IIM-I340 Course Pack
Database Systems

Gregory D. Weber

Version 2.2.0.1
January 16, 2010
## CONTENTS

3.4 Introduction to the SELECT Statement .......................... 41
3.5 Basic Set Operations ........................................... 54
3.6 Aggregation ...................................................... 55
3.7 Nulls .............................................................. 58
3.8 Additional Operations on Subqueries ........................... 59
3.9 Views ............................................................. 63
3.10 Insertion ......................................................... 66
3.11 Updates ......................................................... 68
3.12 Deletion ......................................................... 69
3.13 The Case Expression .......................................... 70
3.14 Transactions .................................................... 72
3.15 Joins Revisited ................................................ 72

4 Advanced SQL ..................................................... 79
4.1 More SQL Data Types ........................................... 79
4.2 Tables ........................................................... 80
4.3 Integrity .......................................................... 82
4.4 SQL Programming Interfaces ................................. 89

5 Other Database Languages ...................................... 93
5.1 Overview ......................................................... 93
5.2 Logic ............................................................ 93
5.3 The Tuple Relational Calculus ................................. 96
5.4 The Domain Relational Calculus .............................. 97
5.5 Comparisons .................................................... 98
5.6 Query By Example (QBE) ..................................... 99
5.7 Datalog .......................................................... 101

6 The Entity-Relationship Model .................................. 107
6.1 Introduction ..................................................... 107
6.2 Basic Concepts ................................................ 107
6.3 Constraints ...................................................... 111
6.4 Keys ............................................................. 113
6.5 Exercises ......................................................... 114
6.6 Strong and Weak Entity Sets ................................ 114
6.7 Specialization .................................................. 116
6.8 Reducing an E-R Schema to Tables ......................... 116
6.9 Unified Modeling Language (UML) .......................... 120

7 Relational Database Design ..................................... 121
7.1 Outline ........................................................ 121
7.2 Overview ....................................................... 121
7.3 Functional Dependency ....................................... 121
7.4 Normal Forms ................................................ 125
## 8 Application Design and Development

8.1 Outline ............................................. 129
8.2 Form and Report Generator Tools .................. 129
8.3 Databases and the World-Wide Web ................. 130
8.4 Triggers .............................................. 131
8.5 Security .............................................. 132
8.6 Ethical Responsibilities .............................. 137

## 9 Physical Design of Databases

9.1 Storage Hierarchy .................................... 139
9.2 Disk Organization .................................... 139
9.3 Time for Disk I/O ................................. 140
9.4 Indexing Overview ................................. 141
9.5 Motivation ............................................ 141
9.6 Kinds of Indexes .................................... 142
9.7 Tree Indexes ......................................... 142
9.8 Hash Indexes ......................................... 142
9.9 SQL Statements ..................................... 143

## 10 Transactions

10.1 Outline ............................................... 145
10.2 The Concept of Transaction ......................... 145
10.3 States of a Transaction ............................. 146
10.4 Transactions in SQL ............................... 147

## 11 Data Analysis and Mining

11.1 Introduction ......................................... 151
11.2 Online Analytical Processing (OLAP) ............... 152
11.3 Data Warehousing .................................. 153
11.4 Data Mining ......................................... 154
Chapter 1

Introduction to Databases


1.1 Fundamental Concepts

**Question:** What is a database?

**Answer:** Roughly speaking, a database is a collection of data. We will refine this concept.

**Question:** What is a database management system (DBMS)?

**Answer:** The combination of a database with a collection of programs that provide convenient and efficient means to accessing the database, i.e., retrieving and storing information.

**Question:** (For discussion) Give examples of database systems that you have used or worked with (not DBMS products, such as Oracle or Microsoft SQL Server, but applications such as airline reservations, hospital database, personnel database). What were the kinds of information stored in the database, as far as you knew? What were the operations that could be performed on the database, in terms of retrieving, storing, and updating information?

**Question:** What is a file-based information system?

**Answer:** In a file-based system, there is a collection of files and a collection of programs that access the files. E.g., a book dealer might have an inventory file listing the authors, titles, publishers, selling price, and quantity on hand of books; a customer order file that lists the names and addresses of customers and the books they have ordered but not received; a customer account file that lists the names and addresses of customers and their account history and balance. Programs might include a “sell” program which updates the inventory, customer order, and customer account files; a “ship” program which updates the customer order file; and a “receive payment” program which updates the customer account file.
1.2 Advantages and Disadvantages of DBMS

Question: What kinds of problems associated with file-based systems are database systems designed to overcome or avoid?

Answer:

Redundant data. If a customer moves, two files have to be updated in the file-based system. If only one is updated, the company will have inconsistent data. A database system is designed to keep (from a logical point of view) only one copy of the customer address information, so that only one update is needed.

Complex data access. Access to data is through programs, and if there is no program to retrieve the desired information, a new program must generally be written. Database systems provide powerful query languages which make it relatively easy to retrieve information in ways that were not planned for.

Data integrity problems. There are usually rules about the valid values of data. E.g., there may be a rule that says no customer’s order can amount to more than $1000. The burden of enforcing these rules is on the application programmer. It’s easy to forget, of course. In a database system, once the rules have been defined, the database system itself takes on the burden of enforcing them.

Non-atomic transactions. If a customer orders the book, updates to three files (the customer account file, customer order file, and inventory file) are required. What if the system crashes after one of the files has been updated, but before the others? The files will be left in an inconsistent state. E.g., the customer’s balance will reflect purchases that never get removed from inventory and shipped. Database systems guarantee that, after recovery from a system crash, either all or none of the effects of the transaction will be reflected in the database.

Problems with concurrent access. Suppose two customer order clerks simultaneously handle orders by different customers for the same book, and there are 10 copies of the book on hand. Depending on the exact sequence of operations, the file system could be left in a state that says we have 9 copies left. (How could this happen?) A database system guarantees that both updates are made correctly.

Coarse-grained security. The operating system of a computer typically can limit the access of users to files (read, write, execute), but if a user is able to read (write) a file at all, the user is able to read (write) the entire file. This is often undesirable. For example, shipping personnel need to know the address of a customer, but not the customer’s account balance. Database systems provide finer-grained control of access to information.
1.3 Abstractions of Database Systems

**Question:** What important levels of abstraction are there in database systems?

**Answer:**

The **logical level** describes what (kinds of) data are stored in a database and what are the relationships between these data. For example, the database will store customer addresses, and every customer has exactly one address.

The **physical level** can get much more detailed, involving issues such as the storage media (magnetic disk, RAID, optical disk, tape, etc.), the indexing structures (hash tables, etc.).

The **view level.** Views are further abstractions of the logical level. Views permit users to see or modify only the parts of the database that are relevant to their work. For example, the view of a shipping clerk would show customer’s addresses but not their account balances.

1.4 Schemas and Instances

**Question:** What are schemas and instances?

**Answer:**

The **schema** of a database is its overall design. Over a period of time, the specific data values stored in the database typically change even while the design remains the same. The collection of specific values in the database at a particular time is called an **instance** of the database.

Analogies: (1) the species reindeer, an individual reindeer (Rudolph); (2) in a programming language: a variable of a particular type, a value of this variable at a particular time.

Since we can think about databases at various levels of abstraction, we can distinguish between **logical schemas** and **physical schemas** as well as schemas of views.

The terms **schema** and **instance** can be applied to the parts of the database as well as the whole. For example, the book dealer’s database might include a table of customers (i.e., a collection of customer data) with the logical schema

\[
\text{customer} = (\text{name, address})
\]

The table could contain instances such as

- (Charles Smith, 123 Main Street)
- (Aggie Brown, 421 Fifth Avenue)
CHAPTER 1. INTRODUCTION TO DATABASES

Customer table:

<table>
<thead>
<tr>
<th>CustID</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>James</td>
<td>10 Main St.</td>
</tr>
<tr>
<td>2</td>
<td>Jean</td>
<td>221 First Ave.</td>
</tr>
</tbody>
</table>

Book table:

<table>
<thead>
<tr>
<th>BookID</th>
<th>Author</th>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joyce</td>
<td>Dubliners</td>
<td>10.00</td>
</tr>
<tr>
<td>2</td>
<td>Austen</td>
<td>Pride and Prejudice</td>
<td>15.00</td>
</tr>
<tr>
<td>3</td>
<td>Faulkner</td>
<td>As I Lay Dying</td>
<td>18.00</td>
</tr>
</tbody>
</table>

Customer-orders-book table:

<table>
<thead>
<tr>
<th>CustID</th>
<th>BookID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 1.1: Some tables for the book dealer database.

1.5 Data Models

QUESTION: What is a data model?

ANSWER: A data model is a formal, conceptual framework (Silberschatz, Korth, and Sudarshan, p. 8: a “collection of conceptual tools”) that we can use to describe data, relationships between data, integrity rules for data, and so on.

The same data model can be used to think about many different databases (database schemas), for example, bookseller databases, hospital databases, airline flight databases, just as the physicist could use the same model of planetary motion—the same theories of gravitation, etc.—to describe the motions of any planets, whether they are orbiting the sun or another star.

Two important data models are the relational model and the entity-relationship (E-R) model.

The Relational Data Model

QUESTION: What are the characteristics of the relational model?

ANSWER: The relational model represents a database as a collection of tables, formally called relations. Examples of tables are shown in Silberschatz, Korth, and Sudarshan, p. 12.

Example: Figure 1.1 shows part of the book dealer database.

We will study the relational model in detail as a formal algebra, which forms the basis of query languages such as SQL.
The relational model is, in fact, the foundation of nearly all modern database systems.

**The Entity-Relationship Data Model**

**QUESTION:** What are the characteristics of the entity-relationship model?

**ANSWER:** The E-R model is a tool used for designing databases, based on the realization that the world consists of entities and their relationships. An *entity* is something that exists—a thing or a person. Entities are characterized by their *attributes* (properties). A *relationship* involves the interconnection between two or more entities.

Examples:
- Bob is a person (entity). His height is 6 feet (attribute). He is married to Deborah (relationship).
- The Empire State Building is an entity. Its height is 102 stories. It is owned by somebody.
- In a book dealer database, entities would include customers, publishers, and books. Attributes of a customer, e.g., would include name and address. Attributes of a book would include author, title, and price. Relationships would include the customer-orders-book relation (who orders what) and the publisher-publishes-book relation (who publishes what).

The E-R is a higher-level (more abstract, less detailed) model but closely related to the relational model. Both entities and relationships can be mapped onto relations. The relational model makes no distinction between them. The E-R model, however, does not get into the business of rows and columns of tables.

**E-R Diagrams**

*E-R diagrams* are used in database design to graphically depict the entities, their attributes, and relationships. The role of E-R diagrams in database design is similar to that of pseudocode or flowcharts in algorithm and program design.

Figure 1.2 shows an E-R diagram for the book dealer database.

We will cover the E-R model in detail in Coursepack, Chapter 6 and in Silberschatz, Korth, and Sudarshan, Chapter 6.

**Other Data Models**

**QUESTION:** Are there any other data models of interest, then?

**ANSWER:** Yes. Historical interest: before relational databases became popular, there were databases based on the *hierarchical model* and the *network model*. A number of other data models are under development and might become the models for future database systems. These include the *object-oriented data model*, an attempt to extend the ideas of object-oriented programming to database systems; and the *object-relational model*, which combines object-oriented features with the relational model. The *logical model* represents data
as logical expressions and provides facilities for reasoning deductively about the
data. *Semistructured models* allow for “variant types” of records. XML (Ex-
tensible Markup Language) is an important language for semistructured data.
(Curiously, XML uses a tree structure, so in a way it’s a return to the hierar-
chical database model.)

1.6 Languages for Database Systems

**Question:** What kind of languages can be used with databases?

**Answer:** We need at least a *data-definition language (DDL)* for specifying the
schema of the database and a *data manipulation language (DML)* for storing,
retrieving, and updating the database (i.e., manipulating instances).

By far the most important language for working with databases is *SQL*
(Structured Query Language, formerly Sequel). SQL includes statements both
for specifying schemas and for manipulating instances of the database; i.e., it
incorporates both a DDL and a DML. Examples of SQL statements (both DDL
and DML) are shown in Silberschatz, Korth, and Sudarshan, p. 13.

However, applications are not typically written in just SQL. Rather, SQL
or the equivalent statements are embedded in general-purpose programming
languages, such as Java, C, or COBOL, called *host languages*. The embedded
SQL statements perform much of the data manipulation, while the host language
may be used to perform additional data manipulation and to provide user-
friendly interfaces. Another way to achieve the same goal is to provide libraries
of functions that manipulate the database (in ways similar to SQL). Examples
of such libraries are ODBC (Open Database Connectivity) and JDBC (Java
Database Connectivity). More precisely, ODBC and JDBC are *standards* which
may be implemented by particular libraries of functions.
1.7 Kinds of Database Users

**Question:** Who uses databases?

**Answer:** Nearly everyone, but in different ways. The *database administrator* in an organization is responsible for the overall design, maintenance, and supervision of the database: design of logical and physical schemas, revision of the same, database user authorization, and routine maintenance such as backup onto tape and monitoring of performance.

*Application programmers* write and maintain programs which manipulate the database. Typically these are written in a general-purpose host language with calls to SQL or the equivalent, or a special application development language which may be supplied by the DBMS vendor. Usually, these programs support the day-to-day activities of the organization, such as buying and selling.

*Naive users* are those who use the database system indirectly, through the programs written by the application programmers, e.g., in a for-profit firm, order entry personnel, shipping personnel, and customer service representatives; in a hospital, admissions personnel, nurses, and doctors.

*Managers* and *analysts* of various kinds (e.g., marketing, finance) may make *ad hoc* queries to the database to support non-routine decision making. E.g., where should we build a new factory? What kind of people will make donations to a charitable organization? Knowledge Discovery in Databases (KDD) is the discovery of general rules and patterns from large-scale collections of data, also called *data mining*.

1.8 Using Databases in a Networked Environment

**Question:** How can a database system be deployed in a networked environment?

**Answer:** In at least four ways:

1. **Client-server or two-tier architecture**: application (client) \(\leftrightarrow\) DB system (server)

2. **Three-tier architecture**: application client \(\leftrightarrow\) application server \(\leftrightarrow\) DB server

3. **Four-tier architecture**: web browser \(\leftrightarrow\) web server \(\leftrightarrow\) application server \(\leftrightarrow\) DB server

4. Of course, we could also forget that the network exists and just put everything on one server, or *mainframe*. 
1.9 Summary: What is a Database?

**Question:** Earlier you said we would refine the concept of database. When, then, is the refined concept?

**Answer:** A database is a collection of data, together with a collection of tools (the DBMS) which provide convenient and efficient means of querying and updating the database, avoidance of redundancy, maintenance of data integrity, atomic transactions, concurrent access, and fine-grained control over user authorization.
Chapter 2

The Relational Model

Reference: Silberschatz, Korth, and Sudarshan, Chapter 2.

• This chapter is “theoretical.”
• It is not going to be all like this!
• The next chapter will be very “practical.”

2.1 Review of Set Theory

Identifying Sets

Sets are collections. A set may be defined by listing its members (or elements): $A = \{1, 2, 3\}$. We write $a \in A$ to mean $a$ is a member of $A$; $a \notin A$, to mean $a$ is not a member of $A$. The empty set, written $\emptyset$ or $\{\}$, has no members.

The order in which members are listed has no significance; two sets are equal if and only if they have the same members; so $\{1, 2, 3\} = \{3, 2, 1\}$.

A set may also be defined by giving a rule: $S = \{x | x$ is a real number, $x > 5\}$.

Subsets

Definition 1 $A$ is a subset of $B$, written $A \subseteq B$, if every member of $A$ is a member of $B$.

Every set is a subset of itself. The empty set is a subset of every set.

We can use the notion of subset to define equality of sets:

Definition 2 Equality of sets. If $A$ and $B$ are sets, then $A = B$ if $A \subseteq B$ and $B \subseteq A$.

