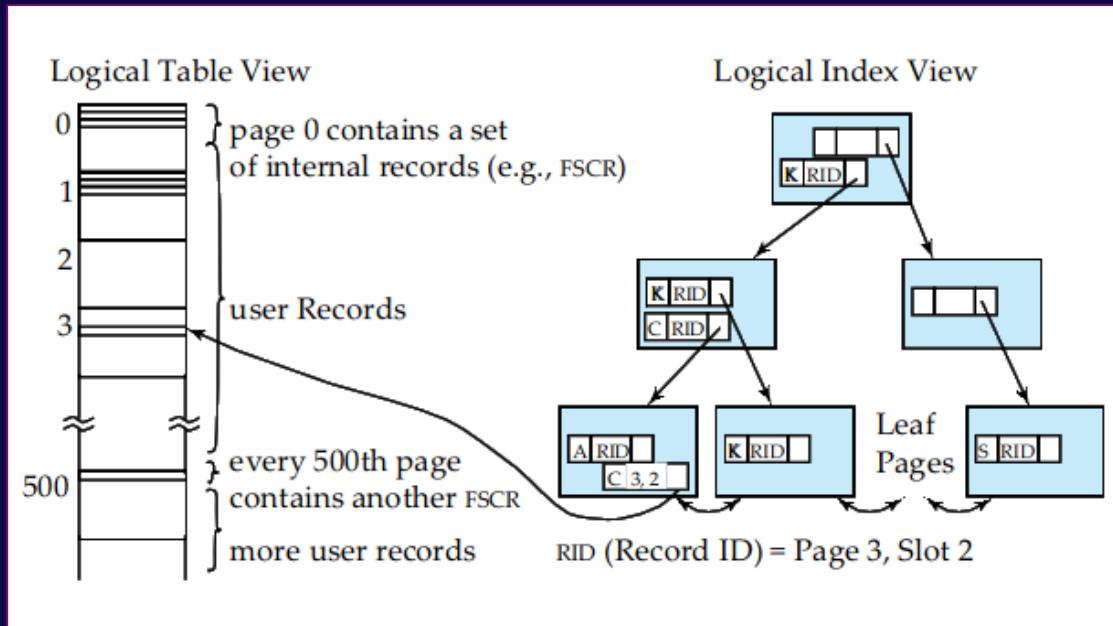
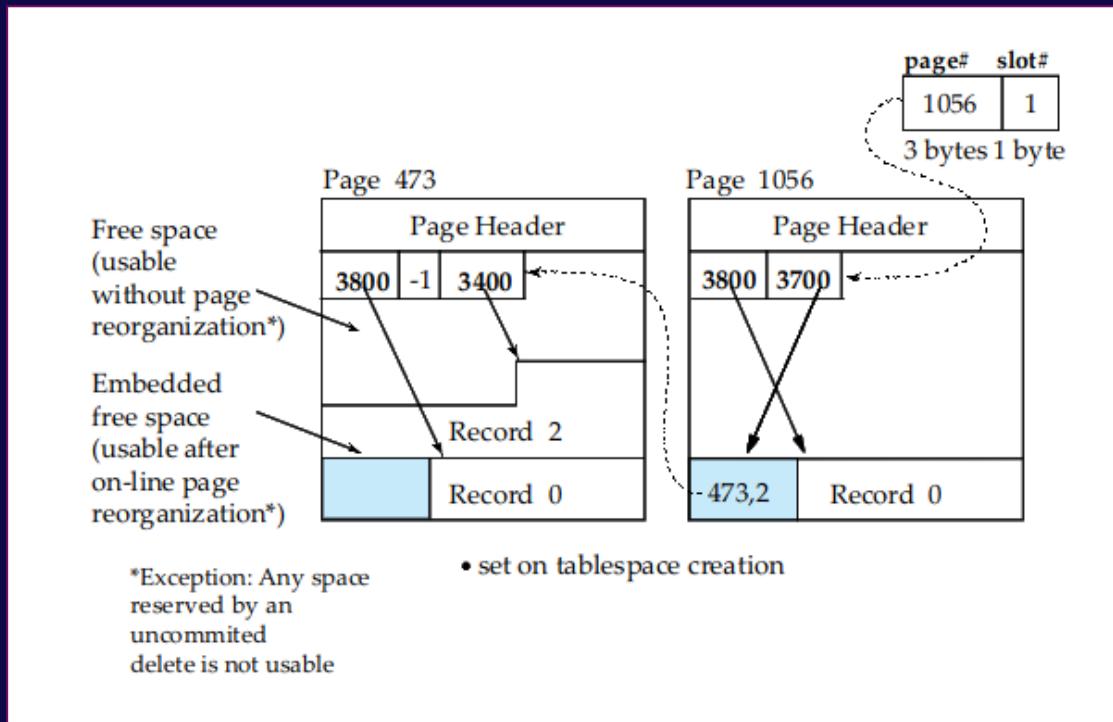


Figure 343 - Logical view of tables and indices in DB2

DB2 also offers support for prefetching and asynchronous writes using separate threads, based on query access patterns.

Tables, **records**, and **indices** are organized efficiently as pages, with free space control records facilitating efficient allocation of space.



The **B+-tree index mechanisms** are used internally, with bidirectional pointers supporting forward and reverse scans. DB2 allows for the use of "*include columns*" in the index definition, offering greater flexibility.

The **DB2 storage** and **indexing architecture** is a well-designed and thoughtfully implemented system that offers unmatched flexibility and efficiency in managing logical database tables in a multinode and multidisk environment.

Multidimensional Clustering

In this section, we delve into the intricacies of **Multidimensional Clustering (MDC)**, a feature that allows for the creation of a DB2 table based on one or more keys as dimensions. By clustering a table along specified dimensions, the organization of the table's data becomes more efficient, as DB2 automatically creates a "*dimension block index*" for each dimension specified. This index allows for quick and efficient data access.

```
create table sales(storeId int,  
                    orderDate date,  
                    shipDate date,  
                    receiptDate date,  
                    region int,  
                    itemId int,  
                    price float  
  
yearOd int generated always as year(orderDate))  
organized by dimensions  
(region, yearOd, itemId);
```

In essence, every unique combination of dimension values forms a logical "cell," which is physically organized as blocks of pages. Each block contains an extent's worth of consecutive pages on disk, and every page of the table is part of exactly one block. The entire table consists of blocks that are all the same size, with block boundaries lining up with extent boundaries.

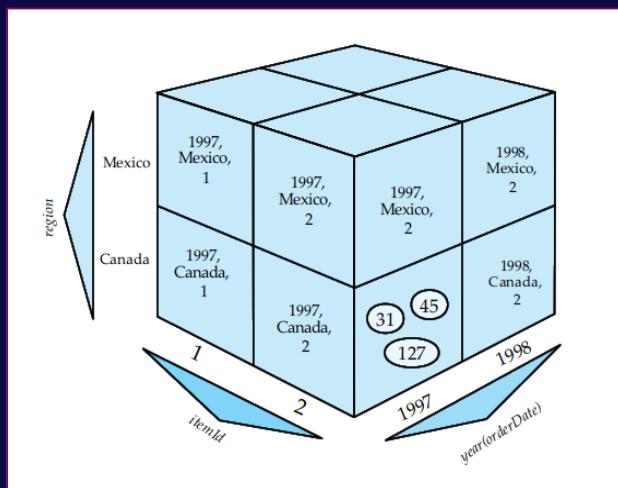


Figure 345 - Logical view of physical layout of an MDC table

To facilitate efficient data access, DB2 creates a dimension block index on each of the specified dimension attributes, pointing to a **block identifier (BID)** at the leaf level. These block indices are much smaller than RID indices and need only be updated when a new block is added to a cell or when existing blocks are emptied and removed.

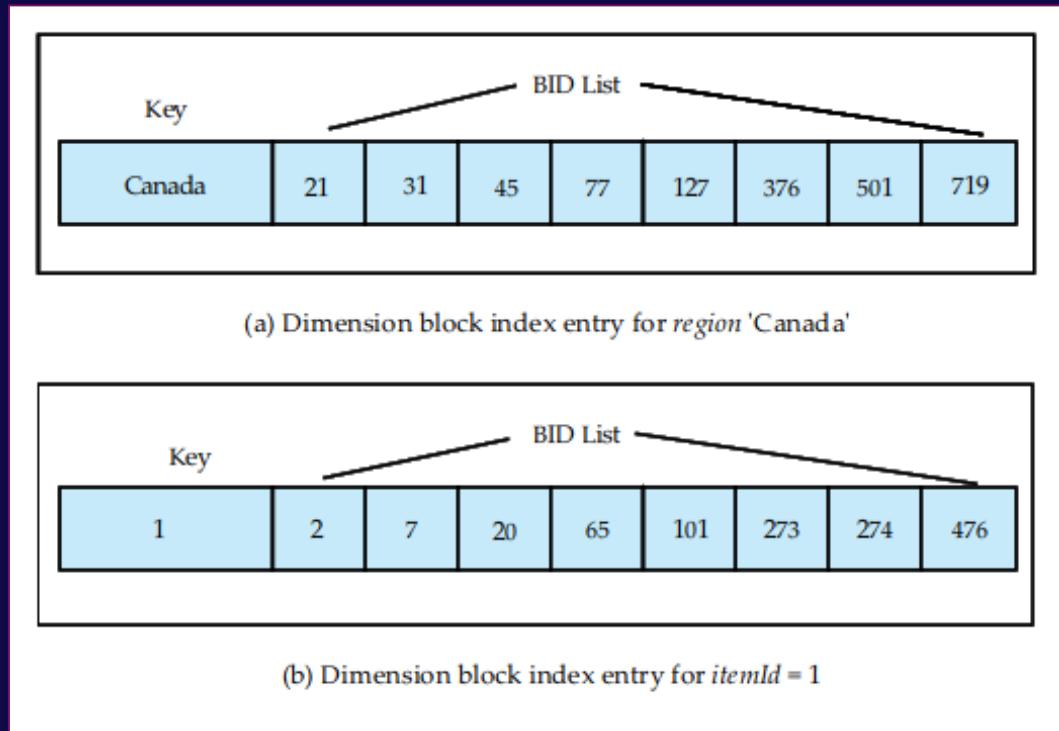


Figure 346 - Block index key entries

A block map is also associated with the table, recording the state of each block belonging to the table, which is used by the data-management layer to determine various processing options. Careful consideration must be given to selecting the right set of dimensions for **clustering a table** and the right **block size parameter** to minimize space utilization, with a focus on maximizing performance and maintenance benefits.