This definition is equivalent to the informal definition previously given.

Definition 3 $A$ is a proper subset of $B$, written $A \subset B$, if $A \subseteq B$ and $A \neq B$.  

15
Set Operations

The basic operations combining sets are union, intersection, difference, and Cartesian product.

**Definition 4** If $A$ and $B$ are sets, then:

1. The **union** of $A$ and $B$ is $A \cup B = \{x | x \in A \text{ or } x \in B\}$.
2. Their **intersection** is $A \cap B = \{x | x \in A \text{ and } x \in B\}$.
3. The difference $A - B = \{x | x \in A \text{ and } x \notin B\}$.

**Example 1** Let $A = \{g, u, m, e\}, B = \{d, o, g, e\}$. Then:

- $A \cup B = \{g, u, m, e, d, o\}$
- $A \cap B = \{g, e\}$
- $A - B = \{u, m\}$
- $B - A = \{d, o\}$

The Cartesian product $A \times B$ is formed by pairing every element of $A$ with every element of $B$. The result is a set of ordered pairs. Formally:

**Definition 5** [Cartesian product] If $A$ and $B$ are sets, then

$$A \times B = \{(a, b) | a \in A \text{ and } b \in B\}$$

In general, the Cartesian product of $n$ sets is a set of $n$-tuples

**Example 2** Let $A = \{a, b\}, M = \{m, n, o\}$. Then

$$A \times M = \{(a, m), (a, n), (a, o), (b, m), (b, n), (b, o)\}$$

**Exercises**

Given: $Q = \{1, 7, 9\}, R = \{7, 11, 15\}, S = \{1, 11\}$.

1. List all the subsets of $S$. (Hint: there are 4.)
2. List all the subsets of $R$. (Hint: there are 8.)
3. Write the result of each operation by listing the members of the set:

   (a) $Q \cap R$
   (b) $R \cup S$
   (c) $Q - S$
   (d) $(Q \cup R) - S$
   (e) $(Q - R) \cup (R - Q)$
   (f) $Q \times S$
2.2. INTRODUCTION

(g) \( S \times Q \)

(h) \( \{(q, r) \mid (q, r) \in Q \times R \text{ and } q + r < 20 \} \)

4. How many elements are in \( A \times B \) if \( A \) and \( B \) each contain 10 elements?

5. How many subsets are there of a set of 10 elements?

2.2 Introduction

The relational model—the brain-child of E. F. Codd, ca. 1970—is the “primary data model” in use today, as the textbook points out, because of its simplicity (compared to the alternatives). The relational model can be formulated using any of three very formal, user-unfriendly languages:

1. the relational algebra (covered in detail in this chapter)
2. the tuple relational calculus
3. the domain relational calculus

This chapter presents important aspects of the theory of relational databases. Why not skip the theory and jump right into SQL? We could do that, just as one could study a numerical programming language, such as C or FORTRAN, without first studying the algebra of numbers. But we wouldn’t have a firm foundation, and our ability to use the language would be limited.

Relations

A relational database (RDB) is a collection of relations. Informally, relations are tables of data. Formally, relations are defined using concepts from set theory.

Relations in Set Theory

The Cartesian product of two sets \( A, B \) is \( A \times B = \{(a, b) \mid a \in A \text{ and } b \in B \} \), the set of all ordered pairs in which the first element is taken from \( A \) and the second from \( B \). A binary relation on \( A, B \) is a subset of \( A \times B \). That is, it is a set of ordered pairs \((a, b)\) subject to the restriction that \( a \in A \) and \( b \in B \).

Example 3 Let \( A = \{a, b, c\}, B = \{1, 2\}, \) and \( R = \{(a, 2), (b, 1), (b, 2)\} \). Then \( A \times B = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\} \). Since \( R \subseteq A \times B \), \( R \) is a binary relation on \( A, B \).

We may generalize the concepts of Cartesian product and relation to \( n \) sets. The Cartesian product of the sets \( S_1, S_2, \ldots, S_n \) is \( S_1 \times S_2 \times \ldots \times S_n = \{(e_1, e_2, \ldots, e_n) \mid e_1 \in S_1, e_2 \in S_2, \ldots, e_n \in S_n \} \). A relation on \( S_1, S_2, \ldots, S_n \) is a subset of \( S_1 \times S_2 \times \ldots \times S_n \); that is, it is a set of \( n \)-tuples \((e_1, e_2, \ldots, e_n)\) where \( e_i \in S_i \) for each \( i \). A relation on \( n \) sets is called an \( n \)-ary relation.
Examples of Relations

**Example 4** Some states and their capital cities: \{ (Indiana, Indianapolis), (Illinois, Springfield), (Ohio, Columbus) \}. (A relation on the set of states and the set of cities)

**Example 5** Some pairs of cities and the (approximate) distance between them: \{ (Richmond, Indianapolis, 70), (Richmond, Dayton, 45), (Dayton, Springfield, 20), (Indianapolis, Bloomington, 50) \}. (A relation between the set of cities, the set of cities, and the non-negative real numbers)

**Example 6** Some companies and the products they make: \{ (IBM, computers), (IBM, software), (HP, computers), (Microsoft, software), (Kraft, cheese), (Kellogg’s, cereal), (Post, cereal) \}. (A relation between what?)

In Examples 4 and 5, the relations are *functions*, since each state has only one capital, and each pair of cities has only one distance. In Example 6, the relation is not a function, since some companies make more than one product.

### 2.3 Relations in Database Theory

Our database relations are like those in set theory, except that we will identify the components of \( n \)-tuples by name instead of number. A set theorist, looking at the two parts of (Kraft, cheese), typically refers to them as \( x_1 \) and \( x_2 \); we prefer to give them meaningful names, such as company and product. We typically write the relation in the form of a table, using the names as column headings:

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>computers</td>
</tr>
<tr>
<td>IBM</td>
<td>software</td>
</tr>
<tr>
<td>HP</td>
<td>computers</td>
</tr>
<tr>
<td>Microsoft</td>
<td>software</td>
</tr>
<tr>
<td>Kraft</td>
<td>cheese</td>
</tr>
<tr>
<td>Kellogg’s</td>
<td>cereal</td>
</tr>
<tr>
<td>Post</td>
<td>cereal</td>
</tr>
</tbody>
</table>

**Attributes, Domains, Tuples**

Each column of the table is called an *attribute*. Each attribute has a *domain* or set of allowed values (i.e., the domains of the relation are the sets that it is “on”).

The rows of the table are called *tuples*, which should make the set theorist happy :-).
2.3. RELATIONS IN DATABASE THEORY

Atomic Domains

Most work on relational databases assumes that the domains are atomic, i.e., the allowed values of any attribute cannot be divided into meaningful parts. For example, numbers and strings are atomic data. On the other hand, a list of numbers, a set of strings, or a relation consisting of number-string tuples, is non-atomic and would not be permitted.

No Tables Within Tables

The prohibition against relations as domain values means that we cannot have tables within tables, such as this:

<table>
<thead>
<tr>
<th>teacher</th>
<th>classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td></td>
</tr>
<tr>
<td></td>
<td>course-number</td>
</tr>
<tr>
<td></td>
<td>1300</td>
</tr>
<tr>
<td></td>
<td>1380</td>
</tr>
<tr>
<td>Brown</td>
<td></td>
</tr>
<tr>
<td></td>
<td>course-number</td>
</tr>
<tr>
<td></td>
<td>C201</td>
</tr>
<tr>
<td></td>
<td>C202</td>
</tr>
<tr>
<td></td>
<td>C335</td>
</tr>
</tbody>
</table>

Schemata

The schema (plural schemata) of a relation is a list of its attributes and (possibly) their domains.

Example 7

Class-schema = (course-number, section-number, enrolled).

We can express the fact that class is a relation with this schema by writing

class(course-number, section-number, enrolled)

or

class(Class-schema)

A relation schema corresponds to a type in an ordinary programming language, since it specifies the kind of values that the relation can hold.

A relation instance is the “value,” or contents, of a relation at a particular time (like the value of a variable). Strictly speaking, a relation instance is simply a relation, i.e., a set of n-tuples; and the mutable thing, sometimes called a relation, that can store a succession of relations (relation instances) as
values over time, should be called a relation variable, as C. J. Date has pointed out. But this terminology is not widely used.

The logical design, or schema, of a relational database is the collection of its relation schemata.

**Keys**

We need to be able to distinguish one tuple from another, and, in a database, we can only do this by using their attributes. Consequently, no two tuples in a relation may have the same attribute values, for they could not be distinguished. A key is a set of attributes which uniquely identifies each individual tuple.

More precisely: let \( R \) be a relation with attributes \( A_1, A_2, \ldots, A_n \). Let \( K \subseteq \{A_1, A_2, \ldots, A_n\} \). If the values of the attributes in \( K \) are sufficient to distinguish each tuple in \( R \), then \( K \) is a superkey for \( R \). A superkey may contain more attributes than are really needed to distinguish between tuples. If we eliminate attributes that are not necessary, we are left with a candidate key or minimal superkey. I.e., if \( K \) is a superkey for \( R \), and no proper subset of \( K \) is a superkey for \( R \), then \( K \) is a candidate key for \( R \). The database designer selects one of the candidate keys as the primary key, i.e., the chief means of identifying individual tuples in \( R \).\(^1\)

**Example 8** Suppose that the Student relation has attributes name, address, major, sid (student ID), and credits (credit hours). Suppose further that no two students can have the same name and the same address, and no two students can have the same SID. Which of the following are superkeys? Which are candidate keys? Which would be a good primary key?

- \{name, address, major, sid, credits\}
- \{name, major, credits \}
- \{name, address, sid\}
- \{name, sid, credits\}
- \{name, address\}
- \{sid\}

Why would \{name, address\} not be a good choice of primary key? (What happens if the student moves or changes her name?)

**Schema Notation for Keys**

The primary key should be underlined in a relation schema. Example:

\[
\text{Class-schema} = (\text{course-number}, \text{section-number}, \text{enrolled})
\]

\(^1\)Candidate keys, then, are so called because they are the ones considered as possible candidates to become the primary key.
2.4 DATA FOR EXAMPLES

Foreign Keys

A relation may also have a foreign key, which is an attribute (or set of attributes) that references the primary key of another relation.

Example 9 In the Salesperson relation, below, the attribute region-number refers to the primary key of the Region relation. Therefore, it is a foreign key.

<table>
<thead>
<tr>
<th>SSN</th>
<th>name</th>
<th>region-number</th>
</tr>
</thead>
<tbody>
<tr>
<td>300-00-0000</td>
<td>Schmidt</td>
<td>9</td>
</tr>
<tr>
<td>300-00-0001</td>
<td>Riley</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>region-number</th>
<th>description</th>
<th>headquarters</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Southwest</td>
<td>Phoenix</td>
</tr>
<tr>
<td>4</td>
<td>Midwest</td>
<td>Chicago</td>
</tr>
<tr>
<td>9</td>
<td>Northeast</td>
<td>Boston</td>
</tr>
</tbody>
</table>

Salesperson is the referencing relation. Region is the referenced relation. The foreign key links the two relations together, so that we can find, for example, what is the headquarters of Schmidt’s region.

Schema Diagrams

Schema diagrams show the linkage between relations made by foreign keys. See Silberschatz, Korth, and Sudarshan, p. 44. In the diagram, each relation is shown as a box; within the box, primary key attributes are listed first, and separated by a horizontal line from the other attributes. Lines with arrows show foreign key dependencies, i.e., the connections made by foreign keys. Each such line points from a foreign key in the referencing relation, to the corresponding primary key in the referenced relation.

2.4 Data for Examples

We will use the following tables in a series of examples that illustrate the relational operations:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>
CHAPTER 2. THE RELATIONAL MODEL

B

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
</tbody>
</table>

product

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>P5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>P6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

distributor

<table>
<thead>
<tr>
<th>did</th>
<th>dname</th>
<th>dcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Peter’s Powerboats</td>
<td>Chicago</td>
</tr>
<tr>
<td>D2</td>
<td>Sarah’s Sails</td>
<td>St. Louis</td>
</tr>
<tr>
<td>D3</td>
<td>Marty’s Marina</td>
<td>Mobile</td>
</tr>
</tbody>
</table>

product-distributor

<table>
<thead>
<tr>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>D3</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>D3</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>D1</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>D3</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>D1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>D2</td>
<td>12</td>
</tr>
<tr>
<td>P6</td>
<td>D2</td>
<td>8</td>
</tr>
</tbody>
</table>

2.5 Union, Intersection, and Difference

Union, intersection, and difference are the same as the operations on sets, but, in their relational forms, they require both operands to have the same schema. The result must contain tuples of only one type. It would make no sense, for example, to form the union of a table of automobiles and a table of owners, where the automobile table has attributes (VIN, make, model, year) and the owner table has attributes (name, license-number).
2.6 Select and Project

The select operator ($\sigma$) forms a table by selecting tuples (rows) from a given table that meet a specified criterion.

The project operator ($\pi$) forms a table by selecting specified attributes (columns) from a given table.

Note: Silberschatz, Korth, and Sudarshan use $\Pi$ (capital pi) for the projection operator, but the lower-case $\pi$ is more commonly used.

Example 13 $\sigma_{\text{name} = \text{"Motorboat"}}(\text{product}) =$

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>P4</td>
<td>motorboat</td>
<td>30</td>
</tr>
</tbody>
</table>

Example 14 $\sigma_{\text{length} > 25}(\text{product}) =$

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>P6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

Example 15 $\pi_{\text{pid, name}}(\text{product}) =$

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>rowboat</td>
</tr>
<tr>
<td>P2</td>
<td>canoe</td>
</tr>
<tr>
<td>P3</td>
<td>motorboat</td>
</tr>
<tr>
<td>P4</td>
<td>motorboat</td>
</tr>
<tr>
<td>P5</td>
<td>sailboat</td>
</tr>
<tr>
<td>P6</td>
<td>sailboat</td>
</tr>
</tbody>
</table>
Example 16 \[ \pi_{\text{pname}}(\text{product}) = \]
\[
\begin{array}{l}
\text{name} \\
\text{rowboat} \\
\text{canoe} \\
\text{motorboat} \\
\text{sailboat} \\
\end{array}
\]

Note the elimination of duplicates.

### 2.7 Combinations of Operators

Often, queries require a combination of relational operators.

**Example 17** “Find the lengths of the sailboats.”

\[ \pi_{\text{length}}(\sigma_{\text{pname} = \text{"Sailboat"}}(\text{product})) = \]
\[
\begin{array}{c}
\text{length} \\
20 \\
30 \\
\end{array}
\]

**Example 18** “Find the pid’s and names of products under twenty feet in length.”

\[ \pi_{\text{pid}, \text{pname}}(\sigma_{\text{length} < 20}(\text{product})) \]

**Example 19** “Find the lengths of sailboats and motorboats.”

\[ \pi_{\text{length}}(\sigma_{\text{pname} = \text{"Motorboat"}}(\text{product}) \cup \sigma_{\text{pname} = \text{"Sailboat"}}(\text{product})) \]

This could also be done using the logical operator “or” (\( \lor \)) in the select criterion.

**Example 20** “Find product IDs of products under 30 feet in length that are not sailboats.”

\[ \pi_{\text{pid}}(\sigma_{\text{length} < 30}(\text{product}) - \sigma_{\text{pname} = \text{"Sailboat"}}(\text{product})) \]

This could also be done using the logical operators “and” (\( \land \)) and “not” (\( \neg \)) in the select criterion.

### 2.8 Cartesian Product and Natural Join

**Cartesian Product**

The Cartesian product is similar to the one for sets, but if \( R \) is an \( m \)-ary relation, and \( S \) is an \( n \)-ary relation, then \( R \times S \) is an \( (m + n) \)-ary relation, i.e., it consists of \( (m + n) \)-tuples representing every pair of \( r \in R \) and \( s \in S \). The number of rows (tuples) in the product is the product of the number of rows in \( R \) and the number of rows in \( S \).

Clarification of the difference:
2.8. **Cartesian Product and Natural Join**

\[ R = \{(1, 2), \ldots\} \]

\[ S = \{(3, 4), \ldots\} \]

Set theory:

\[ R \times S = \{(1, 2), (3, 4), \ldots\} \]

Relational algebra:

\[ R \times S = \{(1, 2, 3, 4), \ldots\} \]

**Example 21** Let \( P \) be the first two rows of the product relation, and let \( Q \) be the first three rows of the product-distributor relation.

\[ P = \begin{array}{lll}
\text{pid} & \text{pname} & \text{length} \\
P1 & \text{rowboat} & 10 \\
P2 & \text{canoe} & 10 \\
\end{array} \]

\[ Q = \begin{array}{lll}
\text{pid} & \text{did} & \text{quantity} \\
P1 & D3 & 10 \\
P2 & D3 & 10 \\
P3 & D1 & 10 \\
\end{array} \]

\[ R = P \times Q = \begin{array}{llllll}
P1 & \text{rowboat} & 10 & P1 & D3 & 10 \\
P1 & \text{rowboat} & 10 & P2 & D3 & 10 \\
P1 & \text{rowboat} & 10 & P3 & D1 & 10 \\
P2 & \text{canoe} & 10 & P1 & D3 & 10 \\
P2 & \text{canoe} & 10 & P2 & D3 & 10 \\
P2 & \text{canoe} & 10 & P3 & D1 & 10 \\
\end{array} \]

\( P \) and \( Q \) have three attributes each, so \( R \) has \( 3 + 3 = 6 \) attributes.
Note that we have *renamed* the two “pid” columns to avoid ambiguity.

**Natural Join**

The rows where \( P.\text{pid} = Q.\text{pid} \) are of special interest, for they are the ones in which the tuple from \( P \) is related to the tuple from \( Q \) by having the same pid.

If we select these rows and “merge” the two “pid” columns, we have the *natural join* of \( P \) and \( Q \):

\[ P \bowtie Q = \begin{array}{llll}
\text{pid} & \text{pname} & \text{length} & \text{did} & \text{quantity} \\
P1 & \text{rowboat} & 10 & D3 & 10 \\
P2 & \text{canoe} & 10 & D3 & 10 \\
\end{array} \]
Defining Natural Join

For relations $A$ and $B$ having one shared attribute $S$, the natural join is defined as

$$A \bowtie B = \sigma_{A.S = B.S}(A \times B).$$

The result has only a single column for $S$. I.e., the column data for $A.S$ and $B.S$ are the same; only the names are different; so we remove one and rename the other to just plain $S$.

Two relations might have more than one attribute in common. In that case, the natural join is formed from the Cartesian product by selecting the rows in which all of the shared attributes are equal. E.g., if the two shared attributes were $A$ and $B$, the natural join would contain the rows in which $P.A = Q.A$ and $P.B = Q.B$.

In general, if $A$ and $B$ share $n$ attributes $S_1, \ldots, S_n$, then

$$A \bowtie B = \sigma_{A.S_1 = B.S_1 \wedge \ldots \wedge A.S_n = B.S_n}(A \times B)$$

with the same elimination and renaming of duplicate columns.

Characteristics of Natural Join

1. The natural join is associative: $(P \bowtie Q) \bowtie R \equiv P \bowtie (Q \bowtie R)$.

2. It is more efficient to compute the natural join directly, without computing the Cartesian product. Just select and join together the matching rows of the two tables. (Query optimization)

Examples of Natural Join

Example 22 “Find the names of distributors who sell rowboats.”

$$\pi_{\text{dname}}(\sigma_{\text{pname} = \text{"Rowboat"}}(\text{product} \bowtie \text{product-distributor} \bowtie \text{distributor}))$$

Example 23 “Find the names of products and distributors where the quantity is more than ten.”

$$\pi_{\text{pname, dname}}(\sigma_{\text{quantity} > 10}(\text{product-distributor} \bowtie \text{product} \bowtie \text{distributor}))$$

Example 24 “Find the names of products sold in St. Louis.”

$$\pi_{\text{pname}}(\sigma_{\text{city} = \text{"St. Louis"}}(\text{distributor} \bowtie \text{product-distributor} \bowtie \text{product}))$$

2.9 Assignment

Sometimes it is convenient to break up long expressions by writing assignments to local variables.
Example 25 We could rewrite the last example, “Find the names of products sold in St. Louis,” as

\[
R_1 \leftarrow \sigma_{\text{city} = \text{"St. Louis"}}(\text{distributor})
\]

\[
R_2 \leftarrow R_1 \bowtie \text{product-distributor} \bowtie \text{product}
\]

Answer \leftarrow \pi_{\text{pname}}(R_2)

Note that \( R_1 \) and \( R_2 \) are temporary relations, not relation variables that exist permanently in the database. We are merely querying, at this stage, not updating.

2.10 Rename

It is occasionally useful to name or rename a relation and/or its attributes. This trivial operation is performed by the rename operator (\( \rho \)).

In the simplest form, the name (or new name) of a relation is written as a subscript: \( \rho_C(A) \) is the relation \( A \), renamed \( C \); \( \rho_D(A \cup B) \) is \( A \cup B \) under the name of \( D \).

In addition, the attributes can be renamed by writing their new names in a list following the new relation name.

Example 26 \( \rho_{E(\text{called}, \text{years})}(A) = \)

<table>
<thead>
<tr>
<th>called</th>
<th>years</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>

2.11 Division (Optional)

Probably the most confusing of the relational operators.

The division operator (\( \div \)) is the inverse of the Cartesian product, in the same way that integer division is the inverse of integer multiplication. Surprisingly, \( \div \) is useful primarily as a way to formulate queries involving the word “all” or “each” or “every”, as in “Find the publishers who publish every composition of J. S. Bach.”

Inverting Multiplication 1: Analogy with Numbers

Let’s examine the pair of integer operations first. If \( a, b, c \) are integers, and “/” represents integer division (i.e., discarding any remainder), then \( c = ab \neq 0 \) implies both \( c/a = b \) and \( c/b = a \). For example, \( 28 = 7 \cdot 4 \), and consequently both \( 28/7 = 4 \) and \( 28/4 = 7 \). However, we cannot turn the implication around in general: neither \( c/a = b \) nor \( c/b = a \) implies that \( c = ab \).

For example, \( 31/7 = 4 \) (remember, we are discarding any remainder), but \( 7 \cdot 4 \neq 31 \). If the remainder is zero, however, the implication goes both ways.
CHAPTER 2. THE RELATIONAL MODEL

Inverting Multiplication 2: Relations

Now, back to relations. The inverse of $\times$ is $\div$, in the sense that, if $A, B, C$ are relations, then $C = A \times B \neq \emptyset$ implies both $C \div A = B$ and $C \div B = A$. However, the reverse implications are not necessarily true.

Example 27 Let $P, Q$, and $R$ be the relations given in Example 21. Since $R = P \times Q \neq \emptyset$, it follows that $R \div P = Q$ and $R \div Q = P$.

How to Divide Relations

How do we compute the division? Of course, it is the reverse of computing the Cartesian product. To compute $C = A \times B$, we need to determine both the columns (attributes) and the rows (tuples) of the result. The attributes of $C$ will be all the attributes of $A$ and all the attributes of $B$. To find the tuples of $C$, we take each tuple of $A$ and combine it with each tuple of $B$ to form a tuple in $C$. In computing $A = C \div B$, we do the reverse. The attributes of $A$ are the attributes of $C$ excluding the attributes of $B$. The tuples in $A$ are those that are combined with each tuple of $B$ to form tuples in $C$. That explains why division is good for formulating queries involving “each” and similar words.

Division: Computational Example

Example 28 Find the distributors (did’s) who distribute every model of motorboat.

Method of solution: First find the pid’s of products that are motorboats:

$$M \leftarrow \pi_{\text{pid}}(\sigma_{\text{name} = \text{"Motorboat"}}(\text{product}))$$

Then get the (pid, did) pairs where the quantity is positive:

$$S \leftarrow \pi_{\text{pid, did}}(\sigma_{\text{quantity} > 0}(\text{product-distributor}))$$

Finally, divide $S \div M$ to get the answer.

We now carry out the solution computationally:

$$M = \begin{array}{c}
\text{pid} \\
P3 \\
P4
\end{array}$$

$$S = \begin{array}{cc}
\text{pid} & \text{did} \\
P1 & D3 \\
P2 & D3 \\
P3 & D1 \\
P3 & D3 \\
P4 & D1 \\
P5 & D2 \\
P6 & D2
\end{array}$$
Now let’s find $T = S \div M$. The attributes of $T$ will be $\{\text{pid, did}\} - \{\text{pid}\} = \{\text{did}\}$. $T$ contains a tuple for D1, since D1 occurs in $S$ with both P3 and P4, i.e., with all the rows of $M$. $T$ contains no tuples for D2 or D3, since neither of these occurs in $S$ with both P3 and P4.

Therefore,

$$T = \begin{array}{c}
\text{did} \\
\text{D1}
\end{array}$$

Using Division to Formulate Queries

**Example 29** “Find the publishers who publish every composition by J. S. Bach.” Assume the relation schema composition(composer, title, publisher).

$$PT \leftarrow \pi_{\text{title, publisher}}(\sigma_{\text{composer} = \text{“J. S. Bach”}}(\text{composition}))$$

$$T \leftarrow \pi_{\text{title}}(PT)$$

$$\text{Answer} \leftarrow PT \div T$$

**Example 30** “Find computer science majors who have taken every required computer science course.” Assume the relation schemata major(student, subject); required-course(subject, course); taken-course(student, course).

$$C \leftarrow \pi_{\text{course}}(\sigma_{\text{subject} = \text{“computer science”}}(\text{required-course}))$$

$$SC \leftarrow \text{taken-course} \bowtie \pi_{\text{student}}(\sigma_{\text{subject} = \text{“computer science”}}(\text{major}))$$

$$\text{Answer} \leftarrow SC \div C$$

**Example 31** “Find the senators who voted for each of bills A, B, and C.” Assume the relation schema voted(senator, bill, vote), where vote = “yea” or “nay.”

$$SB \leftarrow \pi_{\text{senator, bill}}(\sigma_{\text{vote} = \text{“yea”}}(\text{voted}))$$

$$\text{Answer} \leftarrow SB \div \{(A), (B), (C)\}$$

### 2.12 Generalized Projection

We extend the idea of projection to allow creating new columns based on functions (e.g., arithmetic functions) of the old columns.

**Example 32** If length is given in feet, convert it to yards:

$$\pi_{\text{pid, length}/3} \text{ as yards}(\text{product})$$
2.13 Aggregation

Aggregate operations are those that combine all the values of a column into a single value, such as the sum, average, or minimum. The symbol $\mathcal{G}$ is used to represent an aggregate operation. Note that for the purpose of computing the aggregate value, duplicate values in the column are not eliminated.

Examples of Aggregation

Example 33

\[ \mathcal{G}_{\text{sum}(\text{length})}(\text{product}) = \frac{\text{sum}(\text{length})}{120} \]

Example 34

\[ \mathcal{G}_{\text{min}(\text{quantity})}(\text{product-distributor}) = \frac{\text{min}(\text{quantity})}{5} \]

Grouping the Results of Aggregation

Sometimes it is desirable to combine, or group, aggregate values according to the value of some other column(s). For example, we might want the average of “quantity”, grouped by distributor:

\[ \text{did} \mathcal{G}_{\text{average}(\text{quantity})}(\text{product-distributor}) = \]
\begin{tabular}{|c|c|}
\hline
\text{did} & \text{average(quantity)} \\
\hline
D1 & 7.5 \\
D2 & 10.0 \\
D3 & 13.3333333333 \\
\hline
\end{tabular}

or grouped by product:

\[ \text{pid} \mathcal{G}_{\text{average}(\text{quantity})}(\text{product-distributor}) = \]
\begin{tabular}{|c|c|}
\hline
\text{pid} & \text{average(quantity)} \\
\hline
P1 & 10 \\
P2 & 10 \\
P3 & 15 \\
P4 & 5 \\
P5 & 12 \\
P6 & 8 \\
\hline
\end{tabular}

Multiple Groupings

An aggregate function can be grouped by more than one attribute. For example, state, party $\mathcal{G}_{\text{sum}(\text{membership})}(\text{registration})$ would have column headings state, party, and sum(membership) and would have a row for each possible value of (state, party).
2.14 Outer Joins

An outer join is an extension of the natural join operation which adds tuples from one or the other of the two relations that are unmatched in the other. These tuples must have null for some of their attributes, since the attribute values are unknown. The left outer join adds tuples from the left relation. The right outer join adds tuples from the right relation; and the full outer join adds tuples from both relations. For examples, see Silberschatz, Korth, and Sudarshan, Figures 2.30–2.33 on pages 64–66.

2.15 Null Values at Work

Nulls signify a missing value (unknown or non-existent). The use of null “values” complicates the relational algebra.

1. The use of null in arithmetic expressions results in a null value for the expression. That is, for if either $a$ is null or $b$ is null, then the values of $a + b$, $a - b$, $a \cdot b$, and $a/b$ are all null.

2. The use of null in numerical or string comparisons results in an unknown value for the comparison. That is, if either $a$ is null or $b$ is null, then the values of $a < b$, $a \leq b$, $a = b$, $a \neq b$, $a \geq b$, and $a > b$ are all unknown, a new truth value.

3. How should the relational operators ($\pi$, etc.) handle nulls?

Three-Valued Logic

To accommodate unknown as a truth value, we must use a three-valued logic.

The following truth tables extend the logical operations AND, OR, NOT, representing the truth value unknown as $U$:

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F   (why?)</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T   (why?)</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>F</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Note that “not unknown” does not equal “known”!

Does three-valued logic confuse knowledge of the subject with knowledge of one’s knowledge of the subject (meta-knowledge)?

Does three-valued logic make better sense with the Haskell \texttt{Maybe} type?

\texttt{Maybe Bool = Nothing | Just Bool}

The three values are:

- Nothing
- Just False
- Just True

**Null in Relational Operations**

How the relational operators ($\sigma$, $\pi$, $\times$, etc.) should handle null values is partly an arbitrary decision. The following rules accord with the treatment of null values in SQL:

1. \( \sigma P(E) \) contains tuple \( t \in E \) only if \( P(t) = \text{true} \) (i.e., not false or unknown).

2. \( \pi \) treats all null values as equal in eliminating duplicates. E.g., if a result contains (before elimination of duplicates) \((a, b, \text{null})\) and \((a, b, \text{null})\), the two tuples are considered duplicates and only one will remain. Discuss: Why might this be a poor choice?

3. Similarly, the operations \( \cup, \cap, - \) all treat null values as equal. In the case of union, this means that, e.g., \((a, b, \text{null})\) and \((a, b, \text{null})\) are treated as one tuple and only one will remain in the result. In the case of intersection and difference, if \((a, b, \text{null}) \in A\) and \((a, b, \text{null}) \in B\), then \(A \cap B\) contains \((a, b, \text{null})\), and \(A - B\) does not. Again, why might these choices be unfortunate?

4. Since \( \times \) can be defined in terms of \( \times \) and \( \sigma \), its behavior depends on the choice we made for \( \sigma \). \( \times \) treats tuples with null in the common attribute as not matching, so they are left out of the result.

5. How nulls are treated by generalized projection, aggregation, and outer joins is presented in Silberschatz, Korth, and Sudarshan, pp. 67–68.

**Null as Nothing**

The Haskell value \texttt{Nothing} seems to correspond to the SQL value (or non-value) \texttt{null}. Yet while \texttt{null} is a possible string value in SQL, \texttt{Nothing} is not a possible string value in Haskell. It is a possible value of type \texttt{Maybe String}.