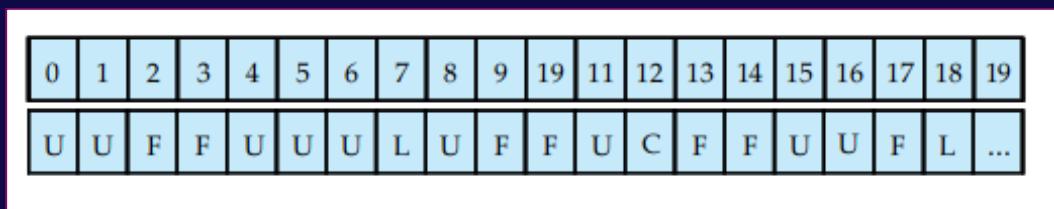


Figure 347 - Block map entries

Query Processing and Optimization

In the realm of query processing and optimization, IBM's DB2 has proven to be a titan of innovation. Its query compiler transforms complex SQL queries into an operator tree, which is then used at execution time for processing. With a rich set of query operators at its disposal, **DB2 flexes** its processing muscles, offering users the best strategies for executing even the most intricate tasks.

```
-- 'TPCD Local Supplier Volume Query (Q5)';
select n.name, sum(l.ExtendedPrice*(1-l.Discount)) as revenue
from tpcd.customer, tpcd.orders, tpcd.lineitem,
     tpcd.supplier, tpcd.nation, tpcd.region
where c.custkey = o.custkey and
      o.orderkey = l.orderkey and
      l.suppkey = s.suppkey and
      c.nationkey = s.nationkey and
      s.nationkey = n.nationkey and
      n.regionkey = r.regionkey and
      r.name = 'MIDDLE EAST' and
      o.orderdate >= date('1995-01-01') and
      o.orderdate < date('1995-01-01') + 1 year
group by n.name
order by revenue desc;
```

Figure 348 - Example

To illustrate this point, the Figures depict a representative complex query from the TPC-H benchmark, containing several joins and aggregations. Although the query plan selected for this example is relatively simple, DB2 provides users with various "explain" facilities, including a powerful visual explain feature in the Control Center that allows them to gain a better understanding of the details of a query-execution plan. With visual explanation, users can discern the cost and other relevant properties of the different operations of the query plan.

To illustrate this point, the Figures depict a representative complex query from the **TPC-H benchmark**, containing several joins and aggregations. Although the query plan selected for this example is relatively simple, DB2 provides users with various “*explain*” facilities, including a powerful visual explain feature in the Control Center that allows them to gain a better understanding of the details of a query-execution plan. With visual explanation, users can discern the cost and other relevant properties of the different operations of the query plan.

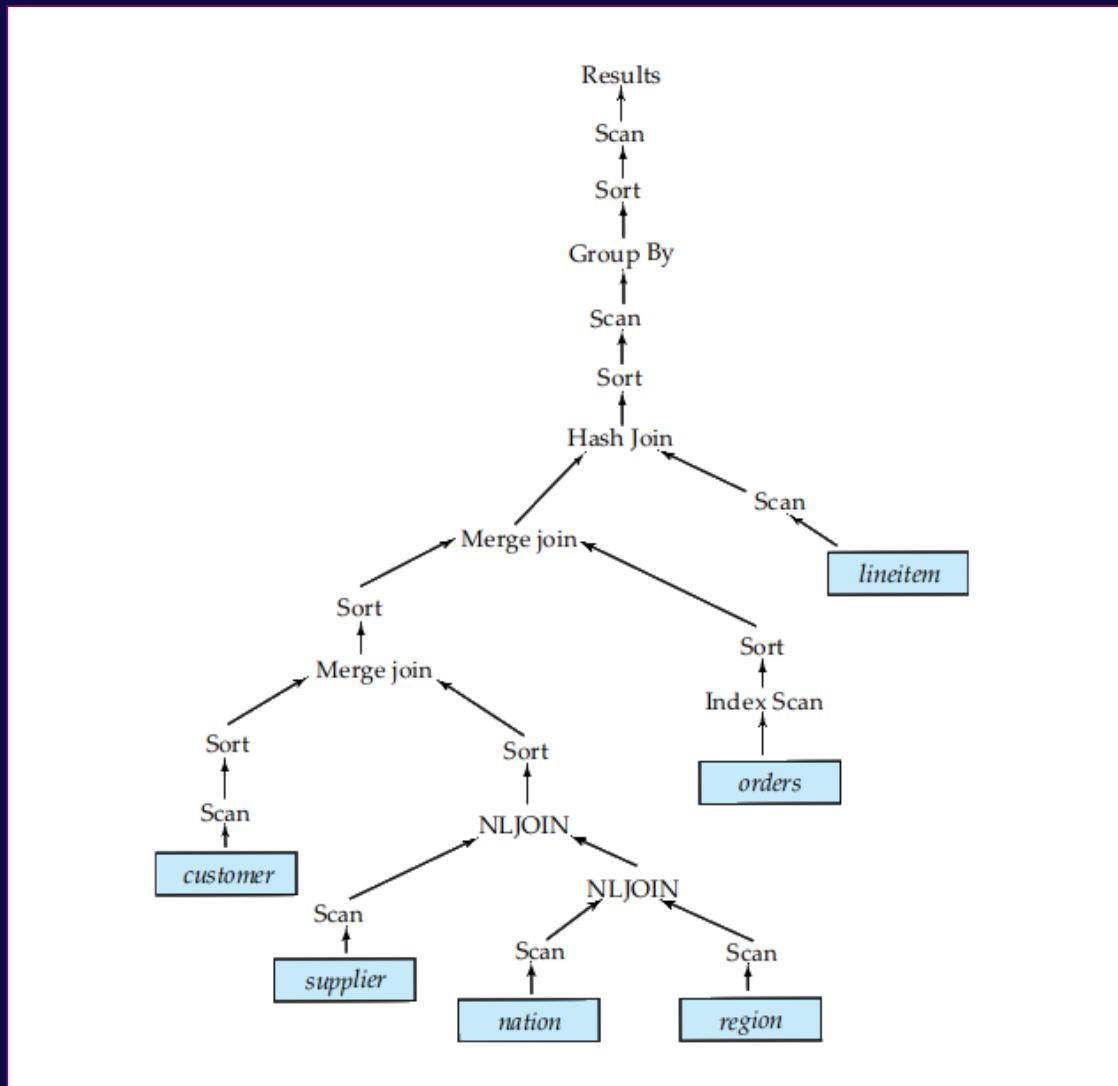


Figure 349 - DB2 query plan (graphical explain)

DB2's access methods for relational tables are equally comprehensive, including table scans, index scans, block index scans, index only, list prefetch, and block and record index ANDing and ordering.

These access methods manipulate records in database tables, with the base or leaf operators of the query tree performing the bulk of the work. Intermediate operations of the tree, such as join, set operations, and aggregation, comprise the meat of the processing. Meanwhile, the root of the tree produces the desired results of the query or SQL statement.

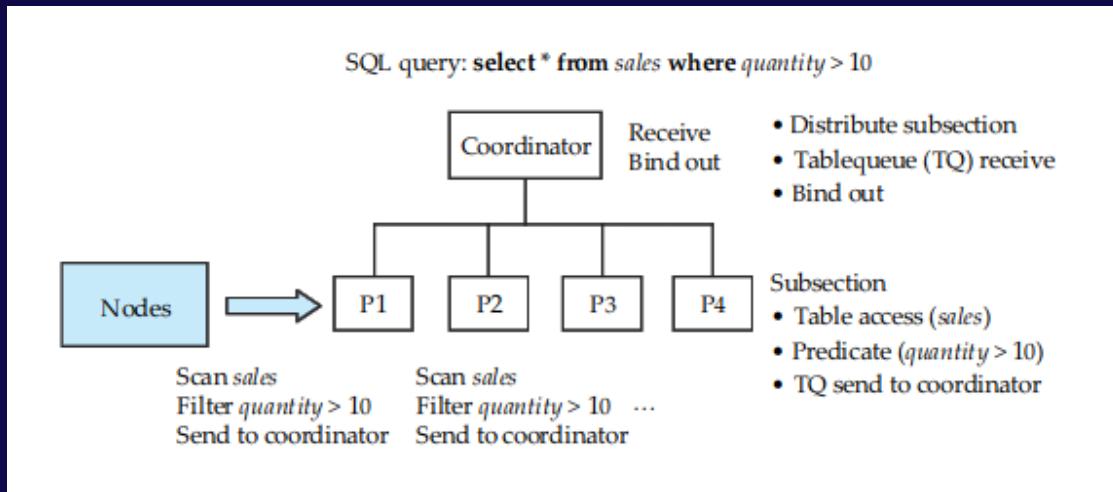


Figure 350 - DB2 MPP query processing using function shipping

When it comes to join, aggregation, and set operations, DB2 doesn't miss a beat. It offers **nested-loop**, **sort-merge**, and **hash-join techniques** for join operations, with each technique suited to different circumstances.