The Haskell type system forces a distinction between something of type \( a \), and something which is normally of type \( a \), but may be \texttt{null} (or \texttt{Nothing} in Haskell terms). The latter does not have type \( a \), but type \texttt{Maybe a}.
For example, a name would normally be a string. If it is always a string, its Haskell type would be String, but if it is sometimes null, then we have to express its Haskell type as 

<table>
<thead>
<tr>
<th>SQL string</th>
<th>Haskell String</th>
<th>Haskell Maybe String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Joel&quot;</td>
<td>&quot;Joel&quot;</td>
<td>Just &quot;Joel&quot;</td>
</tr>
<tr>
<td>&quot;Hannah&quot;</td>
<td>&quot;Hannah&quot;</td>
<td>Just &quot;Hannah&quot;</td>
</tr>
<tr>
<td>null</td>
<td>Nothing</td>
<td></td>
</tr>
</tbody>
</table>

2.16 Mutating the Database

Since a database models a part of the world, and that part of the world may be changing, we need to be able to mutate a database, by inserting, deleting, and updating tuples in its relations. We use the assignment operator (<-) to do this. Let $R$ be an existing relation (not local variable). Let $E$ and $E'$ be expressions in relational algebra, denoting sets of tuples. Then,

1. We can insert the tuples of $E$ into $R$:

   $$ R \leftarrow R \cup E $$

2. We can delete the tuples of $E$ from $R$:

   $$ R \leftarrow R - E $$

3. We can replace the tuples of $E$ in $R$ by tuples in $E'$:

   $$ R \leftarrow (R - E) \cup E' $$

   If the tuples of $E'$ are derived from the tuples of $E$, then we are in some sense updating the tuples in $E$.

Mutation Examples

Example 35 Delete Carol from B:

$$ B \leftarrow B - \sigma_{\text{name} = \text{"Carol"}}(B) $$

Example 36 Delete the tuple (P3, D3, 20) from product-distributor:

$$ \text{product-distributor} \leftarrow \text{product-distributor} - \{(P3, D3, 20)\} $$

Example 37 Delete all tuples having pid = P3 from product-distributor:

$$ \text{product-distributor} \leftarrow \text{product-distributor} - \sigma_{\text{pid}=P3}(\text{product-distributor}) $$

Example 38 Insert the tuple (P7, submarine, 50) into product:

$$ \text{product} \leftarrow \text{product} \cup \{(P7, \text{submarine}, 50)\} $$
Example 39 Extend the length of each boat in the product relation by adding 10 feet:

\[
\text{product} \leftarrow \pi_{\text{pid, pname, length + 10 as length}}(\text{product})
\]

Example 40 Reduce the quantity of P3 at D1 by 1.

\[
\begin{align*}
M & \leftarrow \sigma_{\text{pid = P3} \land \text{did = D1}}(\text{product-distributor}) \\
M' & \leftarrow \pi_{\text{pid, did, quantity - 1 as quantity}}(M) \\
\text{product-distributor} & \leftarrow (\text{product-distributor} - M) \cup M'
\end{align*}
\]

Here, the local variable \( M \) contains the tuples to be updated, and the local variable \( M' \) contains the updated tuples, i.e., the new values.

### 2.17 Closing Remarks on the Relational Algebra

The relational algebra is a powerful language for querying and updating relational databases. Although it is imposingly formal and unfriendly, it is the basis of the friendlier language SQL, which is the *lingua franca* of modern database systems.

It is worth noting that the relational algebra is a *procedural language*, in the sense that C and Java are procedural languages and formal logic is not. That is, a relational algebra expression implies a certain *order of operations*, just like in the algebra of numbers.

For example, in \( \pi_{a, b}(\sigma_P(R)) \), the select operation is performed first, and then the projection. It is not the same as \( \sigma_P(\pi_{a, b}(R)) \), and may have a different value.

The relational algebra also teaches us that in some cases the order of operations makes no difference to the final result, although it may make a great deal of difference in computational time. For example, the natural join is associative, so that \( (A \bowtie B) \bowtie C \equiv A \bowtie (B \bowtie C) \). Knowing this (and other laws of the same sort) enables database system programmers to write *query optimizers*, i.e., a query is transformed from an inefficient form to an efficient form that gives the same result, saving computational resources.
Chapter 3

SQL

References:

- Silberschatz, Korth, and Sudarshan, Chapter 3.
- The PostgreSQL Global Development Group, PostgreSQL 8.1.0 Documentation, 2005:
  - I. Tutorial (entire).
  - II. The SQL Language.
  - VI. Reference, Part I, “SQL Commands.”

The I340 class home page has links to the PostgreSQL documentation. Reading the entire Tutorial is strongly recommended. The other documents can be referred to as need or interest arises.

3.1 Getting Started

Shell command to start the PostgreSQL command monitor:

$ psql [-h servername] [-U username] [dbname]

Remarks:

- []’s indicate optional arguments.
  (If you use the optional arguments, do not type the brackets.)

- -h servername specifies the host PostgreSQL server host.
  The computer on which the PostgreSQL server is running.
  Default = localhost.

- -U username specifies the PostgreSQL user name.
  Default = Linux user name.
• `dbname` specifies the database name.  
  Default = user name.

• To quit, enter `\q`

• To list all available databases: `psql -l` or `psql --list`

### Getting Started: Examples

**Example 41** To connect as PostgreSQL user “herbert” to database “testdb” on PostgreSQL server at mis.iue.edu:

```
$ psql -h mis.iue.edu -U herbert testdb
```

**Example 42** A very brief PostgreSQL session: see see Listing 3.1.

Listing 3.1: Starting and stopping psql.

```bash
1 bash-2.05b$ psql test
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
      \h for help with SQL commands
      \? for help on internal slash commands
      \g or terminate with semicolon to execute query
      \q to quit

test=# \q
11 bash-2.05b$
```

### 3.2 Data for Examples

Most or all of the examples will use the same data as the examples in Coursepack Chapter 2, except that the relations A and B are renamed “alpha” and “beta”.

For reference, the data are shown in Listing 3.2.

Listing 3.2: Data for SQL examples, as printed by psql.

```
test=# -- Display all the tables in the database
test=# \dt
List of relations

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>alpha</td>
<td>table</td>
<td>postgres</td>
</tr>
<tr>
<td>beta</td>
<td>table</td>
<td>postgres</td>
</tr>
<tr>
<td>distributor</td>
<td>table</td>
<td>postgres</td>
</tr>
<tr>
<td>product</td>
<td>table</td>
<td>postgres</td>
</tr>
<tr>
<td>product.distributor</td>
<td>table</td>
<td>postgres</td>
</tr>
</tbody>
</table>
(5 rows)
```
3.2. DATA FOR EXAMPLES

test=# select * from alpha;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Cath</td>
<td>23</td>
</tr>
<tr>
<td>Geo</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>

(4 rows)

test=# select * from beta;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Cath</td>
<td>23</td>
</tr>
</tbody>
</table>

(4 rows)

test=# select * from distributor;

<table>
<thead>
<tr>
<th>did</th>
<th>dname</th>
<th>dcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Peter’s Powerboats</td>
<td>Chicago</td>
</tr>
<tr>
<td>d2</td>
<td>Sarah’s Sails</td>
<td>St. Louis</td>
</tr>
<tr>
<td>d3</td>
<td>Marty’s Marina</td>
<td>Mobile</td>
</tr>
</tbody>
</table>

(3 rows)

test=# select * from product;

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(6 rows)

test=# select * from product_distributor;

<table>
<thead>
<tr>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>d1</td>
<td>5</td>
</tr>
</tbody>
</table>

(5 rows)
3.3 SQL Data Definition Language

A Data Definition Language (DDL) provides facilities for defining or declaring relations. Declaring relations in SQL is a lot like declaring a structure data type or class in a programming language. We will first briefly survey some of the built-in data types of SQL, and then statements that are used to declare tables.

SQL Data Types

Before creating a table, we need to know how to specify the domain type for each attribute. SQL data types include:

- Character strings
- Numbers
- Boolean
- Temporal data such as dates and times
- Large character and binary object (clob and blob) data types.

String Types

1. Character string with fixed length $n$: character(n) or, equivalently, char(n).
2. Character string with variable length up to $n$: character varying(n) or, equivalently, varchar(n).

Numeric Types

1. Integer (exact) types:
   integer or, equivalently, int
   smallint (2 bytes in PostgreSQL)
2. Floating-point (inexact) types:
   real (single precision)
   double precision
3. Exact, arbitrary-precision types:
   numeric(p, s) where $p$ is the precision (total number of decimal digits) and $s$ is the scale (number of digits to right of decimal point). Good good for monetary amounts.
   Example: $-123.45$ is a value of type numeric(5, 2) (precision 5, scale 2).
Tables

Creating Tables

The `create table` statement defines a table. It specifies the table’s name and a list of attributes and their domains (types), and may also specify primary keys and data integrity constraints.

Syntax (simplified)

```
create table R ( 
    a_1 t_1, 
    ..., 
    a_n t_n, 
    G_1, 
    ..., 
    G_m 
); 
```

where \( R \) is the name of the table being created, \( a_i \) is the name of the \( i \)th attribute, \( t_i \) is the type of the \( i \)th attribute, and \( G_1, \ldots, G_m \) are table constraints.

Table Constraints

One type of table constraint is the primary key constraint. It is written as

```
primary key (a_1, \ldots, a_k) 
```

and means that attributes \( a_1, \ldots, a_k \) form the primary key. The DBMS then will not allow more than one tuple with the same primary key values.

Example 43

See see Listing 3.3.

Listing 3.3: The `create table` statement (create-basic.sql, source code).

```
-- Define the tables used in our examples.
-- (Basic usage)
-- 1/3/2005

5 create table alpha ( 
    name varchar(20), 
    age smallint, 
    primary key(name) 
); 

10 create table beta ( 
    name varchar(20), 
    age smallint, 
)
primary key (name)
);

create table product (
    pid char(5),
    pname varchar(20),
    length smallint,
    primary key (pid)
);

cREATE TABLE Distributor (
    did char(5),
    dname varchar(40),
    dcity varchar(20),
    primary key (did)
);

create table product_distributor (
    pid char(5),
    did char(5),
    quantity smallint,
    primary key (pid, did)
);

Remarks:

• "--" introduces a comment which continues to the end of the line.

• The create table statement creates empty tables. We have to populate them (fill them with tuples) separately. (Covered later)

Altering and Dropping Tables

The alter table statement changes characteristics of a table.

• Add/remove attributes

• Add/remove constraints

If you want to use it, see the reference manual for details.

The drop table statement deletes a table.

• Usage:

    drop table R;
3.4 Introduction to the SELECT Statement

The `SELECT` statement in SQL is a multi-purpose query statement, combining features of the relational algebra operations project, select, Cartesian product, and rename.

**As a Projection Operator**

The most basic form is

\[ \text{select } a_1, \ldots, a_n \text{ from } R; \]

where \( a_1, \ldots, a_n \) are the desired attributes from relation \( R \).

It is equivalent to \( \pi_{a_1, \ldots, a_n}(R) \).

**Example 44 Using the Select Statement for Projection.**

See see Listing 3.4.

Listing 3.4: SELECT as a projection operator.

```
test=# select name
    name
John
Catherine
George
Helen
   (4 rows)

test=# select name, age from alpha;
     name   |   age
     ---   | ---
   John   |  21
  Catherine |  23
     George  |  17
       Helen  |  15
   (4 rows)

test=# select * from alpha;
    name   |   age
    ---   | ---
   John   |  21
  Catherine |  23
     George  |  17
       Helen  |  15
   (4 rows)
```
### Chapter 3. SQL

#### 29
```sql
select * from product;
```

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(6 rows)

#### 39
```sql
select pid, pname from product;
```

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
</tr>
</tbody>
</table>

(6 rows)

#### 49
```sql
select pname from product;
```

<table>
<thead>
<tr>
<th>pname</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowboat</td>
</tr>
<tr>
<td>canoe</td>
</tr>
<tr>
<td>motorboat</td>
</tr>
<tr>
<td>motorboat</td>
</tr>
<tr>
<td>sailboat</td>
</tr>
<tr>
<td>sailboat</td>
</tr>
</tbody>
</table>

(6 rows)

#### 59
```sql
select distinct pname from product;
```

<table>
<thead>
<tr>
<th>pname</th>
</tr>
</thead>
<tbody>
<tr>
<td>canoe</td>
</tr>
<tr>
<td>motorboat</td>
</tr>
<tr>
<td>rowboat</td>
</tr>
<tr>
<td>sailboat</td>
</tr>
<tr>
<td>sailboat</td>
</tr>
</tbody>
</table>

(4 rows)

#### 69
```sql
select all pname from product;
```

<table>
<thead>
<tr>
<th>pname</th>
</tr>
</thead>
</table>


3.4. **INTRODUCTION TO THE SELECT STATEMENT**

Remarks:

1. "*" represents all attributes of a table.

2. SQL is not quite the same as relational algebra. One difference is, SQL does not automatically remove duplicates from the result of a select statement. To remove duplicates, say `select distinct`. To emphasize that duplicates should not be removed, say `select all`.

**As a Selection Operator**

**Example 45** See see Listing 3.5.

Listing 3.5: SELECT as a selection operator.

```
test=# select * from product where pname = 'motorboat';
pid | pname | length
-----+-------+-----
p3  | motorboat | 20
p4  | motorboat | 30
(2 rows)
```

```
test=# select * from product where length > 25;
pid | pname | length
-----+-------+-----
p4  | motorboat | 30
p6  | sailboat | 30
(2 rows)
```

Remarks:

- The notation "<>" means "is not equal to."

- Combinations using the boolean operators **and**, **or**, **not** are also permitted.
As a Project and Select Operator

Example 46 Naturally, the project and select aspects of the select statement can be combined.

Listing 3.6: SELECT as a projection and a selection operator combined.

```sql
44
2 test=# select length from product where pname = 'sailboat';
  length
  20
  30
  (2 rows)
7 test=# select pid, pname from product where length < 20;
   pid |    pname
     |----------
   p1 | rowboat
   p2 | canoe
  (2 rows)
17 test=# select length from product
       test=# where pname = 'sailboat' or pname = 'motorboat';
        length
          20
          30
        (4 rows)
27 test=# select distinct length from product
       test=# where pname = 'sailboat' or pname = 'motorboat';
            length
              20
              30
            (2 rows)
37 test=# select pid from product
       test=# where length < 30 and pname <> 'sailboat';
        pid
           p1
           p2
```
3.4. INTRODUCTION TO THE SELECT STATEMENT

p3
(3 rows)

```sql
test=# select pid from product
      test-# where length < 30 and not pname = 'sailboat';
pid
```

47 p1
p2
p3
(3 rows)

Remark:

- The last two queries in this example file mean the same thing.

### As a Cartesian Product Operator

The Cartesian product can be obtained by mentioning more than one relation in the `from` clause.

**Syntax:**

```sql
select a_1, ..., a_n
from R_1, ..., R_m
```

Equivalent to \( \pi_{a_1, \ldots, a_n}(R_1 \times \ldots \times R_m) \)

**Example 47** See see Listing 3.7.

Listing 3.7: SELECT as a Cartesian product operator.

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p3</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p3</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p4</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p5</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p6</td>
<td>d2</td>
<td>8</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p3</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p3</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p4</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p5</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p6</td>
<td>d2</td>
<td>8</td>
</tr>
</tbody>
</table>
As a Join Operator

Since natural join is definable in terms of Cartesian product, select (where common attributes are equal), and project (take the non-duplicate attributes), it is possible to form the natural join of two relations using the select statement.

Example 48 See see Listing 3.8.

Listing 3.8: SELECT as a join operator.

test=# select * from product, product_distributor
test-# where product.pid = product_distributor.pid
test-# ;

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
</tbody>
</table>

(42 rows)
### 3.4. INTRODUCTION TO THE SELECT STATEMENT

#### Example 49

Naturally, multiple joins are possible:

See see Listing 3.9.

Remarks:

- Attributes may be renamed by writing “as newname” in the select clause of the select statement.
- Similarly, relations can be renamed by writing “as newname” in the from clause.

```sql
test=# select product.pid as pid,
    test=#     pname, length, did, quantity
test-# from product, product.distributor
test-# where product.pid = product.distributor.pid;
```
Listing 3.9: Multiple joins using the SELECT statement.

```
test=# -- Multiple joins.
test=# -- Find names of distributors who distribute rowboats.
test=# select dname
    from product as p,
    product_distributor as pd,
    distributor as d
    where p.pid = pd.pid
    and d.did = pd.did
    and pname = 'rowboat';

   dname
----------
Marty's Marina
(1 row)

test=# -- Get product and distributor names where quantity > 10

    pname | dname
----------|--------
motorboat | Marty's Marina
sailboat  | Sarah's Sails
(2 rows)

test=# -- Names of products sold in St. Louis

    pname
----------
sailboat
(2 rows)
```
test=# select distinct pname
    from product as p,
    product_distributor as pd,
    distributor as d
    where p.pid = pd.pid
    and d.did = pd.did
    and dcity = 'St. Louis';

    ____________________________
    |    pname                  |
    |----------------------------|
    | sailboat                  |
    | (1 row)                   |
    ____________________________

57 test=# -- Another way (nested query)
    select pname
    from (select * from distributor
        where dcity = 'St. Louis')
        as d,
    product as p,
    product_distributor as pd
    where p.pid = pd.pid
    and d.did = pd.did;

    ____________________________
    |    pname                  |
    |----------------------------|
    | sailboat                  |
    | sailboat                  |
    | (2 rows)                  |
    ____________________________

72 test=# select distinct pname
    from (select * from distributor
        where dcity = 'St. Louis')
        as d,
    product as p,
    product_distributor as pd
    where p.pid = pd.pid
    and d.did = pd.did;

    ____________________________
    |    pname                  |
    |----------------------------|
    | sailboat                  |
    | (1 row)                   |
    ____________________________

Remark:

- This is not the only way to compute joins in SQL, as we shall see later. But perhaps for many people it is the easiest to understand, because it makes clear what is being selected and projected.
Comparing Strings

Order and Equality

String comparison can use any of the operators <, <=, =, <>, >, >=. Comparisons use alphabetical order. Comparisons are case-sensitive. The string functions upper and lower change case.

‘Like’ Comparisons

In addition, SQL allows us to compare strings using like, which permits a simple form of pattern matching.

Usage: string like 'pattern'

- _ in pattern matches any single character.
- % in pattern matches any string of zero or more characters.
- It is %, not *!
- Other characters in pattern match themselves.
- Case-sensitive.

Example 50 ‘Like’ comparisons

1. 'dog' is like 'dog'
2. 'Dog' is not like 'dog'
3. 'dog' is like '_og', '_g', '_'
4. 'cat' is like '%cat', '%at', '%t', 'c%t', 'ca%', 'cat%', etc.

Example 51 Using String Comparisons

See see Listing 3.10.

Listing 3.10: String comparisons.

gdweber=> select * from alpha;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>

(4 rows)

gdweber=> select * from alpha where name = 'john';

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
</table>

(2 rows)
3.4. INTRODUCTION TO THE SELECT STATEMENT

(0 rows)

gdweber⇒ `select * from alpha where upper(name) = 'JOHN';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(1 row)

gdweber⇒ `select * from alpha where lower(name) = 'john';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(1 row)

gdweber⇒ `select * from alpha where name > 'G';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>

(3 rows)

gdweber⇒ `select * from alpha where name <= 'G';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catharine</td>
<td>23</td>
</tr>
</tbody>
</table>

(1 row)

gdweber⇒ `select * from alpha where name like 'John';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(1 row)

gdweber⇒ `select * from alpha where name like 'Joh_';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(1 row)

gdweber⇒ `select * from alpha where name like 'Joh%';`

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(1 row)
Regular Expressions

SQL provides are more sophisticated form of pattern matching based on regular expressions, as well as a variety of other string functions and operations, but those are beyond the scope of this course.

Ordering of Tuples

In relational algebra, relations are sets, so the order of elements is immaterial. But SQL allows us to specify the order of tuples in an output table. We do this by using the ordered by clause in the select statement, listing the attributes to sort by. Optionally, we may qualify sort attributes by asc (ascending) or desc (descending). Default order: unspecified. Typically, SQL will present the tuples in whatever order it can generate them efficiently.

Example 52 See see Listing 3.11.

Listing 3.11: Sorting the results of a query.

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(6 rows)

Regular Expressions

SQL provides are more sophisticated form of pattern matching based on regular expressions, as well as a variety of other string functions and operations, but those are beyond the scope of this course.

Ordering of Tuples

In relational algebra, relations are sets, so the order of elements is immaterial. But SQL allows us to specify the order of tuples in an output table. We do this by using the ordered by clause in the select statement, listing the attributes to sort by. Optionally, we may qualify sort attributes by asc (ascending) or desc (descending). Default order: unspecified. Typically, SQL will present the tuples in whatever order it can generate them efficiently.

Example 52 See see Listing 3.11.

Listing 3.11: Sorting the results of a query.
### 3.4. INTRODUCTION TO THE SELECT STATEMENT

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(6 rows)

```sql
gdweber=> select * from product
        order by pname asc;
```

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(6 rows)

```sql
gdweber=> select * from product
        order bypname desc;
```

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
</tbody>
</table>

(6 rows)

```sql
gdweber=> select * from product
        order by pname asc, length desc;
```

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
</tr>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
</tr>
</tbody>
</table>

(6 rows)
3.5 Basic Set Operations

The set operations union, intersection, and difference are represented in SQL by the keywords union, intersect, and except. The operand sets must be expressed as select statements. By default, the set operations remove duplicates from the result. To retain duplicates, use the word all, e.g., union all.

Example 53 See see Listing 3.12.

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(6 rows)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(8 rows)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(2 rows)

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 3.12: Set operations.
3.6 Aggregation

The select statement can be used to perform aggregation. Aggregate functions are \texttt{avg}, \texttt{min}, \texttt{max}, \texttt{sum}, \texttt{count}, \texttt{stddev}, \texttt{variance}.

If applied to an empty set of tuples, \texttt{count} returns 0; all others return \texttt{null}.

By default, duplicates are retained. For example, two 20’s would be summed as 40 and counted as 2. Use the world \texttt{distinct} to eliminate duplicates.

Each function may be applied to a single attribute. The \texttt{count} function may also be applied to \texttt{*}, meaning just count the tuples.

Example 54 Basic aggregation: See see Listing 3.13.

Listing 3.13: Basic aggregation.

```
gdweber=> \texttt{select length from product;}
length
10
5
total | average | n
| | | |
120 | 20.0000000000 | 6
(1 row)
```

```
20 gdweber=> \texttt{select sum(d\texttt{is\texttt{tinct}} length) as total, avg(d\texttt{is\texttt{tinct}} length) as average,}
gdweber=> \texttt{count(d\texttt{is\texttt{tinct}} length) as N}
gdweber=> \texttt{from product;}
total | average | n
| | | |
60 | 20.0000000000 | 3
(1 row)
```
Aggregation with Grouping

Aggregate values can be grouped. For example, we may want the average boat length for each type (pname) of boat—rowboats, canoes, motorboats, and sailboats. Use the group by clause. Attributes that are not aggregated must be used for grouping. We can also select from the groups, using the having clause.

Example 55 See see Listing 3.14.

Listing 3.14: Aggregation with grouping.

gdweber=> select count(*) from alpha;

    count

      4

(1 row)

gdweber=> selectpname, avg(length) from product

gdweber=> group by pname;

<table>
<thead>
<tr>
<th>pname</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>canoe</td>
<td>10.0000000000</td>
</tr>
<tr>
<td>motorboat</td>
<td>25.0000000000</td>
</tr>
<tr>
<td>rowboat</td>
<td>10.0000000000</td>
</tr>
</tbody>
</table>

(4 rows)

ERROR: Attribute product.pname must be GROUPed or used in an aggregate function

gdweber=> select pname, avg(length) as avglength from product

where length < 20;

<table>
<thead>
<tr>
<th>pname</th>
<th>avglength</th>
</tr>
</thead>
<tbody>
<tr>
<td>canoe</td>
<td>10.0000000000</td>
</tr>
<tr>
<td>rowboat</td>
<td>10.0000000000</td>
</tr>
</tbody>
</table>

(2 rows)
3.6. AGGREGATION

```sql
gdweber⇒ select pname, avg(length) as avglength
from product
where avglength > 15;
ERROR: Attribute 'avglength' not found
```

```sql
gdweber⇒ select pname, avg(length) as avglength
from product
where avglength > 15;
pname | avglength
motorboat | 25.00000000000
sailboat | 25.00000000000
(2 rows)
```

### How Do ‘Where’ and ‘Having’ Work Together?

Since the `where` and `having` clauses both, in a sense, select rows, it is important to clarify the differences between them.

Consider the query

```sql
select a, sum(b)
from R
where C
group by a
having P
```

where `a` and `b` are attributes, `R` is a relation, and `C` and `P` are conditions (i.e., boolean expressions). Logically speaking, the various parts of this query can be thought of as a series of “filters” which are applied in the following order:

1. Apply the `where` clause to select tuples from `R`: keep those that satisfy `C`.
2. Apply grouping to tuples from the previous stage: join tuples into groups with same `a` values.
3. Apply aggregation to the groups from the previous stage: compute the `sum` of `b` for each group.
4. Apply the `having` clause to select groups: keep the groups that satisfy `P`.

Thus, the `where` clause is applied before aggregation, to select `tuples`; the `having` clause is applied after aggregation, to select `groups`.
3.7 Nulls

Recall that null, representing the absence of any known value, introduces complications into logic.

Any SQL logical or comparison operation with null has value unknown.

In particular, (null = null) is unknown since the first null might stand for a different unknown value from the second.

The select statement’s where clause chooses tuples for which its condition is true, not unknown or false.

Consequently, if we want to find the tuples where some attribute a is null, we must use

where a is null

We cannot use

where a = null

for the result would unknown.

Similarly, we can use the predicates is not null, is unknown, is not unknown.

Examples: see Listing 3.15.

Listing 3.15: Nulls.

```sql
gdweber=> 
Testing for null

gdweber=> select * from alpha;
   name | age
-------+-----
    John |  21
 Catherine |  23
    George |  17
     Helen |  15
     (4 rows)

gdweber=> select name from alpha
          where age = null;

name

(0 rows)

gdweber=> select name from alpha
          where age <> null;

name

(0 rows)

gdweber=> select name from alpha
          where age is not null;

name
```

58
3.8 Additional Operations on Subqueries

In addition to the set operations of \texttt{union}, \texttt{intersect}, and \texttt{except} (set difference), SQL provides a number of set or set-related operations that are applied to the results of subqueries. A subquery is a query within a query, often found in the \texttt{where} clause of the outer query. These operations may make some kinds of queries easier to express. They do not extend the expressive power of the language. That is, we could do without them; it just might not be as convenient. Or then again, maybe it would be.

- Test for set membership (\texttt{in}).
  Usage: \texttt{where }E\ [\texttt{not}] \texttt{in } (S)

- Compare to some.
  Usage example: \texttt{where }E > \texttt{some } (S)

- Compare to all.
  Usage example: \texttt{where }E > \texttt{all } (S)

- Test for non-empty set (\texttt{exists}).
  Usage: \texttt{where }[\texttt{not}] \texttt{exists } (S)

- Test for uniqueness (i.e., no duplicate elements) (\texttt{unique}).
  Usage: \texttt{where }[\texttt{not}] \texttt{unique } (S)

where \(E\) is any expression of appropriate type and \(S\) is a subquery (i.e., a select statement).

The brackets [] indicate that the word \texttt{not} is optional. They are not part of the statement.

The compare to some and compare to all allow any of the six comparison operators \(<\), \(\leq\), \(=\), \(\neq\), \(>\), \(\geq\).\n
\textbf{Example 56} See see Listing 3.16.

Listing 3.16: Subqueries.