Set operations are implemented using sorting and merging techniques, with DB2 eliminating duplicates in the case of union, while duplicates are forwarded in the case of intersection.

```

create table emp.dept(dept.id integer, emp.id integer,
                      emp.name varchar(100), mgr.id integer) as
select dept.id, emp.id, emp.name, mgr.id
from employee, department
data initially deferred
refresh immediate -- (or deferred)
maintained by user -- (or system)
  
```

Figure 351 - DB2 materialized query tables

Materialized Query Tables

DB2, an eminent database management system, supports **materialized query tables (MQTs)** that enable faster query processing by maintaining a persistent copy of view data.

These MQTs can reference other MQTs, creating a highly scalable forest of dependent views. The beauty of MQTs lies in their seamless integration with the query-compiler infrastructure, which leverages the **internal QGM model** to match **input queries against available MQT definitions**, choosing the most suitable MQT reroute version for optimization.

To be valuable, MQTs must be maintained efficiently, and DB2 supports immediate and deferred maintenance options that offer time and cost flexibility. Immediate maintenance creates internal triggers to process updates to the dependent MQTs, while deferred maintenance requires an explicit refresh statement after moving updated tables into an integrity mode. The choice of incremental or full maintenance depends on the size of the MQT and the frequency of updates.

Choices	Incremental	Full
Immediate	Yes, After insert/update/delete	Usually no
Deferred	Yes, After load	Yes

Figure 352 - Options for MQT maintenance in DB2

DB2 provides users with the option to maintain MQTs explicitly, using SQL or utilities. For instance, after loading new data into one of the sources, the following commands perform deferred maintenance for the emp dept materialized view:

```
insert into employee;  
refresh table emp dept
```

MQTs are a powerful tool in DB2's arsenal, enhancing query processing capabilities and improving overall system performance.

Autonomic Features in DB2

IBM's DB2 UDB has introduced **autonomic features** that are set to revolutionize the field. Autonomic computing, a set of techniques that enable computing environments to manage themselves and reduce external dependencies in the face of changing circumstances, is at the forefront of these innovations. The benefits of autonomic computing are manifold, ranging from configuration to optimization, protection, and monitoring.

In this context, DB2's configuration and optimization capabilities are particularly noteworthy. The former provides automatic tuning for system configuration parameters such as buffer pool and sort heap sizes. DB2 monitors the system and modifies these parameters based on workload characteristics, gradually growing or shrinking heap memory sizes as needed.

- DB2's **Design Advisor tool** provides workload-based recommendations for these features, using optimization techniques to analyze a workload and suggest the most appropriate choices.
- The **Design Advisor command** syntax is straightforward and customizable, allowing users to specify options for materialized query tables, indices, clustering, and partitioning keys. The advisor uses the full power of DB2's query optimization framework and has full knowledge of the underlying schema and data statistics, using combinatorial techniques to identify the best indices, MQTs, MDCs, and partitioning keys for a given workload.
- **DB2's utility load-throttling mechanism** is a significant development in the field of load balancing. Given the trend towards online utilities, there is a need to balance the load consumption of utilities and prevent them from overwhelming the system and reducing user workload performance. The throttling technique is based on feedback control theory and continually adjusts and throttles the performance of the backup utility using specific control parameters.

Tools and Utilities

DB2, the relational database management system, is equipped with an array of tools to facilitate its use and administration.

The core set of tools is complemented by a multitude of vendor tools to enhance the user experience. The DB2 Control Center serves as the primary interface for managing DB2 databases, available across various workstation platforms. The interface is structured by data objects such as servers, databases, tables, and indices, offering task-oriented interfaces to execute commands and generate SQL scripts.

A snapshot of the main panel of the **Control Center** can be seen in the Figure below, showcasing a list of tables in the Sample database on node Crankarm. The Control Center boasts an extensive suite of component tools, including the command center, script center, journal, license management, alert center, performance monitor, visual explain, remote database management, storage management, and support for replication.

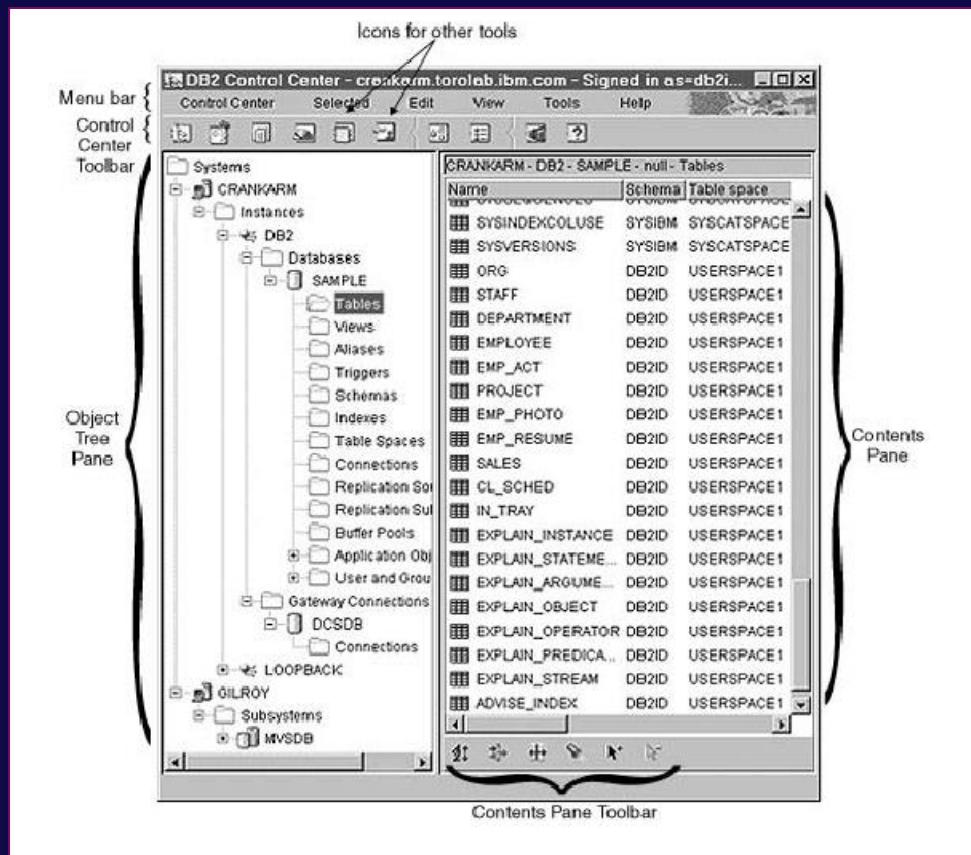


Figure 352 - DB2 Control Center

Notably, the command center allows users and administrators to issue database commands and SQL, while the script center facilitates running SQL scripts interactively or from a file. Additionally, the performance monitor allows monitoring of various events in the database system, and "*SmartGuides*" offer help in configuring parameters and setting up the DB2 system. Visual explain, an index wizard, and stored-procedure builder further streamline administration tasks, while the explained facility, explain tables, and graphically explain provide detailed breakdowns of query plans to users.

DB2 also provides direct access to most tools, such as **load**, **import**, **export**, **reorg**, **redistribute**, and other data-related utilities that support incremental and online processing capabilities. Administrators have access to a comprehensive set of tools, including an audit facility, governor facility, query patroller facility, trace and diagnostic facilities, and event monitoring facilities.

DB2 for OS/390 has a rich set of tools, including QMF, a widely used tool for generating ad hoc queries and integrating them into applications. With these tools, DB2 ensures ease of use and efficient administration for a seamless database experience.

Concurrency Control and Recovery

In the world of databases, concurrency control, and recovery are of utmost importance to ensure the integrity and availability of data. Fortunately, DB2 offers a robust set of techniques to handle these crucial aspects.

When it comes to concurrency and isolation, DB2 provides various modes such as repeatable read, read stability, cursor stability, and uncommitted read. **Record-level** and **table-level locks** are employed to implement these modes, and intent locks at the table level are utilized to maximize concurrency.

Moreover, next-key locking and variant schemes are implemented to mitigate Halloween and phantom-read problems.

Lock Mode	Objects	Interpretation
IN (intent none)	Tablespaces, tables	Read with no row locks
IS (intent share)	Tablespaces, tables	Read with row locks
NS (next key share)	Rows	Read locks for RS or CS isolation levels
S (share)	Rows, tables	Read lock
IX (intent exclusive)	Tablespaces, tables	Intend to update rows
SIX (share with intent exclusive)	Tables	No read locks on rows but X locks on updated rows
U (update)	Rows, tables	Update lock but allows others to read
NX (next-key exclusive)	Rows	Next key lock for inserts/deletes to prevent phantom reads during RR index scans
X (exclusive)	Rows, tables	Only uncommitted readers allowed
Z (superexclusive)	Tablespaces, tables	Complete exclusive access

Figure 353 - DB2 lock modes

In addition to locking mechanisms, DB2 offers commit and rollback functionality, with explicit commit and rollback statements available for applications to control the scope of transactions. The system also supports strict **ARIES logging** and **recovery schemes**, with write-ahead logging and two log modes (circular and archive) available to ensure data durability and facilitate recovery.

System Architecture

In the intricate world of DB2 servers, a complex network of processes and threads works in harmony to ensure efficient **data management**.