```
gdweber=> --- Set-like operations on subqueries.
gdweber=> --- Data:
gdweber=> select * from product;
pid | pname | length
-- | ---- | ----
p1 | rowboat | 10
```
CHAP TE R  3. SQL

<table>
<thead>
<tr>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p4</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p5</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p6</td>
<td>d2</td>
<td>8</td>
</tr>
</tbody>
</table>

(6 rows)

\[ \text{gdweber} \Rightarrow \text{select } \ast \text{ from product_distributor;} \]

\[ \text{did } | \text{dname } | \text{dcity} \]

<table>
<thead>
<tr>
<th>did</th>
<th>dname</th>
<th>dcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Peter’s Powerboats</td>
<td>Chicago</td>
</tr>
<tr>
<td>d2</td>
<td>Sarah’s Sails</td>
<td>St. Louis</td>
</tr>
<tr>
<td>d3</td>
<td>Marty’s Marina</td>
<td>Mobile</td>
</tr>
</tbody>
</table>

(3 rows)

\[ \text{gdweber} \Rightarrow \text{select } \ast \text{ from distributor;} \]

\[ \text{pid } | \text{did } | \text{quantity} \]

<table>
<thead>
<tr>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p5</td>
<td>d2</td>
<td>12</td>
</tr>
</tbody>
</table>

(2 rows)

\[ \text{gdweber} \Rightarrow \text{select } \ast \text{ from product_distributor; } \]

\[ \text{gdweber} \Rightarrow \text{select } \text{pname } \text{from product; } \]

\[ \text{gdweber} \Rightarrow \text{where } \text{pid in } \text{(select pid from product_distributor; } \]

\[ \text{gdweber(> from product_distributor; } \]

\[ \text{gdweber(> where quantity > 10; } \]

\[ \text{pname} \]
3.8. ADDITIONAL OPERATIONS ON SUBQUERIES

motorboat
sailboat
(2 rows)

gdweber=> Another expression of the same query
gdweber=> select pname

gdweber=> from product natural join
gdweber=> product_distributor
gdweber=> where quantity > 10;

gdweber=>
motorboat
sailboat
(2 rows)

--- More data

gdweber=> select * from alpha;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>
(4 rows)

---

gdweber=> select * from beta;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
</tbody>
</table>
(4 rows)

---

gdweber=> Find names of people in alpha who are older
gdweber=> than somebody in beta.

gdweber=> select name
gdweber=> from alpha
gdweber=> where age > some (select age from beta);

---

---

gdweber=> Same query another way

---
CHAPTER 3. SQL

99  gdweber> select a.name
    gdweber> from alpha as a, beta as b
    gdweber> where a.age > b.age;
    name

104  John
    Catherine
    Catherine
    (3 rows)

109  gdweber> select distinct a.name
    gdweber> from alpha as a, beta as b
    gdweber> where a.age > b.age;
    name

114  Catherine
    John
    (2 rows)

119  gdweber> — Find names of people in beta who are older
    gdweber> — than everybody in alpha
    gdweber> select name
    gdweber> from beta
    gdweber> where age > all (select age from alpha);
    name

124  Carol
    (1 row)

129  gdweber> — Another way
    gdweber> select name
    gdweber> from beta as b,
    gdweber> (select max(age) as amaxage
    gdweber> from alpha)
    gdweber> where age > amaxage;

134  ERROR: sub-SELECT in FROM must have an alias
For example, FROM (SELECT ...) [AS] foo
    gdweber> select name
    gdweber> from beta as b,
    gdweber> (select max(age) as amaxage
    gdweber> from alpha)
    gdweber> as amax
    gdweber> where age > amaxage;
    name

144  Carol
3.9. Views

A *view* is a “virtual relation” which either enhances or restricts a particular user’s or a group of users’ perception of what is physically stored in the database. It is not a “real” relation, in the sense that it is not stored permanently on disk; but it has the appearance of being real to the users. A view can *enhance* a user’s perception of the database by adding attributes to make the relation more easily understood. A view can *restrict* a user’s perception of the database by hiding attributes. This is done for security. A view may both restrict and enhance the user’s perception.
The \textit{create view} statement defines a view. The \textit{drop view} statement deletes a view definition. The \textit{alter view} statement modifies a view definition. Syntax:

\begin{verbatim}
create view \textit{V} as \textit{Q} 
drop view \textit{V}
\end{verbatim}

where \textit{V} is the name of the view and \textit{Q} is a select query expression.

\textbf{Example 57} See see Listing 3.17.

\begin{verbatim}
Listing 3.17: The \textit{create view} statement.
gdweber=> \textit{Some views} 
2 gdweber=> \textbf{select} * \textbf{from} alpha;
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catharine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
</tbody>
</table>
(4 rows)

gdweber=> \textbf{select} * \textbf{from} beta;
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Catharine</td>
<td>23</td>
</tr>
</tbody>
</table>
(4 rows)

gdweber=> \textbf{create view} gamma \textbf{as} 
gdweber-> (\textbf{select} * \textbf{from} alpha) \textbf{union} (\textbf{select} * \textbf{from} beta);
22 \text{CREATE}

gdweber=> \textbf{select} * \textbf{from} gamma;
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Catharine</td>
<td>23</td>
</tr>
<tr>
<td>George</td>
<td>17</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Tim</td>
<td>18</td>
</tr>
</tbody>
</table>
(6 rows)
\end{verbatim}
gdweber=> select *
37 gdweber=> from gamma
gdweber=> where age >= 20;

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>24</td>
</tr>
<tr>
<td>Catherine</td>
<td>23</td>
</tr>
<tr>
<td>John</td>
<td>21</td>
</tr>
</tbody>
</table>

(3 rows)

gdweber=> create view ppdd as
47 gdweber=> select *
47 gdweber=> from product natural join
47 gdweber=> product_distributor natural join
gdweber=> distributor;

CREATE

52 gdweber=> select * from ppdd;

<table>
<thead>
<tr>
<th>did</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>p3</td>
<td>motorboat</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>d1</td>
<td>p4</td>
<td>motorboat</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>d2</td>
<td>p5</td>
<td>sailboat</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>d2</td>
<td>p6</td>
<td>sailboat</td>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>d3</td>
<td>p1</td>
<td>rowboat</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>d3</td>
<td>p2</td>
<td>canoe</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>d3</td>
<td>p3</td>
<td>motorboat</td>
<td>21</td>
<td>20</td>
</tr>
</tbody>
</table>

57

<table>
<thead>
<tr>
<th>dname</th>
<th>dcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter’s Powerboats</td>
<td>Chicago</td>
</tr>
<tr>
<td>Peter’s Powerboats</td>
<td>Chicago</td>
</tr>
</tbody>
</table>

62

57 Sarah’s Sails | St. Louis
| Sarah’s Sails  | St. Louis
| Marty’s Marina | Mobile
| Marty’s Marina | Mobile
| Marty’s Marina | Mobile

72 (7 rows)

gdweber=> select pname, dname, dcity, quantity
gdweber=> from ppdd

77 gdweber=> where (pname = 'motorboat' or pname = 'sailboat')

<table>
<thead>
<tr>
<th>pname</th>
<th>dname</th>
<th>dcity</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>motorboat</td>
<td>Marty’s Marina</td>
<td>Mobile</td>
<td>20</td>
</tr>
<tr>
<td>sailboat</td>
<td>Sarah’s Sails</td>
<td>St. Louis</td>
<td>12</td>
</tr>
</tbody>
</table>
View Storage and Mutation

How views are stored and modified, if at all, is system-dependent. In PostgreSQL, views are not materialized (i.e., they are not “permanently” stored, but generated as needed for queries), and they are read-only (i.e., no insert, update, or delete operation is permitted on a view).

View Implementations

Views are normally not stored permanently in the database, but are computed on the fly as needed (i.e., when somebody executes a query that uses the view, the view is created in temporary storage for the duration of that query). I.e., a defined view is treated as a mere abbreviation for its defining expression, like a macro in a program. Some database systems do allow views to be stored like a permanent relation (relation variable). Such views are called materialized views, and the DBMS must take responsibility to keep the view up to date whenever its source relations are modified.

Mutability of Views

Can one insert, delete, or update tuples in a view? Unrestricted mutability of views creates problems involving nulls, at least for views which exclude some of the information in their sources. E.g., inserting the tuple (hovercraft, Herbert's, 10) into the view NNPD above requires changes to the product, distributor, and product-distributor relations. But the insertion in the product relation must have null values for pid and length, since these attributes are not part of NNPD; and similarly in distributor and product-distributor. To make matters worse, when the view is recomputed from the sources, the tuple which was “inserted” will not be present, because of the way it treats null values. Because of problems like this, database systems allow mutation of views only in restricted conditions. Each DBMS has its own rules about what kind of view updates to allow.

3.10 Insertion

There are three forms of the `insert` statement.

Syntax and semantics:
3.10. INSERTION

1. \texttt{insert into } R \texttt{values } (v_1, \ldots, v_k) \\
   The first form specifies a tuple to be inserted as a list of attribute values. 
   The values are given in the same order as the attributes occur in the 
   relation. 
   This is apt to be confusing, since one may forget the order of the relation’s 
   attributes.

2. \texttt{insert into } R(a_1, \ldots, a_k) \texttt{values } (v_1, \ldots, v_k) \\
   The second form is similar, but allows the user to specify the order of the 
   attributes.

3. \texttt{insert into } R \texttt{select} . . . \\
   The third form inserts the results of a \texttt{select} query. It may insert any 
   number of tuples.

Example 58

- For examples of the first two forms, see see Listing 3.18.
- For an example of the third form, see SKS, pp. 159–160.

Listing 3.18: The SQL \texttt{insert} statement.

\begin{verbatim}
12 gdweber=> --- Mutation of the database
2 gdweber=> --- Insertion
3 gdweber=> select * from product;
         pid |       pname    | length 
--------+----------------+--------
        p1 |     rowboat    | 10
        p2 |       canoe    | 10
        p3 |  motorboat     | 20
        p4 |  motorboat     | 30
        p5 |     sailboat   | 20
        p6 |     sailboat   | 30
12 (6 rows)
5 gdweber=> insert into product
6 gdweber=> values (p7, raft, 15);
ERROR: Attribute 'p7' not found
17 gdweber=> insert into product
17 gdweber=> values ('p7', 'raft', 15);
INSERT 16772 1
\end{verbatim}
3.11 Updates

Syntax:

\[
\text{update } R \\
\text{set } a_1 = v_1, \ldots, a_j = v_j \\
\text{where } P
\]

where \( R \) is a relation, \( P \) is a predicate (condition), \( a_i \) is the \( i \)th attribute, and \( v_i \) is the corresponding new value \( (1 \leq i \leq j) \). The attribute-value pairs may be given in any order, and there may be any number of them up to the number
3.12. DELETION

The values may be given as expressions (e.g., arithmetic expressions); they don’t have to be constants.

Semantics: An update can affect zero, one, or more tuples of the \( R \), depending on how many of them satisfy \( P \). There where clause is optional; if omitted, every tuple in \( R \) is updated.

### 3.12 Deletion

**Syntax:**

\[
\text{delete from } R
\text{ where } P
\]

where \( R \) is a relation and \( P \) is a predicate (condition). **Semantics:** Tuples which satisfy \( P \) are removed from \( R \). The delete statement can remove zero, one, or more tuples from \( R \), depending on how many satisfy \( P \).

**Caution.** The where clause is optional; if omitted, all tuples are removed from \( R \). Be warned: the system will not ask you for confirmation before doing this, nor will it allow you to “undo” the operation!

**Example 59 Updating and Deleting.** See see Listing 3.19.

Listing 3.19: The SQL update and delete statements.

```sql
1 gdweber=> select * from product;
2 pid | pname   | length
3 p1  | rowboat | 10
4 p2  | canoe   | 10
5 p3  | motorboat | 20
6 p4  | motorboat | 30
7 p5  | sailboat | 20
8 p6  | sailboat | 30
9 p7  | raft    | 15
10 p8 | rubber duck | 1
11 p9 | rubber duck | 2
12 (9 rows)
13 gdweber=> update product
14 gdweber=> set length = length + 1
15 gdweber=> where pname = 'rubber duck';
16 UPDATE 2
17 gdweber=> select * from product;
18```

### 3.13 The Case Expression

Useful in update statements, and in other contexts (e.g., in a select statement).

Syntax:

```sql
case
    when C_1 then E_1
    when C_2 then E_2
    ...
    when C_k then E_k
    else E_{k+1}
end
```

where $C_i$ is the $i$th condition, and $E_i$ is the $i$th value (expression), the last (with else) being the default value. Semantics:

- Find first condition $C_i$ which is true; evaluate corresponding $E_i$.
- If none is true, evaluate $E_{k+1}$.
3.13. THE CASE EXPRESSION

It is like the C++ or Java switch statement.

*Quirks:* psql seems not to like having when on a new line.

**Example 60** See see Listing 3.20.

```
Listing 3.20: case.sio, source code.
gdweber=> --- Using the CASE conditional expression
gdweber=> select * from product
gdweber=> where length >
gdweber=> case
  when pname = 'rowboat' or pname = 'canoe'
  then 8
  else 24
end;

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
</tr>
</tbody>
</table>

(4 rows)

gdweber=> update product
gdweber=> set length = case
  when pname = 'rowboat' or
  then length - 1
  when pname = 'motorboat'
  then length + 1
  else
  length + 3
end;

UPDATE 6

gdweber=> select * from product;

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>9</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>9</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>21</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>31</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>23</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>33</td>
</tr>
</tbody>
</table>

(6 rows)
```
3.14 Transactions

A transaction is a sequence of operations that should either all succeed or all fail. I.e., if any of the actions fails, the others should fail too.

Example: moving a dollar from one account to another involves decrementing the balance of one account and incrementing the other. If one, but not the other, happens, there’s a problem.

SQL supports transactions through the commit and rollback statements:

1. According to the standard, a transaction begins implicitly with any statement.

2. The commit (or commit work) statement terminates a transaction (successfully).

3. The rollback (or rollback work) statement is used when an error is detected; it is an “undo” command for the previous steps of the transaction.

(Details in a later chapter)

3.15 Joins Revisited

Recall that

\[
\text{select distinct } a_1, \ldots, a_n \\
\text{from } R_1, \ldots, R_m \\
\text{where } C
\]

is equivalent to the relational algebra

\[
\pi_{a_1, \ldots, a_n}(\sigma_C(R_1 \times \ldots \times R_m))
\]

One way to compute the natural join of two relations, then, is to use \( C \) to test for equality of all shared attributes.

SQL also allows the word join and related keywords to be used in the from clause, to express a variety of kinds of join. Options include:

- \( R_1 \textbf{ natural join } R_2 \)
- \( R_1 \textbf{ join } R_2 \textbf{ using } (a_i, a_j) \)
- \( R_1 \textbf{ left outer join } R_2 \textbf{ on } R_1.a_i = R_2.a_j \)
- \( R_1 \textbf{ natural full outer join } R_2 \)
- \( R_1 \textbf{ inner join } R_2 \textbf{ on } R_1.a_i < R_2.a_j \)

where \( R_1, R_2 \) are relations, and \( a_i, a_j, \text{ etc.} \), are attribute names.
Join Types and Conditions

To make some sense of the numerous possible combinations, let us look separately at join types and join conditions. Join conditions specify how to match tuples. Join types specify what to do with non-matching tuples.

Join Types

The join types are [inner] join, left [outer] join, right [outer] join, full [outer] join, with brackets [] around optional words.

Inner join (default) discards all non-matching tuples. Left outer join includes non-matching tuples from the left relation $R_1$, filling in null for the attributes not found in $R_2$. Right outer join does the corresponding thing for the right relation $R_2$. Full outer join includes non-matching tuples from both $R_1$ and $R_2$.

Join Conditions

The join conditions are specified by one of the words natural, on, and using. Only one of these is used for a particular join.

- **natural** means that matching tuples are those having equal values for all the attributes common to both $R_1$ and $R_2$. It is written to the left of the join type, e.g., $R_1$ natural inner join $R_2$.

- **using** specifies a list of shared attributes which must be equal to make a match. It is written to the right of the join type, e.g., $R_1$ inner join $R_2$ using $(a_1, a_2)$. The join includes tuples in which

  $$R_1.a_1 = R_2.a_1 \land R_1.a_2 = R_2.a_2.$$  

- **on** specifies a predicate (boolean expression) for selecting the matched tuples. It is written to the right of the join type, e.g., $R_1$ inner join $R_2$ on $R_1.a_1 = R_2.a_2$.

  It is the most general type of join condition, since (a) the attributes compared may have different names, and (b) the comparison may be something other than equality.

  Note that the “on” version does not eliminate duplicate columns.

Many Options

SQL provides so many options here that there are typically more than one way to express any desired join.

- Any natural join may be expressed with using.

- Any join expressed with using may be expressed with on.\(^1\)

---

\(^1\)Except for duplicate column elimination.
• Any join whatsoever may be expressed without using the word `join`, just using the features of the `select` statement that perform the \(\pi, \sigma, \times\) operations.

**Example 61 Join Revisited**

See see Listing 3.21.

Listing 3.21: Additional join operations with the `SELECT` statement.

```sql
gdweber=> -- Joins revisited
2  gdweber=> select * from product;
    pid |   pname | length
  ---+--------+-------
    1 |  rowboat |   10
    2 |    canoe |   10
    3 | motorboat |   20
    4 | motorboat |   30
    5 |  sailboat |   20
    6 |  sailboat |   30
(6 rows)
```

```sql
12  gdweber=> select * from product.distributor;
    pid |   did | quantity
  ---+--------+-------
    1 |  d3    |   10
    2 |  d3    |   10
    3 |  d1    |   10
    4 |  d3    |   20
    5 |  d2    |    5
    6 |  d2    |    8
(7 rows)
```

```sql
gdweber=> select product.pid as pid,
    gdweber=>      pname, length, did, quantity
27  gdweber=> from product, product.distributor
    gdweber=> where product.pid = product.distributor.pid;
    pid |   pname | length |   did | quantity
  ---+--------+-------+-------+-------
    1 | rowboat |    10 |  d3   |    10
    2 |     canoe |   10 |  d3   |    10
    3 | motorboat |   20 |  d1   |    10
    4 | motorboat |   20 |  d3   |    20
    5 |  sailboat |   30 |  d1   |    5
    6 |  sailboat |   30 |  d2   |    12
(7 rows)
```
3.15. JOINS REVISITED

```
gdweber=> select *  
gdweber-> from product natural join product_distributor;  

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
<td>d2</td>
<td>8</td>
</tr>
</tbody>
</table>

(7 rows)
```

```
gdweber=> select *  
gdweber-> from product inner join product_distributor  
gdweber-> using (pid);  

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
<td>d2</td>
<td>8</td>
</tr>
</tbody>
</table>

(7 rows)
```

```
gdweber=> select *  
gdweber-> from product inner join product_distributor  
gdweber-> on product.pid = product_distributor.pid;  

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>length</th>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>rowboat</td>
<td>10</td>
<td>p1</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p2</td>
<td>canoe</td>
<td>10</td>
<td>p2</td>
<td>d3</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>p3</td>
<td>d1</td>
<td>10</td>
</tr>
<tr>
<td>p3</td>
<td>motorboat</td>
<td>20</td>
<td>p3</td>
<td>d3</td>
<td>20</td>
</tr>
<tr>
<td>p4</td>
<td>motorboat</td>
<td>30</td>
<td>p4</td>
<td>d1</td>
<td>5</td>
</tr>
<tr>
<td>p5</td>
<td>sailboat</td>
<td>20</td>
<td>p5</td>
<td>d2</td>
<td>12</td>
</tr>
<tr>
<td>p6</td>
<td>sailboat</td>
<td>30</td>
<td>p6</td>
<td>d2</td>
<td>8</td>
</tr>
</tbody>
</table>

(7 rows)
```

```
gdweber=> -- Find people with same name in alpha and beta  
gdweber=> select *  
gdweber-> from alpha inner join beta  
gdweber-> on alpha.name = beta.name;
```
CHAPTER 3. SQL

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td></td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

```sql
SELECT alpha.name as aname, alpha.age as aage, beta.name as bname, beta.age as bage
FROM alpha INNER JOIN beta
ON alpha.name = beta.name;
```

ERROR: Attribute 'aname' not found

```sql
SELECT * FROM alpha INNER JOIN beta USING (age);
```

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helen</td>
<td>15</td>
<td>Tim</td>
<td>18</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
<td>John</td>
<td>21</td>
</tr>
<tr>
<td>Helen</td>
<td>15</td>
<td>Catherine</td>
<td>23</td>
</tr>
</tbody>
</table>

(11 rows)
Chapter 4

Advanced SQL

References:

- Silberschatz, Korth, and Sudarshan, Chapter 4.
- The PostgreSQL Global Development Group, *PostgreSQL 8.1.0 Documentation*, 2005:
  - I. Tutorial (entire).
  - II. The SQL Language.
  - IV. Client Interfaces.
  - VI. Reference, Part I, “SQL Commands.”