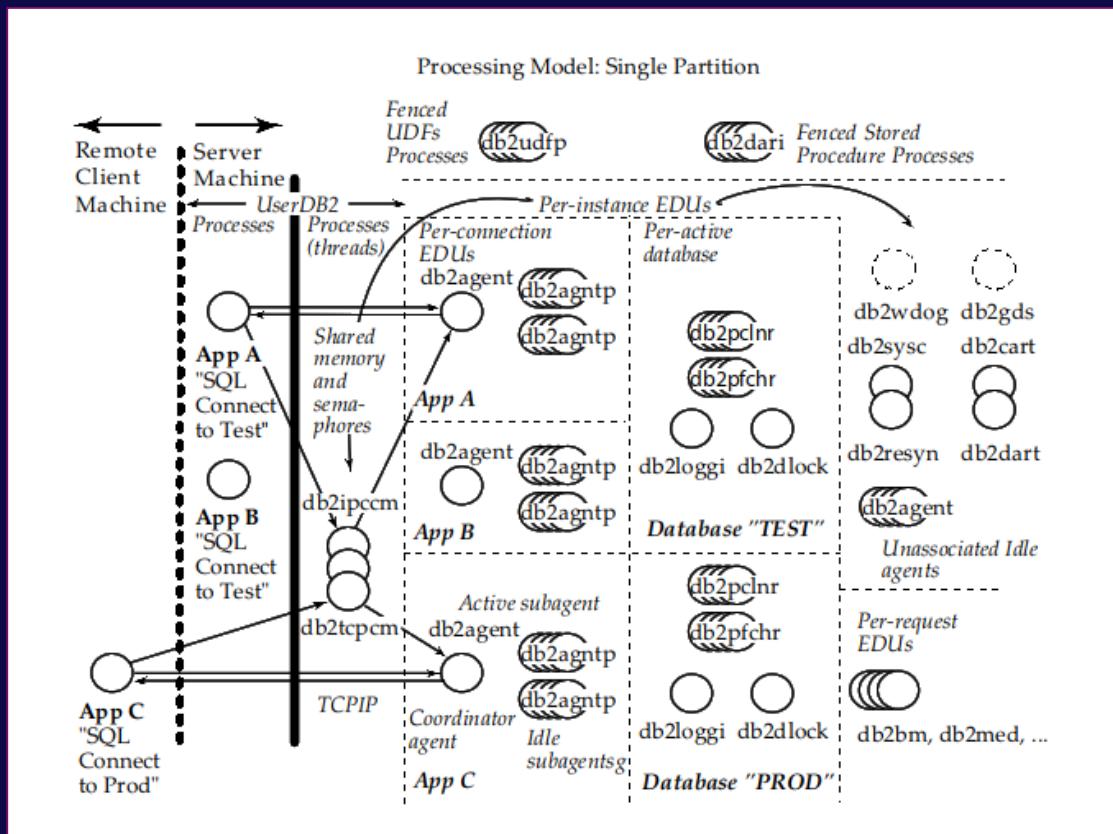


Figure 354 - DB2 lock modes

The figure above offers a glimpse into this intricate architecture, where **remote client applications** connect with the **server** via communication agents like *db2tcpccm*, and each application is assigned an agent, the *db2agent* thread. These agents, along with their subordinate agents, work in tandem to execute application-related tasks. Meanwhile, a separate set of threads performs tasks such as prefetching, page cleaning, logging, and deadlock detection for each database.

At the highest level, the server is equipped with a set of agents that perform vital tasks such as **crash detection**, **license server management**, and **control of system resources**. DB2 offers an array of configuration parameters to regulate the number of threads and processes in a server, and nearly all agent types can be controlled through these settings.

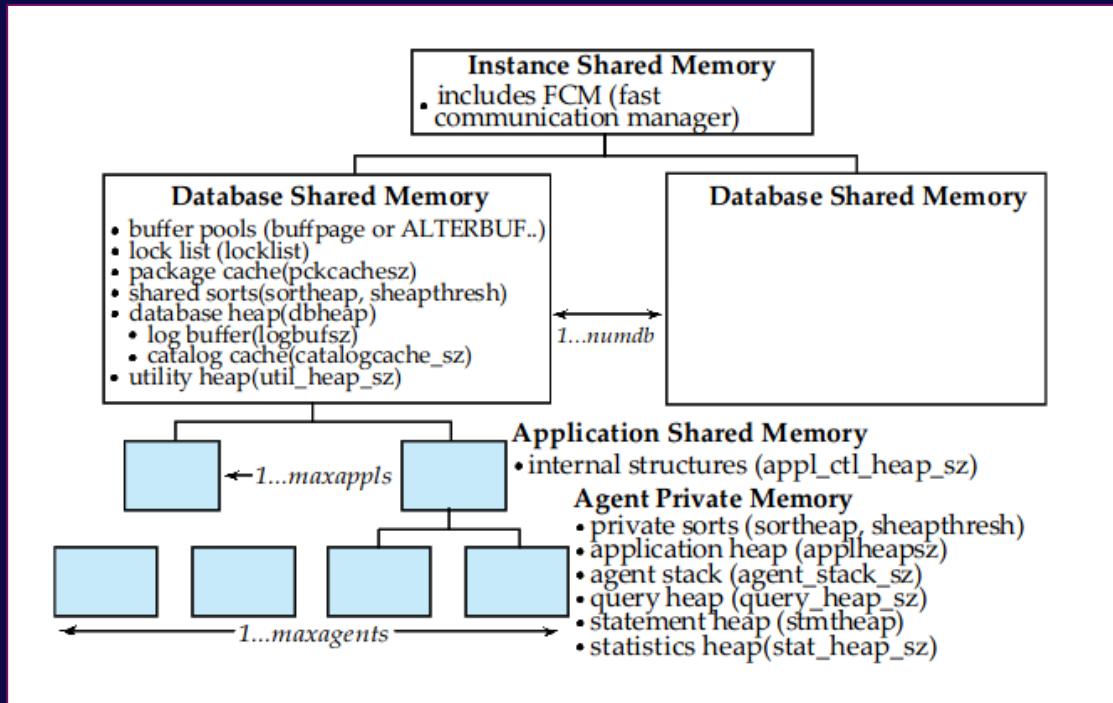


Figure 355 - DB2 memory model

The figure above provides a visual representation of the different memory segments in DB2, including private and shared memory.

Private memory is used for local variables and data structures, while shared memory contains essential data structures like the buffer pool, lock lists, application package caches, and shared sort areas. Multiple buffer pools can be created for a database using the "*create buffer pool*" statement and can be associated with tablespaces. However, the creation of buffer pools should only be undertaken after a careful analysis of workload requirements.

DB2 offers a vast array of memory configuration and tuning parameters to fine-tune the database's performance. From the default buffer pool to the sort heap, package cache, application-control heaps, and lock-list area, DB2 users can customize their memory settings to match their specific needs.

Replication, Distribution, and External Data

In the complex world of database management, DB2 offers a variety of tools for replication, distribution, and external data access.

DB2 Replication, a feature of the DB2 family, enables replication between various **DB2 relational** and **non-relational data sources**. The change-capture process is either log-based or trigger-based, and the captured changes are temporarily stored in staging tables before being applied to destination tables using SQL statements. Transformations can be applied to the intermediate tables, and the process is entirely managed by the administration facility.

Another noteworthy member of the DB2 family is the **DB2 Information Integrator**, which offers a range of capabilities including federation, replication, and search capabilities. The federated edition allows for remote DB2 or other relational databases to be integrated into a single distributed database, and non-relational data sources can be accessed using wrapper technology.

DB2 also provides support for user-defined table functions, enabling access to external data sources, and can participate in the **OLE DB protocols**. And when it comes to distributed transaction processing, DB2 has got you covered, offering full support for the two-phase commit protocol.

As a coordinator or participant, DB2 can interact with **any commercial distributed transaction manager**. In short, DB2 offers a comprehensive suite of tools for managing complex data environments.

Business Intelligence Features 1221 Bibliographical Notes

DB2 Data Warehouse Edition, a product in the illustrious DB2 family, offers a plethora of business intelligence features that elevate it above its peers. Its foundation is built on the rock-solid DB2 engine, which is augmented with state-of-the-art features for **Extract, Transform, Load (ETL)**, **Online Analytical Processing (OLAP)**, **mining**, and **online reporting**.

One of the key strengths of the DB2 engine is its **scalability**, which is achieved through the use of its **massively parallel processing (MPP) capabilities**. This allows DB2 to support configurations with several hundreds of nodes, even for large database sizes of several terabytes. Other features, such as **multi-dimensional clustering (MDC)** and **materialized query tables (MQT)**, provide support for the complex query processing requirements of business intelligence.

Online Analytical Processing (OLAP) is another important aspect of business intelligence, and the DB2 family includes a powerful feature called cube views that provides a mechanism for constructing data structures and MQTs within DB2. These can be used for relational OLAP processing and provide support for multidimensional cubes. **Cube views** can also take advantage of DB2's native support for cube-by and rollup operations, which are used to generate aggregated cubes. Additionally, cube views provide modeling support for mapping multidimensional cubes to a relational star schema, allowing for tight integration with OLAP vendors such as Business Objects, Microstrategy, and Cognos.

Multidimensional OLAP support is provided by the DB2 OLAP server, which creates a multi-dimensional data mart from an underlying DB2 database for analysis by OLAP techniques. This OLAP server uses the engine from the Essbase product for its OLAP operations.

DB2 Alphablox is another exciting feature of the Data Warehouse Edition, which provides online, interactive, reporting, and analysis capabilities. Its unique feature is the ability to construct new Web-based analysis forms rapidly, using a building block approach called blox. Finally, for advanced analytics, **DB2 Intelligent Miner** provides several components for modeling, scoring, and visualizing data. Its mining capabilities allow users to perform classification, prediction, clustering, segmentation, and association against large data sets, making it an essential tool for any business intelligence analyst.

10.4 Microsoft SQL Server

Microsoft SQL Server is a versatile relational database-management system that has made significant inroads in the computing world since its inception in the 1980s. Originally developed by Sybase for UNIX systems, the platform was later adopted by Microsoft for their **Windows NT systems**, and since 1994, has been released independently by the tech giant. The latest iteration, SQL Server 2008, has been localized in numerous languages and comes in express, standard, and enterprise editions, which cater to a broad range of users.

SQL Server's remarkable flexibility has helped it scale from laptops and desktops to enterprise servers, with a compatible version available for handheld devices such as Pocket PCs, SmartPhones, and Portable Media Centers. One of SQL Server's most significant advantages is its replication services, which enable the transfer of data between multiple copies of SQL Server as well as other database systems. The system also comes equipped with Analysis Services that include **online analytical processing (OLAP)** and **data-mining facilities**.

SQL Server's extensive array of graphical tools and wizards make database management tasks, such as backups and performance tuning, a breeze for administrators.

Additionally, many development environments, including **Microsoft's Visual Studio** and **.NET** products and services, support SQL Server, making it a top choice for programmers and developers alike. In summary, Microsoft SQL Server's comprehensive capabilities and user-friendly features have firmly established it as a key player in the database-management world.

Management, Design, and Querying Tools

In a world increasingly driven by data, the need for efficient and effective database management tools cannot be overstated. Enter SQL Server - a suite of tools designed to tackle every aspect of **SQL Server development, administration, and querying with ease and sophistication.**

From the visual aids provided by the **Database Designer**, **Table Designer**, and **View Designer**, to the powerful Query Editor that supports multiple languages and the SQL Profiler for performance optimization, SQL Server boasts a formidable arsenal of features to make the lives of database administrators and developers a whole lot easier.

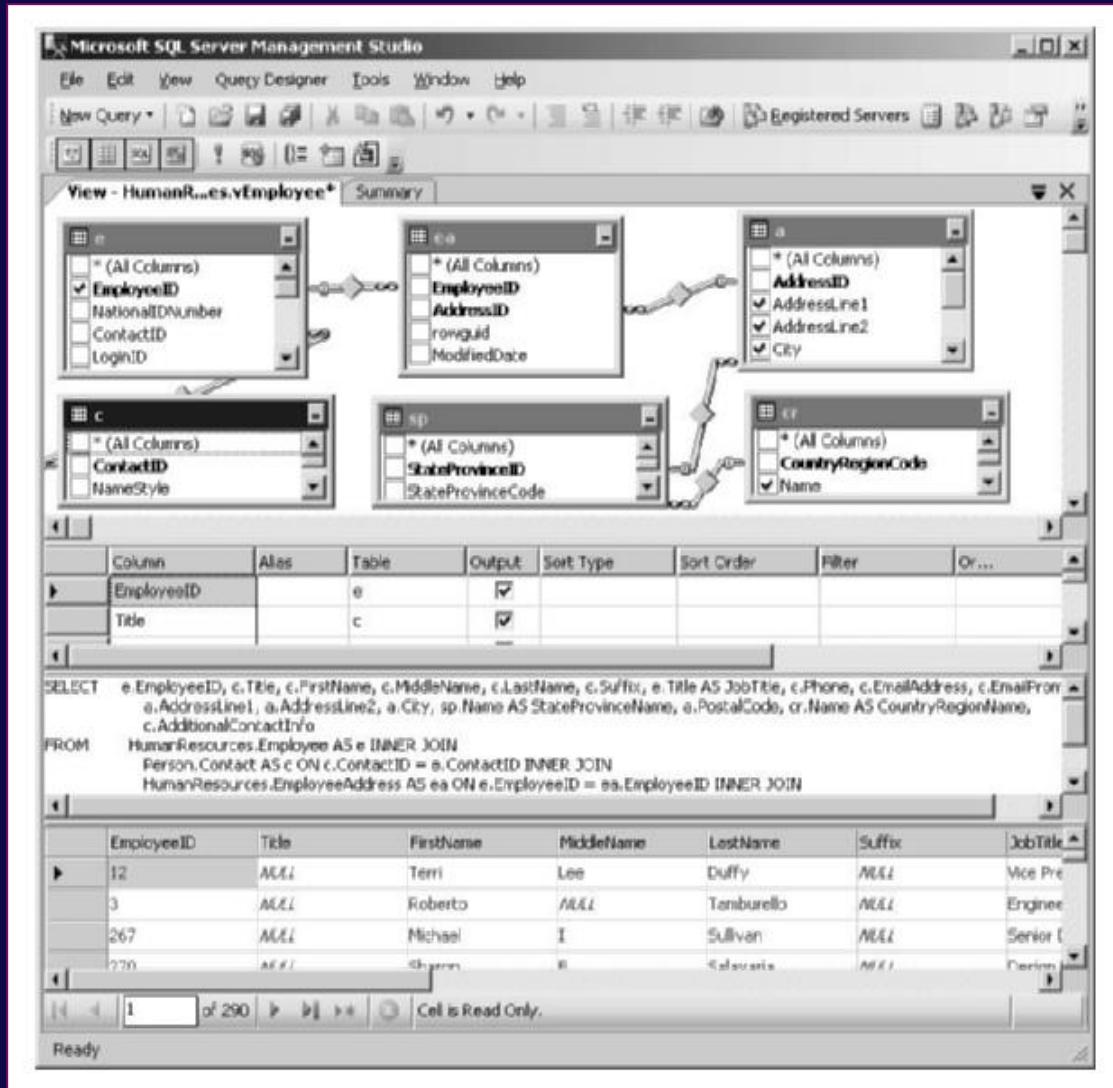


Figure 356 - The View Designer opened for the HumanResources.vEmployee view

The **Query Editor**, in particular, stands out for its sleek graphical interface and versatility. Its ability to generate graphical representations of showplan - the optimized execution steps - for queries, as well as provide detailed statistics regarding the resources required for execution, is simply awe-inspiring. With the Query Editor, administrators can analyze queries, format SQL queries, and even use templates for stored procedures and basic SQL statements.

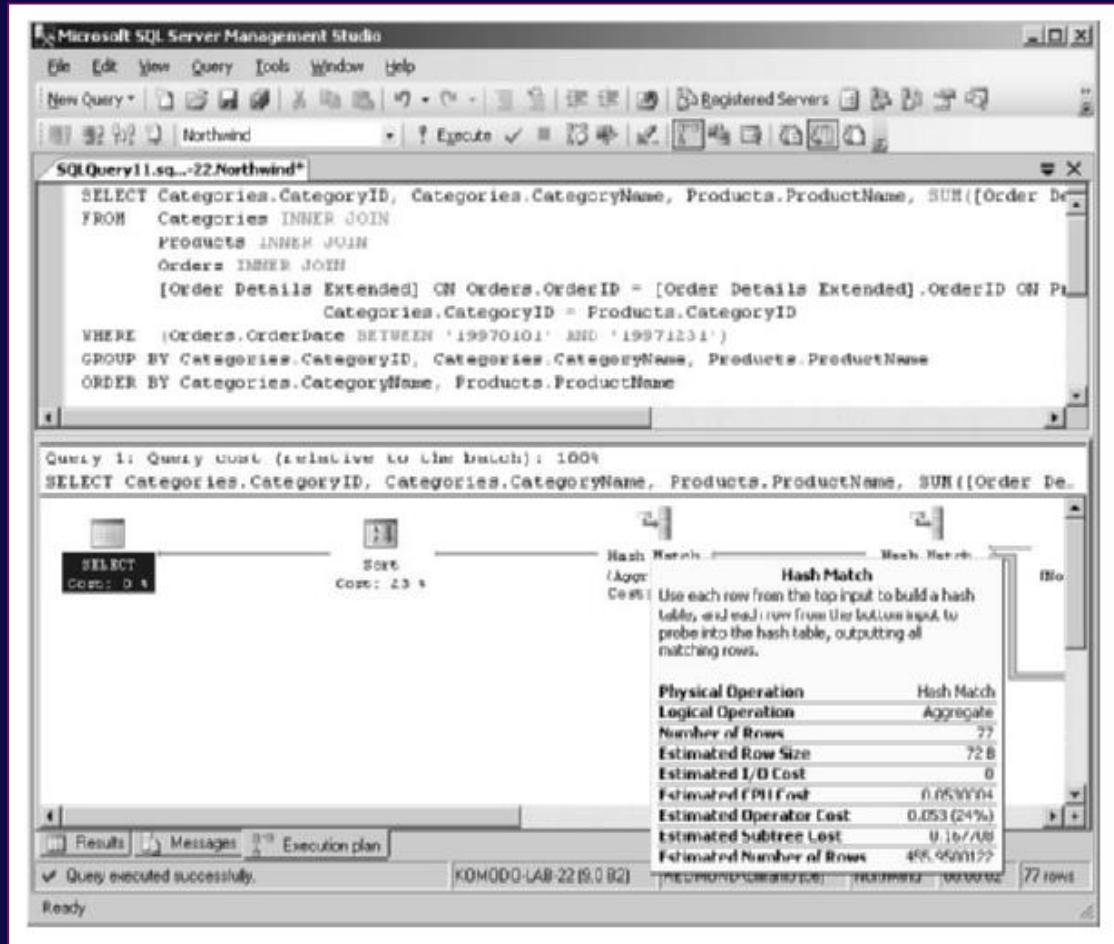


Figure 357 - A showplan for a four-table join with group by aggregation

SQL Profiler, on the other hand, serves as a powerful monitoring tool for recording and analyzing database activity, providing a real-time display of all server activity and performance data, including the time taken to execute queries, **CPU** and **I/O usage**, and the execution plans used.

With dozens of events and data items that can be captured, the **SQL Profiler** provides a wealth of information for administrators to delve deeper into database performance optimization.