4.1 More SQL Data Types

In addition to the numeric and character data types presented in the previous chapter, SQL types include:

- Boolean
- Temporal data
- Other types.

**Boolean Type**

Values of type `boolean`: `true`, `false`.

**Temporal Data Types**

2. **time.** Examples: \texttt{time '01:15:00'} (1:15 a.m.), \texttt{time '15:30:09'} (9 seconds after 3:30 p.m.).

3. **timestamp** combines date and time. Example: \texttt{timestamp '2003-02-21 15:30:09'}.

4. Temporal data types are complicated due to the wide variety of input and output formats and the possibility of time zones.

5. Note: the word \texttt{date}, \texttt{time}, or \texttt{timestamp} is required.

### Other Data Types

The string, numeric, boolean, and temporal types are the most important SQL standard ('92 and '99) types. Other types, either standard or implementation-specific (e.g., PostgreSQL), include large character and binary objects, geometric types, and network address types. Users can also define new types with the \texttt{create type} statement.

#### 4.2 Tables

### Creating Tables

The \texttt{create table} statement allows column constraints as well as table constraints. A column constraint is a restriction on the data of a particular column. It is written after the attribute type (underscored in the syntax description below).

#### Syntax

```
create table R (  
  a_1 t_1 [C_{i1}, \ldots],  
  \ldots,  
  a_n t_n [C_{i1}, \ldots],  
  G_1,  
  \ldots,  
  G_m  
);  
```

where \( R \) is the name of the table being created, \( a_i \) is the name of the \( i \)th attribute, \( t_i \) is the type of the \( i \)th attribute, \( C_{ij} \) are its column constraints (1 \( \leq i \leq n \)), and \( G_1, \ldots, G_m \) are table constraints.

#### Constraints

Both the column and table constraints are integrity constraints. They both express similar kinds of constraints, but each column constraint applies to a single attribute. A table constraint may apply to one or more attributes.
A constraint involving more than one attribute must be written as table constraints. A constraint involving only one attribute may be written as either a column constraint or a table constraint.

Column constraints may include:

- **primary key**
- **not null**
- **check** \( P \)
  - \( P \) is a boolean expression that must be true of the attribute value.
- **references** \( T \) or **references** \( T(k) \)
  - Attribute references a foreign key; \( T \) = referenced relation, \( k \) = referenced key attribute in \( T \).
  - If \( k \) is unspecified, the referenced key attribute has the same name as the referencing attribute.

Table constraints may include:

- **primary key** \( (a_1, \ldots, a_k) \), covered in the Coursepack Chapter 3.
- **check** \( (P) \)
  - \( P \) is a boolean expression that must be true of every tuple in \( R \).
  - Examples: **check** \((\text{age} \geq 0)\), **check** \((\text{length} > 0)\), **check** \((\text{pid} \text{ like } 'p\%')\).

**Populating Tables**

When a table is “created,” it is empty. It can be populated with tuples either by the **insert** statement or the **copy** statement. Although **copy** is not part of the DDL of SQL, we cover it here because it is useful for filling newly created tables with data from files. It can also work in the opposite direction. PostgreSQL extension; do other DBMSs have similar commands? Usage:

```
copy R from 'inputfilename';
copy R to 'outputfilename';
```

There is also the **psql** \texttt{\copy} command, like **copy** but using files on the database client, not server.

**Example 62** See Listing 4.1.

Listing 4.1: The **create table** statement (create.sql, source code).

```sql
create table alpha ( 
  name varchar(20) primary key,
  age smallint
) 
```

---

2/21/2003
create table beta (name varchar(20) primary key,
age smallint);

create table product (pid char(5) primary key,
pname varchar(20),
length smallint);

create table distributor (did char(5) primary key,
dname varchar(40),
dcity varchar(20));

create table product_distributor (pid char(5) references product,
did char(5) references distributor,
quantity smallint,
primary key (pid, did));

-- Load tables from files
\copy alpha from './alpha.data'
\copy beta from './beta.data'
\copy product from './product.data'
\copy distributor from './distributor.data'
\copy product_distributor from './product_distributor.data'

4.3 Integrity

Integrity constraints protect the database against unintended damage to data, e.g., input errors. (Security measures, to protect against intentional damage, are covered in a later chapter.)

This presentation will concentrate on aspects of integrity which are supported by both standard SQL ('92 or 1999) and PostgreSQL. Secondary attention will be given to standard SQL features not implemented in PostgreSQL, and PostgreSQL features which are not standard SQL.

For details of any SQL command implemented in PostgreSQL, see the “SQL Commands” of the PostgreSQL Reference Manual.
4.3. INTEGRITY

Concepts

Domain constraints: An attribute’s value must belong to a particular domain or type. Examples: Name must be a string of up to 40 characters. age must be a small integer. Domains are like the data types of programming languages, but they may impose additional rules. Examples: Product ID must be the letter P followed by two digits. Salary must be a positive monetary amount (i.e., a number precise to 1/100’s).

Key constraints: A primary key must have unique, non-null values.

Example 63 The product tuple (null, raft, 10) is invalid. The two tuples (p1, cabinruser, 31),(p1, raft, 10) are not both allowed in product.

Referential integrity constraints: When attribute A in relation R₁ references a foreign key in relation R₂, the foreign key value must exist in R₂.

Example 64 In product-distributor table, the pid attribute references the key of the product relation, and did references the key of the distributor relation. A tuple (p7, d3, 10) in product-distributor would be in error if: There were no tuple pid = p7 in product. Or if there were no tuple with did = d3 in distributor. (**insert diagram?**)

SQL Support

Standard Features Implemented in PostgreSQL

Constraints in the create table Statement.

- Constraint types supported as both attribute and table constraints: unique, primary key, check, foreign key references.
- Constraints supported only as attribute constraints: null (!), not null.

Example 65 Constraint definitions in create table.

1. Using attribute constraints only: see Listing 4.2.

2. Using a mixture of table and attribute constraints (mostly table constraints): see Listing 4.3.

Listing 4.2: integrity-1.sql, source code.

```sql
create table club (...
```
create table club (club_id varchar(20) primary key, 
    club_name varchar(40) unique)
);

create table officer ( 
    name varchar(20) primary key,
    age smallint not null check(age >= 2 and age < 140),
    office varchar(30) unique,
    club_id varchar(20) references club
);

— Populate club table

insert into club(club_id, club_name) 
values('C21', 'Computer Club');

— Duplicate key error

insert into club(club_id, club_name) 
values('C21', 'Psychology Club');

— Populate officer table

insert into officer(name, age, office, club_id) 
values('John Smith', 21, 'President', 'C21');

— Invalid age

insert into officer(name, age, office, club_id) 
values('Sandy Jones', 1, 'Vice President', 'C21');

— Invalid club reference

insert into officer(name, age, office, club_id) 
values('Sandy Jones', 12, 'Vice President', 'C22');

— Get it right

insert into officer(name, age, office, club_id) 
values('Sandy Jones', 12, 'Vice President', 'C21');

drop table officer;
drop table club;

Listing 4.3: integrity-2.sql, source code.
4.3. INTEGRITY

— Create tables with integrity constraints
— using table constraints:

```sql
create table club (
    club_id varchar(20),
    club_division smallint,
    club_name varchar(40),
    primary key(club_id, club_division),
    unique(club_name)
);
```

```sql
create table officer (
    name varchar(20),
    age smallint not null,
    office varchar(30),
    club_id varchar(20),
    club_div smallint,
    primary key(name),
    check(age >= 2 and age < 140),
    unique(office),
    foreign key(club_id, club_div) references club
);
```

— Populate club table

```sql
insert into club(club_id, club_division, club_name) values ('C21', 6, 'Computer Club');
```

— Duplicate key error

```sql
insert into club(club_id, club_division, club_name) values ('C21', 6, 'Psychology Club');
```

— Populate officer table

```sql
insert into officer(name, age, office, club_id, club_div) values ('John Smith', 21, 'President', 'C21', 6);
```

— Invalid age

```sql
insert into officer(name, age, office, club_id, club_div) values ('Sandy Jones', 1, 'Vice President', 'C21', 6);
```

— Invalid club reference

```sql
insert into officer(name, age, office, club_id, club_div) values
```
values ('Sandy Jones', 12, 'Vice President', 'C21', 5);

−− Get it right

insert into officer(name, age, office, club_id, club_div) values ('Sandy Jones', 12, 'Vice President', 'C21', 6);

drop table officer;

drop table club;

Update and Delete with Referential Integrity Constraints.

Updating or deleting tuples may cause violations of referential integrity constraints. Suppose table A’s key is referenced by table B. Then updating or deleting a tuple of A may cause tuples of B to become invalid, since the key values they refer to no longer exist in A.

Example 66 In the relation product-distributor(pid, did, quantity), pid references product(pid, pname, length), and did references distributor(did, dname, city). Suppose product-distributor contains the tuples (p21, d19, 15) and (p22, d35, 20). Then there must be tuples in product with pid = p21 and pid = p22. Also there must be tuples in distributor with did = d19 and did = d35. What happens if one of these tuples is deleted from product or distributor? What happens if it is updated, changing the primary key?

What to Do About It. The create table statement allows specifying what to do in such cases, using the on update and on delete clauses. Options include:

- cascade: the operation propagates to the other table: for a delete operation, deletes the tuples that reference the deleted tuple; for an update, updates the tuples that reference the updated tuple.
- set null: assigns null to referencing attributes.
- set default: assigns default values to referencing attributes.\(^1\)
- restrict: the operation produces an error (default).
- no action: same as restrict.\(^2\)

Note that cascading may set off a “chain reaction” where table $T_1$ references table $T_2$, $T_2$ references $T_3$, and so on.

\(^1\)Default values for an attribute may be specified in the create table statement.

\(^2\)At least that’s what the PostgreSQL reference manual says. C. J. Date, however, writes that no action causes the update or delete to be “performed exactly as requested,” as though there were no constraint (An Introduction to Database Systems, 7th ed., Addison-Wesley, 2000, p. 265). But what’s the use of that? We might as well drop the constraint.
Example 67 See see Listing 4.4.

Listing 4.4: integrity-3.sql, source code.

```sql
create table club (  
    club_id varchar(20),  
    club_division smallint,  
    club_name varchar(40) unique,  
    primary key(club_id, club_division)
);

create table officer (  
    name varchar(20) primary key,  
    age smallint not null,  
    office varchar(30) not null,  
    club_id varchar(20),  
    club_div smallint,  
    check(age >= 2 and age < 140),  
    foreign key(club_id, club_div) references club  
        on delete cascade  
        on update cascade
);

insert into club(club_id, club_division, club_name)  
    values( 'C21', 6, 'Computer Club');

insert into club(club_id, club_division, club_name)  
    values( 'C22', 3, 'Psychology Club');

insert into officer(name, age, office, club_id, club_div)  
    values( 'John Smith', 21, 'President', 'C21', 6);  

insert into officer(name, age, office, club_id, club_div)  
    values( 'Sandy Jones', 12, 'Vice President', 'C21', 6);  

insert into officer(name, age, office, club_id, club_div)  
    values( 'Sigmund Freud', 21, 'President', 'C22', 3);  

insert into officer(name, age, office, club_id, club_div)
```
values ('B. F. Skinner', 12, 'Vice President', 'C22', 3);

select * from club;

update club
    set club_id = 'C23'
    where club_id = 'C21';

select * from officer;

delete from club
where club_id = 'C23';

select * from officer;

drop table officer;

drop table club;

The create domain Statement.

Example 68

create domain salary numeric(10, 2)
    default 1.00
    constraint salary_is_positive
        check (value > 0);

create domain wall_color varchar(10)
    default 'white'
    constraint valid_wall_color
        check (value in ('green', 'white', 'red', 'blue'));

PostgreSQL now implements the create domain statement. PostgreSQL also has a create type statement, but it is different from the SQL standard create type statement, requiring the user to define input and output procedures.

Column constraints in create table can achieve the same results as create domain, but are more verbose because we must write the constraint for every attribute that it applies to.

Standard SQL Features Not Implemented in PostgreSQL

The create assertion Statement. An assertion is a statement or query that the database guarantees to be true. It is checked when created. Subsequently,
it is checked to ensure that modifications of the database do not violate it. (If
they do, they’re not allowed.)

Syntax:

```
create assertion name check query
```

Example 69

```sql
create assertion low_salary
    check (exists (select * from employee
        where salary < 10000));
```

The assertion states that there is at least one employee with a salary below
10,000.

**Generality of Assertions.** Assertions are the most general kind of con-
straints. An assertion can examine multiple tuples, even multiple relations. *Any*
kind of constraint can be rewritten as an assertion (e.g., domain constraints, re-
ferential integrity constraints, primary key constraints). That does not mean it
is a good idea to do so. Complex assertions may involve a large computational
overhead. Consequently, they should be used with care.

Not implemented in PostgreSQL.

## 4.4 SQL Programming Interfaces

Although SQL is a powerful query language, it does not provide everything we
might want in a programming language:

- Functions requiring iteration or recursion.
- Fancy output formatting for reports.
- User-friendly input (text or graphical), i.e., without using the `insert` and
  `update` statements.

Consequently, for many applications, we will need to combine SQL with a
full-featured programming language, called the *host language*. Typical host
languages include C, C++, Java, COBOL, and FORTRAN. There are various
ways of doing this, with different degrees of flexibility:

- Embedded SQL
- Dynamic SQL
- Function or method call interfaces
Learning Objectives

- Be aware that interfaces to SQL from other languages exist.
- Know the main varieties (names, characteristics).
- No detailed knowledge of usage. We do not have time to cover the details in this class. One of the methods (JDBC) is covered in INFO-I 320 Distributed Computing.

Embedded SQL

SQL statements are mingled with host language statements, embedded in special syntax. E.g., **EXEC SQL** may signal the beginning of an SQL statement. If a query retrieves a set of tuples from the database, these tuples can be made available sequentially to the host program using a **cursor**, which is similar to an **iterator**. I.e., we can use the cursor to get the first tuple of the result, and the next tuple, and the next tuple, and so on, until there are no more tuples left. Embedded SQL is limited by the fact that the embedded statements must be translated by a preprocessor, since the embedding extends the syntax of the host language. Also, the SQL statements must be completely specified before the program is compiled.