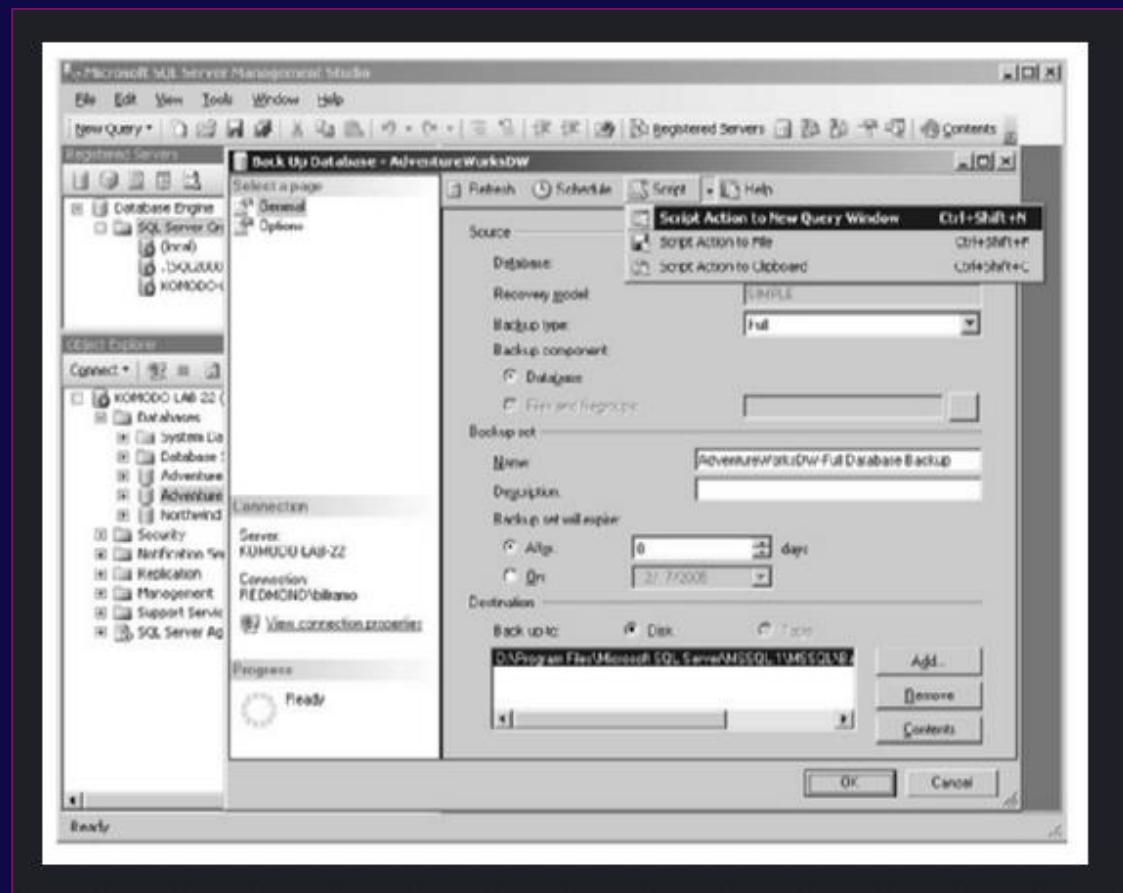


Figure 358 - The SQL Server Management Studio interface

The fact that SQL Server is designed to cater to a wide range of database environments, including **Transact-SQL**, **SQLCMD**, **MDX**, **DMX**, **XMLA**, and **SQL Server Mobile**, only adds to its appeal. Whether you're a seasoned database administrator or a developer just starting out, SQL Server is the go-to tool for managing and optimizing SQL Server environments, all within a sleek, unified interface.

SQL Variations and Extensions

In a world of dynamic and constantly evolving technology, SQL Server stands tall as an enabler for application developers to write server-side business logic with ease.

With its powerful and versatile **Transact-SQL programming language**, **data-definition** and **data-manipulation statements**, and iterative and conditional statements, it provides developers with a complete package that is compliant with most of the mandatory **DDL query** and **data modification** statements in the *SQL:2003* standard.

However, that's not all there is to SQL Server. It also supports numerous optional features such as recursive queries, common table expressions, user-defined functions, and relational operators, including intersect and except. The system even allows the use of **.NET programming languages** such as *C#*, *Visual Basic*, *COBOL*, or *J++* to write **server-side business logic**.

```
select *  
from MonthlySales pivot(sum(SalesQty)  
for Month in ('Jan', 'Feb', 'Mar'))  
T;
```

In addition to the mandatory features, SQL Server also supports many unique data types such as large character and binary string types, XML types, SQL variant, hierarchyId data type, and geospatial data types. These data types enable developers to store and manipulate hierarchical data, geospatial data, and data whose type cannot be anticipated at the data-definition time. SQL Server's ability to store and query geospatial data makes it an ideal platform for location-based applications.

```
select *  
from Departments D cross  
apply FindReports(D.ManagerID)
```

SQL Server also supports the use of table types and cursor types that cannot be used as columns in a table but can be used in Transact-SQL language as variables. A table type is primarily used to hold temporary results in a stored procedure or as the return value of a table-valued function, while a cursor type enables references to a cursor object.

```
update titleview  
set price = price *1.10  
where pub id = '0736';
```

SQL Server's query language enhancements include relational operators such as **pivot**, **unpivot**, and **apply**. The pivot operator transforms the shape of its input result set from two columns that represent **name-value pairs** into multiple columns, while the unpivot operator performs the inverse operation. The apply operator is similar to a join, except its right input is an expression that may contain references to columns in the left input, for example, a table-valued function invocation that takes as input a column from the left input. These query language enhancements provide developers with powerful tools to manipulate and analyze their data.

Storage and Indexing

In the realm of database management, the behemoth **SQL Server looms large**, its unparalleled functionality and versatility unmatched by its peers. Among its many features lies the concept of filegroups, which help streamline the organization and allocation of storage space within a database.

To optimize performance, filegroups are divided into fixed-size **8-kilobyte pages**, with allocation and deallocation of these pages managed by the allocation system in units of eight contiguous pages called extents. The allocation manager makes use of bitmaps to efficiently allocate and deallocate pages, thereby minimizing fragmentation and ensuring smooth scan performance.

SQL Server offers two modes for organizing tables: heap and clustered. A heap-organized table offers no control to the user in terms of row location, instead relying on a fixed identifier, the row **ID (RID)**, to find data. Conversely, in a **clustered-index organization**, rows are stored in a **B+-tree** sorted by the clustering key of the index. This key doubles as a unique identifier for each row and serves as a search structure to locate specific rows.

In addition to tables, SQL Server also supports secondary **B+-tree indices**, or **nonclustered indices**, which allow for queries that retrieve data from leaf-level pages without accessing the clustered index or heap. Furthermore, SQL Server permits the creation of computed columns, which are columns whose value is an expression based on other columns in that row, and which can also be indexed.

Range partitioning on tables and nonclustered indices is yet another feature of SQL Server, allowing for partitioned indices composed of **multiple B+-trees**, one per partition. With SQL Server's comprehensive array of features, it's no wonder it remains a top choice for database management.

Query Processing and Optimization

In the intricate world of SQL query processing and optimization, the quest for the perfect execution plan is a never-ending one. In pursuit of this elusive goal, the query processor of SQL Server employs a versatile and extensible framework, one that can easily incorporate new execution and optimization techniques. At the heart of this framework lies an **extended relational algebra**, which allows any SQL query to be expressed as a tree of operators. These operators are then abstracted into iterators, encapsulating data-processing algorithms as logical units that communicate with each other using a “**GetNextRow()**” interface. Starting with an initial query tree, the query optimizer generates alternatives by utilizing tree transformations and statistical models to estimate the cost of execution.

The optimization process is broken down into four key steps: **parsing/binding**, **simplification/normalization**, **cost-based optimization**, and **plan preparation**. During simplification, the optimizer applies transformation rules that guarantee the generation of less costly substitutes. It pushes selects down the operator tree, checks the predicate for contradictions, and uses them to identify subexpressions that can be removed from the tree. Reordering is a critical component of cost-based optimization. Unlike other commercial systems, SQL Server employs a **purely algebraic framework based on the Cascades optimizer prototype**. The optimizer applies exploration and implementation rules to generate alternatives, estimates execution costs, and chooses the plan with the cheapest anticipated cost.

SQL Server’s query optimizer is equipped with about **350 logical and physical transformation rules**, including inner-join reordering and reordering transformations for outer join, semijoin, and antisemijoin operators. Furthermore, **GbAgg**, the operator used for grouping and aggregation, is also reordered by moving it below or above joins when possible. **SQL Server’s query processor** is a marvel of engineering, employing a versatile and extensible framework to optimize SQL queries. The combination of tree transformations, **statistical models**, and **cost-based optimization** techniques ensures that the most efficient execution plan is selected, ultimately resulting in faster query processing and a better user experience.

Data analysis at optimization time is another area where **SQL Server** has made significant strides. The platform pioneered techniques to perform gathering of statistics as part of ongoing optimization. The computation of result size estimates is based on statistics for columns used in a given expression. These statistics consist of **max-diff histograms** on the column values and a number of counters that capture densities and row sizes, among others. Database administrators may create statistics explicitly by using **extended SQL syntax**. SQL Server has also implemented partial search and heuristics to address the issue of search-space explosion when applications issue queries involving dozens of tables. There are simple and complete transformations geared toward exhaustive optimization, as well as smart transformations that implement various heuristics. Smart transformations generate plans that are very far apart in the search space, while simple transformations explore neighborhoods. Optimization stages apply a mix of both kinds of transformations, first emphasizing smart transformations, and later transitioning to simple transformations. Optimum results on subtrees are preserved, so that later stages can take advantage of results generated earlier.

In terms of query execution, SQL Server supports **both sort-based** and **hash-based processing**, with data structures that are designed to optimize the use of processor cache. The platform's hash operations support basic aggregation and join, with a number of optimizations, extensions, and dynamic tuning for data skew. Additionally, SQL Server's flow-distinct operation is a variant of hash-distinct, where rows are output early, as soon as a new distinct value is found, instead of waiting to process the complete input.

SQL Server's innovative updates and techniques have made it a leader in the field of database optimization. By addressing issues like the Halloween problem, implementing partial search and heuristics, and optimizing query execution, SQL Server has ensured that users can access the data they need quickly and efficiently, no matter how complex their queries may be.

Concurrency and Recovery

SQL Server's intricate **transaction**, **logging**, **locking**, and **recovery subsystems** embody the hallmark ACID properties anticipated of a database system. In its pursuit of seamless data operations, **SQL Server offers** users the flexibility to specify varying levels of isolation for each statement, ensuring atomicity for all transactions. To achieve this, SQL Server utilizes locking, the default method for **concurrency control**. For additional optimization, SQL Server also offers optimistic concurrency control for cursors, a mechanism predicated on the supposition that conflict among multiple users is unlikely, thereby allowing transactions to execute without locking any resources. However, if a conflict does arise, SQL Server will detect it and necessitate the re-reading and change attempt of data by the application.

In line with industry standards, SQL Server supports multiple SQL isolation levels, including **read uncommitted**, **read committed**, **repeatable read**, and **serializable**. Read committed, the default level offers users the assurance that each statement executed within a transaction sees a transactionally consistent snapshot of the data as it existed at the start of the statement. Alternatively, SQL Server offers two snapshot-based isolation levels: **Snapshot**, specifying that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction, and **Read committed snapshot**, allowing each statement executed within a transaction to see a transactionally consistent snapshot of the data as it existed at the start of the statement.

Resource	Description
RID	Row identifier; used to lock a single row within a table
Key	Row lock within an index; protects key ranges in serializable transactions
Page	8-kilobyte table or index page
Extent	Contiguous group of eight data pages or index pages
Table	Entire table, including all data and indices
DB	Database

Figure 359 - Lockable resources

Moreover, locking forms the primary mechanism for enforcing the semantics of the isolation levels. SQL Server supports multi-granularity locking, which enables a transaction to lock different types of resources. The fundamental **lock modes**, **shared (S)**, **update (U)**, and **exclusive (X)**, are augmented by intent locks, which are used for multi-granularity locking.

Fine-granularity locking improves concurrency but requires additional **CPU cycles and memory** to acquire and hold many locks. To optimize lock granularity without sacrificing performance or concurrency, SQL Server automatically adjusts locking granularity and dynamically dedicates memory to the lock manager based on feedback from the system and other applications.

SQL Server has inbuilt mechanisms to automatically detect **deadlocks involving locks** and other resources, and it selects the least expensive transaction to roll back when a deadlock is detected. However, frequent deadlock detection may negatively impact system performance, which SQL Server counters by adjusting the frequency of deadlock detection. Ultimately, **SQL Server's concurrency** and **recovery features** position it as a database system that is both secure and efficient in its operations.

System Architecture

At the heart of **SQL Server's architecture** lies a thread pool that expertly manages client requests, internal background tasks, and parallel query plans. This pool reduces context switching and allows for precise **control over multiprogramming**, ensuring optimal performance and stability.

But it's not just the thread pool that makes SQL Server a top-tier player in the database world. Its memory management system is equally impressive, dynamically partitioning and redistributing memory among its many uses, from buffer pools to plan and execution caches.

SQL Server also boasts a comprehensive suite of **security features**, offering **authentication via both SQL Server-managed username-password pairs and Windows OS accounts**, as well as permission grants and audits for schema and container objects. **Encryption options**, including **Transparent Data Encryption**, ensure that sensitive data is kept secure.

With such a powerful system architecture, memory management, and security features, it's no wonder that SQL Server continues to be a leading player in the world of database management.

Data Access

SQL Server offers a wide range of **application programming interfaces (APIs)** to enable developers to create powerful data-driven applications. These APIs include *ODBC*, *OLE-DB*, *ADO.NET*, *LINQ*, *DB-Lib*, and *HTTP/SOAP*, each designed to support different programming languages and data access patterns.

ODBC is a standard *SQL:1999* call-level interface with object models like **Remote Data Objects** and **Data Access Objects** that simplify the programming of multitier database applications in languages like Visual Basic. **OLE-DB**, on the other hand, is a low-level, **systems-oriented API**, built according to the **Microsoft Component Object Model (COM)**, that encapsulates low-level database services such as rowset providers and query engines. It also includes a higher-level object model called **ActiveX Data Objects (ADO)** that simplifies database programming in Visual Basic.

ADO.NET is a modern **API designed** for *.NET languages* like *C#* and *Visual Basic.NET*, providing simplified data access patterns supported by *ODBC* and *OLE-DB*. Additionally, it introduces a new data set model that enables stateless, disconnected data access applications. **ADO.NET** also comes equipped with the **ADO.NET Entity Framework**, which allows developers to program against data at the conceptual (*entity*) level and reduces the impedance mismatch between applications and data services.

LINQ, or **Language-integrated query**, allows declarative, set-oriented constructs to be used directly in programming languages like *C#* and *Visual Basic*. It defines a set of standard query operators that allow traversal, filtering, joining, projection, sorting, and grouping operations to be expressed in a direct yet declarative way in any .NET-based programming language.

DB-Lib is a **C API** specifically developed for earlier versions of SQL Server that predate the *SQL-92* standard. Finally, SQL Server also supports *HTTP/SOAP* requests, allowing applications to invoke SQL Server queries and procedures through URLs that specify an instance of SQL Server. These requests can include *XPath* queries, *Transact-SQL* statements, or *XML* templates.

Distributed Heterogeneous Query Processing

Microsoft's SQL Server offers a powerful distributed heterogeneous query processing capability that enables transactions and updates against a diverse range of relational and nonrelational sources. Leveraging the **OLE-DB data providers**, these queries can be executed on one or more computers, enabling the server to access data sources that are external to its own system.

To reference these heterogeneous OLE-DB data sources, SQL Server provides two methods. The first is the **linked-server-names method**, which associates a server name with an OLE-DB data source using **system-stored procedures**. Objects in these linked servers can then be referenced in Transact-SQL statements using the four-part name convention. For instance, if a linked server name of *DeptSQLSrvr* is defined against another copy of SQL Server, the following statement references a table on that server:

```
from DeptSQLSrvr.Northwind.dbo.Employees;
```

Furthermore, an **OLE-DB data source** is registered in SQL Server as a linked server, which can be accessed using the four-part name. The following example establishes a linked server to an Oracle server via an OLE-DB provider for Oracle:

```
select e.dept, f.DocAuthor,
       f.FileName
  from OraSvr.Corp.Admin.Employee e,
       openquery(EmpFiles,
                  'select DocAuthor, FileName
                   from scope("c:\EmpDocs")
                   where contains('' '' "Data" near() "Access" ' '')>0') as f
     where e.name = f.DocAuthor
  order by e.dept, f.DocAuthor;
```

In addition, SQL Server supports `openrowset` and `open query`, two built-in parameterized table-valued functions that enable sending uninterpreted queries to a provider or linked server in the dialect supported by the provider. These functions can be combined with relational operations such as **joins**, **restrictions**, **projections**, **sorts**, and **groups by operations**.

Replication

SQL Server replication is a versatile set of technologies that facilitates data distribution and synchronization between databases, enabling replication of changes and maintaining consistency. With inline replication of most database schema changes, it ensures minimal interruptions or reconfigurations, enhancing the application performance by scaling out for improved total read performance among replicas.

The **Publish-Subscribe metaphor** forms the backbone of SQL Server's replication model, with publishers being servers that make data available for replication to other servers. Subscribers, on the other hand, receive replicated data from publishers and can conveniently subscribe to only the publications they require. The addition of an article to a publication allows for extensive customizing of the way the object is replicated, catering to the enterprise's requirements.

The distributor is a server that acts as a repository for history and error state information, and in other cases, it is used as an intermediate store-and-forward queue to scale up the delivery of the replicated payload to all the subscribers.

With **Microsoft SQL Server replication**, there are various options available to determine the application's needs, such as snapshot replication, transactional replication, and merge replication. Snapshot replication is best suited for smaller data sizes and when updates typically affect enough of the data that replicating a complete refresh of the data is efficient. **Transactional replication propagates** an initial snapshot of data to subscribers, forwards incremental data modifications to subscribers, and preserves intermediate states between multiple updates. Merge replication enables each replica in the enterprise to work with total autonomy whether online or offline, tracking metadata on the changes to published objects and ensuring data convergence through automatic conflict detection and resolution.

SQL Server replication allows for flexible replication topologies, catering to the enterprise's requirements and scaling up the delivery of the replicated payload to all the subscribers.

Server Programming in .NET

In an increasingly interconnected world, the need for efficient and effective data processing has become paramount. Enter SQL Server, a powerful tool that offers the ability to run the **.NET Common Language Runtime (CLR)** inside its process. This feature allows database programmers to write business logic as functions, stored procedures, triggers, data types, and aggregates. This unique capability affords application architectures that require business logic to execute close to the data, the luxury of not having to ship data to a middle-tier process to perform computation outside the database, thereby improving overall efficiency and cutting costs.

The CLR, a runtime environment with a strongly typed intermediate language, executes modern programming languages like C#, *Visual Basic*, C++, COBOL, and J++, among others, and has garbage-collected memory, preemptive threading, metadata services (*type reflection*), code verifiability, and code access security. The runtime leverages metadata to locate and load classes, resolve method invocations, enforce security, and set runtime context boundaries.

Assemblies are the units of packaging, deployment, and versioning of application code in .NET, and the preferred method of deploying application code inside the database. Once registered inside the database, users can expose entry points within the assembly via SQL DDL statements, which can act as scalar or table functions, procedures, triggers, types, and aggregates, by using well-defined extensibility contracts enforced during the execution of these DDL statements.

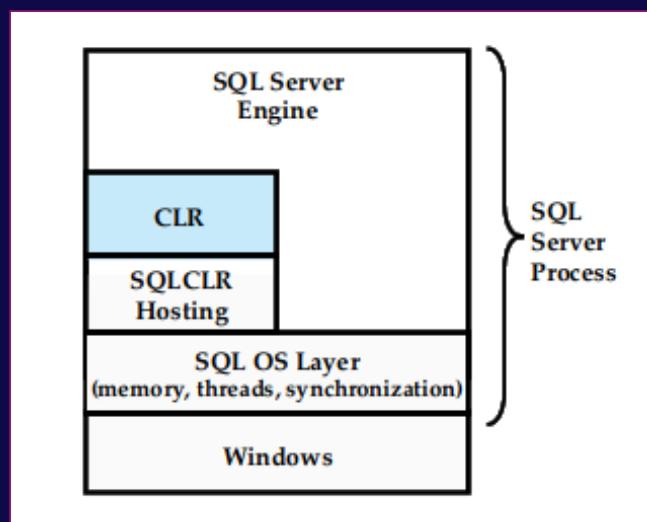


Figure 360 - Integration of CLR with SQL Server operating-system services

The **.NET framework supports** an out-of-band mechanism called custom attributes for annotating classes, properties, functions, and methods with additional information or facets the application may want to capture in metadata. All **.NET compilers** consume these annotations without interpretation and store them in the assembly's metadata. All these annotations can be examined in the same way as any other metadata by using a common set of reflection APIs.

While SQL Server and the CLR are two different runtimes with **different internal models for threading, scheduling, and memory management**, SQL Server solves the integration challenge by becoming the operating system for the CLR when it is hosted inside the SQL Server process. This approach provides common threading, scheduling, and synchronization, allowing for seamless scalability and reliability. With SQL Server, the possibilities for efficient and effective data processing are endless.

XML Support

In recent years, relational database systems have wholeheartedly *embraced XML*, utilizing it in a variety of ways. Initially, the primary focus of *XML* support in these systems was to export relational data as *XML* and to import relational data in *XML* markup form back into a relational representation. However, the main usage scenario for these systems is information exchange in contexts where *XML* is used as the “*wire format*” and where the relational and *XML* schemas are typically predefined independently of each other.

To accommodate this scenario, **Microsoft SQL Server** has implemented extensive functionality, such as the *for XML* publishing rowset aggregator, the **OpenXML rowset** provider, and the *XML* view technology based on annotated schemas. However, shredding *XML* data into a relational schema can be quite difficult or inefficient for storing semistructured data whose structure may vary over time and for storing documents.

```
select Report.query('
    declare namespace c = "urn:example/customer";
    for $cust in /c:doc/c:customer
        where $cust/c:notes//c:saleslead
    return
{
    $cust/c:name,
    $cust/c:notes//c:saleslead
}')
from TripReports;
```

In order to support such applications, SQL Server has implemented native *XML* based on the *SQL:2003 XML* data type. This includes the ability to store *XML* natively, to constrain and type the stored *XML* data with collections of *XML* schemas, and to query and update the *XML* data. To ensure efficient query execution, several types of *XML*-specific indices are provided.

Finally, the native XML support also integrates with the “*shredding*” and “*publishing*” of relational data.

```
update TripReports  
set Report.modify(  
'declare namespace c = "urn:example/customer";  
delete /c:doc/c:customer//c:saleslead[@year < sql:variable("@year")]');
```

SQL Server’s XML data type can store *XML* documents and content fragments and is defined on the basis of the *XQuery 1.0/XPath 2.0 data model*. SQL Server stores data of type XML in an internal binary format as a blob and provides indexing mechanisms for executing queries. Additionally, the internal binary format provides efficient retrieval and reconstruction of the original XML document, along with space savings of up to **20%**.

SQL Server provides several *XQuery-based query* and modification capabilities on the XML data type, including the query method, the value method, the exist method, and the modify method. These methods allow for querying and modifying XML data with precision and accuracy.

SQL Server’s implementation of native XML support provides users with powerful and efficient tools for storing, organizing, querying, and modifying XML data in a relational database system.

SQL Server Service Broker

In the vast realm of database technology, **Service Broker** stands tall as a true marvel of modern engineering. Its ability to facilitate loosely coupled distributed applications is unmatched, owing to its support for reliable, queued messaging in SQL Server.

Service Broker is a godsend for applications that leverage asynchronous processing to enhance scalability and responsiveness in interactive sessions. Unlike traditional messaging systems, Service Broker adopts a **conversation-based** approach to communication. It ensures that each conversation is a **persistent, reliable, full-duplex** stream of messages that is delivered to an application precisely once and in order. To further enhance its capabilities, Service Broker permits the assignment of **priority levels** ranging from **1** to **10** to individual conversations, with higher priority conversations given expedited attention.

Service Broker's architecture is underpinned by a concept of contracts that strictly defines allowable message types for each conversation. SQL Server provides default contracts and message types for basic reliable stream applications. Moreover, to enable **efficient message ordering, related message correlation, and locking**, Service Broker employs internal tables that are not directly accessible. Instead, the system uses queues as views of those internal tables, which applications can receive messages from. By enforcing locking at the conversation group level, Service Broker eliminates the need for applications to include deadlock-avoidance logic, significantly reducing the risk of errors in messaging applications.

Furthermore, **Service Broker** boasts a highly efficient activation mechanism that automatically triggers stored procedures when messages are available for processing. By monitoring queue activity, **Service Broker scales** the number of stored procedures to match the incoming traffic. For applications requiring more sophisticated logic, **SQL Server** provides **External Activator**. This feature enables an application outside of SQL Server to be activated when new messages are added to a queue, reducing the need for **CPU-intensive tasks** to be performed within SQL Server.

Service Broker's end-to-end encryption capabilities for messaging between SQL Server instances make it a valuable asset to developers. The system uses routing to map service names to network addresses, thus avoiding the need for alterations to the queue readers. **SQL Server's load balancing** and **message forwarding services** are also game changers for applications that require reliable, exactly-once-in-order delivery. And with local and remote conversations using the same programming model, Service Broker is the epitome of versatility in database messaging technology.

Business Intelligence

In the world of modern data analysis, the SQL Server's business intelligence component stands out as a true juggernaut, composed of three distinct subcomponents: *Integration Services*, *Analysis Services*, and *Reporting Services*. Each of these subcomponents is deployed on separate servers and can be installed independently, either on the same machine or different ones. Together, they provide an all-encompassing solution for extracting, transforming, and loading data, modeling it, and finally producing and disseminating analytical reports.

At the forefront of these subcomponents is **Integration Services**, which stands as a versatile and robust enterprise data transformation and integration solution. Using **SSIS**, one can extract, aggregate, and consolidate data from disparate sources and transport it to single or multiple destinations, refresh data in **data warehouses** and **data marts**, **cleanse data** before its loading, automate administrative functions, and much more. With its impressive array of services, graphical tools, programmable objects, and APIs, users can craft complex data transformation solutions without resorting to custom programming.

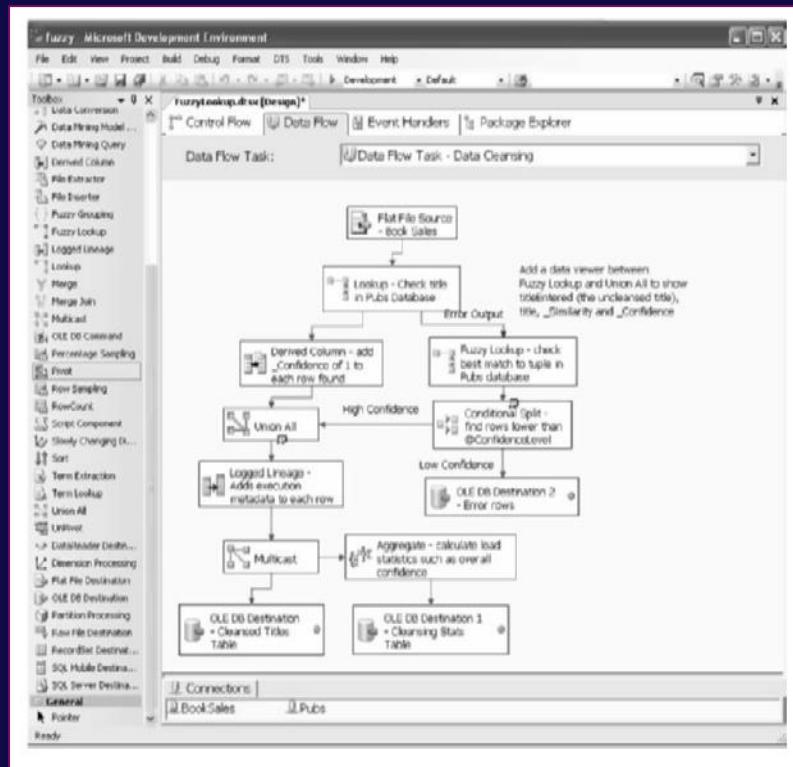


Figure 361 - Loading of data by using fuzzy lookup

Analysis Services, on the other hand, provides **online analytical processing (OLAP)** and **data-mining capabilities** for business intelligence applications. Supporting a thin client architecture, this subcomponent boasts a **Unified Dimensional Model (UDM)** that bridges the gap between traditional relational reporting and *OLAP ad hoc analysis*. Using the UDM, users can generate reports in their native language, define business-oriented perspectives, and measure key performance indicators, all while navigating and querying complex, multidimensional data.

Reporting Services provides users with the tools to generate insightful and customizable reports, providing a comprehensive view of the data stored in the SQL Server. Each of these subcomponents, while distinct in their capabilities and strengths, can be integrated and leveraged in combination with one another, leading to a rich and powerful solution for data analysis, transformation, and presentation. The SQL Server's business intelligence component stands tall as a testament to the power and possibilities of modern data science.