Dynamic SQL

Dynamic SQL provides greater flexibility, allowing queries to be constructed at runtime. The query can use current values of the host program’s variables. Special syntax is used to prepare and execute the SQL queries. The **prepare** statement formulates an SQL query, and the **execute** statement sends it to the database. Dynamic SQL uses cursors, like embedded SQL, to iterate through a set of tuples returned by a query. Since these additional statements extend the syntax of the host language, dynamic SQL, like embedded SQL, requires a program to be preprocessed before it is compiled.

Function or Method Call Interfaces

More modern developments dispense with preprocessing altogether. Instead, the host language uses a library of functions or classes to interface with SQL, just as it might interface to a library of scientific functions or to one of the standard libraries in C++. For example, the application makes a function call to connect to the database, formulates an SQL query as a string, and makes another function call to execute the query. Like embedded and dynamic SQL programs, it uses cursors to iterate through the results of the query. Eventually, the program makes a call to disconnect.

Standards based on the idea of function-call or method-call interface to databases include **ODBC** (Open Database Connectivity), and **JDBC** (like ODBC,
for Java). ODBC originated as Microsoft’s implementation of SQL/CLI (Call-Level Interface), but later extended it. For further information, see:

- *JDBC Tutorial* \(^3\)
- INFO-I 320 Distributed Computing.

\(^3\)URL: http://java.sun.com/docs/books/tutorial/jdbc/index.html
Chapter 5

Other Database Languages

Reference: Silberschatz, Korth, and Sudarshan, Chapter 5.

5.1 Overview

The relational calculi are formal query languages based on logic. After a review of basic logic, we take quick looks at the tuple relational calculus (TRC) and the domain relational calculus (DRC). Then we shall look at two less formal languages, Query By Example (QBE) and Datalog.

5.2 Logic

Propositional Calculus

Propositional variables: $P, Q, \text{etc.}$

Operators:

$\land$ (conjunction, “and”)

$\lor$ (disjunction, “or”)

$\neg$ (negation, “not”)

$\Rightarrow$ (material implication, “if-then”)

Standard propositional calculus is truth-functional, i.e., the truth value of any logical formula is determined by the truth values of its components.

$A \Rightarrow B$ may be defined as $\neg A \lor B$; its truth table is:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \Rightarrow B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Material Implication

In $A \Rightarrow B$, $A$ is called the antecedent, and $B$ is called the consequent.

The behavior of $\Rightarrow$ is somewhat unintuitive in the case where the antecedent is false. For example, it is natural to consider the statement “If England never had a queen named Elizabeth, then air is heavier than water" as being neither true nor false, since the names of English queens are irrelevant to the weights of water and air; and “If Lobachevsky had not been born, there would be no geometry but Euclidean” as false, since both Gauss and Riemann discovered (or invented) non-Euclidean geometries independently. However, the following expressions in propositional calculus are both true:

- England never had a queen named Elizabeth $\Rightarrow$ air is heavier than water.
- Lobachevsky was not born $\Rightarrow$ there is no geometry but Euclidean.

Material implication is as close as standard propositional logic gets to our natural concept of “if.” Using it has the benefit that logical expressions are truth-functional. Other formal systems of logic, such as relevance logic, have been developed with the goal of modeling more closely the common-sense concept of “if—then”.

Predicate Calculus

Predicate calculus extends propositional calculus by giving propositions a “subject-predicate” structure, adding individual variables representing persons or things (entities), and adding quantifiers expressing the concepts of “all” and “some.”

Simple Predicate Calculus Expressions

Individual constants: $a, b$, etc.; and variables: $x, y$, etc.

Predicate symbols: $P, Q, R$, etc. (including propositional variables as zero-ary predicates).

Atomic formulas: $P(x), Q(a), R(b, y)$, etc.

Compound formulas using $\land, \lor, \neg, \Rightarrow$.

Example 70 Predicate Calculus Expressions

Formulate each statement in the predicate calculus:

1. “McGregor is a farmer.”
   
   farmer(mcgregor)

2. “Washington is the capital of the U.S.A.”

   capital(washington, usa)

3. “$x$ is the capital of Ohio.”

   capital($x$, ohio)
4. “Jones resides in Ohio or Indiana.”
   \[\text{resides(jones, ohio)} \lor \text{resides(jones, indiana)}\]

5. “If Brown is hungry, he works.”
   \[\text{hungry(brown)} \Rightarrow \text{works(brown)}\]

   \[\neg\text{hungry(brown)}\]

**Quantification**

- **Existential.** “For some x P” ("There exists x such that P"):
  \[(\exists x)P\]

- **Universal.** “For all x P” (“For every x P”):
  \[(\forall x)P\]

(Where P is any formula, e.g., \(P = F(x)\)).

**Example 7.1 Quantified Expressions**

1. “There is a dog.” (or “Something is a dog,” or “There exists a dog.”)
   \[(\exists x)\text{Dog}(x)\]

2. “Something is red.” (or “There exists a red thing.”)
   \[(\exists x)\text{Red}(x)\]

3. “There is a dog that is red.” (or “Some dog is red,” or “Some dogs are red.”)
   \[(\exists x)(\text{Dog}(x) \land \text{Red}(x))\]

4. “Everything is red.”
   \[(\forall x)\text{Red}(x)\]

5. “Everything is a red dog.”
   \[(\forall x)(\text{Dog}(x) \land \text{Red}(x))\]

6. “Every dog is red.” (or “All dogs are red.”)
   \[(\forall x)(\text{Dog}(x) \Rightarrow \text{Red}(x))\]

7. “Every red thing is a dog.” (or “All red things are dogs.”)
   \[(\forall x)(\text{Red}(x) \Rightarrow \text{Dog}(x))\]
Logic and the Relational Calculi

Both Tuple Relational Calculus and Domain Relational Calculus are applications of predicate calculus to relations (i.e., sets of tuples). The main difference between them is what they see as individuals, i.e., as possible values of individual variables. Suppose we have a tuple \((\text{Joe}, 33) \in A\).

- In Tuple Relational Calculus, variables are *tuple variables*; i.e., individual variables can have tuples as values, e.g.,
  \[ t = (\text{Joe}, 33) \]

- In Domain Relational Calculus, variables are *domain variables*, i.e., their values can be taken from the domains of the attributes. So we would need two variables for a 2-tuple:
  \[ n = \text{Joe}, a = 33 \]

  Equivalently,
  \[ \langle n, a \rangle = (\text{Joe}, 33) \]

5.3 The Tuple Relational Calculus

In the tuple relational calculus, queries have the form \( \{ t \mid P(t) \} \), meaning the set of all tuples \( t \) such that \( P(t) \) is true. The notation \( t[a] \) refers to the attribute \( a \) of tuple \( t \). Individual variables representing tuples are called *tuple variables*. Note: My quantifier notation differs from SKS, for consistency with standard usage in predicate calculus.

Example 72 Queries in Tuple Relational Calculus

The following queries are based on the same example data (the tables A, B, Product, Distributor, and Product-Distributor) as the examples in Chapter 3A.

1. What are the products named “Motorboat”?
   \[ \{ t \mid t \in \text{product} \land t[\text{pname}] = \text{“Motorboat”} \} \]
   (Equivalent to a select statement in relational algebra.)

2. What are the product IDs of all the products?
   \[ \{ t \mid (\exists s)(s \in \text{product} \land t[\text{pid}] = s[\text{pid}]) \} \]
   (Equivalent to a project statement.) The tuples \( t \) are presumed to have only the attributes mentioned as attributes of \( t \), i.e., only the \( \text{pid} \) attribute.
3. List everybody in A and everybody in B.

\[ \{ p \mid p \in A \lor p \in B \} \]

(Equivalent to a relational algebra query using union.)

4. Find the names of distributors who sell rowboats.

\[ \{ n \mid (\exists d)(d \in \text{distributor} \land \\
\text{n[dbname]} = d[dbname] \land \\
(\exists p)(p \in \text{product} \land \\
p[\text{pname}] = \text{{"Rowboat"}} \land \\
(\exists s)(s \in \text{product-distributor} \land \\
p[\text{pid}] = s[\text{pid}] \land \\
d[\text{did}] = s[\text{did}]))) \} \]

(Equivalent to a relational algebra query using project, select, and natural join.)

5. Find the distributors who distribute every model of motorboat.

\[ \{ d \mid d \in \text{distributor} \land \\
(\forall m)(m \in \text{product} \land m[\text{pname}] = \text{{"Motorboat"}} \Rightarrow \\
(\exists s)(s \in \text{product-distributor} \land \\
m[\text{pid}] = s[\text{pid}] \land \\
d[\text{did}] = s[\text{did}] \land \\
s[\text{quantity}] > 0)) \} \]

(Equivalent to a relational algebra query using select and division.)

### 5.4 The Domain Relational Calculus

The domain relational calculus is just like the tuple relational calculus, except that the individual variables represent domain values (i.e., values of attributes) instead of tuples. They are called domain variables. An n-tuple is represented as \( \langle d_1, d_2, \ldots, d_n \rangle \).

**Example 73 Queries in Domain Relational Calculus**

These examples of queries in the DRC answer the same questions as the examples for TRC.

1. What are the products named “Motorboat”?

\[ \{ \langle p, n, l \rangle \mid \langle p, n, l \rangle \in \text{product} \land n = \text{{"Motorboat"}} \} \]

Note that \( p \) is the pid, \( n \) the pname, and \( l \) the length of a product tuple.
2. What are the product IDs of all the products?

\{ \langle p \rangle \mid (\exists n)(\exists l)\langle p, n, l \rangle \in \text{product} \}

The equivalent of projection is achieved by listing only the domain variables for the desired attributes inside the ⟨⟩’s for the query result, making this query considerably simpler in DRC than in TRC.

3. List everybody in A and everybody in B.

\{ \langle n, a \rangle \mid \langle n, a \rangle \in A \lor \langle n, a \rangle \in B \}

This one is a little more complex, though.

4. Find the names of distributors who sell rowboats.

\{ \langle n \rangle \mid (\exists d)(\exists c)(\langle d, n, c \rangle \in \text{distributor} \land (\exists p)(\exists m)(\exists l)(\langle p, m, l \rangle \in \text{product} \land m = \text{“Rowboat”} \land (\exists q)(\langle p, d, q \rangle \in \text{product-distributor}))) \}

5. Find the distributors who distribute every model of motorboat.

\{ \langle d, n, c \rangle \mid \langle d, n, c \rangle \in \text{distributor} \land (\forall p)(\forall m)(\forall l)(\langle p, m, l \rangle \in \text{product} \land m = \text{“Motorboat”} \Rightarrow (\exists q)(\langle p, d, q \rangle \in \text{product-distributor} \land q > 0)) \}

5.5 Comparisons

The unrestricted tuple relational calculus suffers from the problem that some expressions are uncomputable, because they express infinite relations. This can happen using “not”, i.e., if there are a finite number of A’s, there may be infinitely many non-A’s. When restricted to “safe” expressions, which are guaranteed to have finite values, the TRC is equivalent in expressive power to the basic relational algebra (with the six fundamental operations union, difference, Cartesian product, select, project, and rename). I.e., any query that can be expressed in the basic relational algebra can be expressed in the TRC, and vice-versa.

The domain relational calculus has the same expressive power as the TRC, and thus the same expressive power as the basic relational algebra.
Recall that in relational algebra, the operations intersection, natural join, and division can be defined in terms of the six fundamental operations. Their presence is a notational convenience, but it does not increase the expressive power of the language. The extended operations of aggregation and outer join, however, do increase the expressive power of relational algebra. Tuple and domain relational calculi are not equivalent in expressive power to the extended relational algebra, although it is possible to extend the relational calculi with aggregation, at least.

Unlike the relational algebra, both TRC and DRC are non-procedural, i.e., declarative. A query expression in TRC/DRC merely describes the set of tuples desired (the result of the query); it does not tell how to compute it. It is like the definition of $\sqrt{n}$ as “the number $k$ such that $k^2 = n$.” It tells us how to recognize the answer. It does not tell us how to find it.

5.6 Query By Example (QBE)

QBE, developed by IBM in the 1970’s, is now probably best known through the graphical variant used in Microsoft Access (although only IBM uses the name QBE, since it is their trademark). Queries are written in a two-dimensional table format, and “look like” (i.e., are examples of) the data they are supposed to retrieve. Well, sort of.

QBE provides an intuitive interface for simple queries, especially for unsophisticated users. SQL is more suitable for complex queries, e.g., those involving subselects.

Some authors state that QBE is relationally complete; that is, it can express any query that can be expressed in relational algebra. Others say it cannot express some of the relational algebra operations, e.g., union. What is the truth?

In its use of variables for attribute values, it resembles the domain relational calculus.

We do not present the complete QBE language here, but only a few examples to suggest its flavor.

Sample QBE Queries

Example 74 “What are all the products (pid’s, pnames, and lengths)?”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: “P.” means print. The first column is the relation; the other columns are its attributes. Putting “P.” in the first column and no conditions in the other columns means print all of the attributes for each tuple in the relation.

Example 75 “What are the pids of all products?”
CHAPTER 5. OTHER DATABASE LANGUAGES

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.(_x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: “\(_x\)” is a (domain) variable. “P.\(_x\)” in pid column means print the pid value.

**Example 76**  “What are the pid’s of products named ‘Motorboat’?”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.(_x)</td>
<td></td>
<td>Motorboat</td>
<td></td>
</tr>
</tbody>
</table>

**Example 77**  “Find the names of distributors who sell rowboats.”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(_p)</td>
<td></td>
<td>Rowboat</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>product-distributor</th>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(_p)</td>
<td>(_d)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>distributor</th>
<th>did</th>
<th>dname</th>
<th>dcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(_d)</td>
<td>(_p)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: This query joins three tables together. Use of the variable \(_p\) twice requires pid’s to match; similarly, use of \(_d\) twice selects matching did’s.

**Example 78**  “Delete motorboats from the product and product-distributor tables.”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D).</td>
<td>(_p)</td>
<td>Motorboat</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>product-distributor</th>
<th>pid</th>
<th>did</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D).</td>
<td>(_p)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: “\(D\).” means delete.

**Example 79**  “Insert (P7, Rowboat, 15) into product.”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I).</td>
<td>P7</td>
<td>Rowboat</td>
<td>15</td>
</tr>
</tbody>
</table>

Notes: “\(I\).” means insert.

**Example 80**  “Add 1 foot to the length of P6.”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P6)</td>
<td></td>
<td></td>
<td>(U.(_x+1))</td>
</tr>
</tbody>
</table>

Notes: “\(U\).” means update.
Example 81  “Find all boats with length greater than 25.”

<table>
<thead>
<tr>
<th>product</th>
<th>pid</th>
<th>pname</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: The “conditions” box provides a place to write additional select conditions. It is not an additional relation.

5.7 Datalog

Datalog is a database query language related to Prolog. Prolog ("PROgramming in LOGic") was developed as an artificial intelligence programming language based on predicate calculus. As a logic-based language, Prolog is more declarative than most programming languages; still it contains some procedural aspects, e.g., the order in which statements are written affects the order of evaluation. Datalog is a purely declarative subset of Prolog.

Syntax

Names may include capital and lower-case letters, the underscore character ("_"), and digits. Names of variables begin with a Capital letter or the underscore. Variables names beginning with the underscore, including the underscore by itself, are considered “anonymous” or “don’t care” variables for the sake of pattern matching. Names of constants and predicates begin with lower-case letters.

Examples X, Dog, DogHouse, Dog_House are variables. _ X, _Dog are “anonymous” variables. x, dog, doghouse, dog_house, dogHouse are constants or predicates.

More Syntax

Logical operators: “And” is written as a comma (","). “If” is ":-". The Datalog “if” is reverse implication, so that "A :- B" means "if B then A"; we may also read it as "A if B". “Not” in Datalog is “not” or “\+”. The “\+” is more modern; it is a keystroke representation of the logicians’ notation \not\vdash, which means “not provable”.

Identity: X = Y means that X and Y are the same thing. X \= Y means they are not the same (X \not= Y).

Period: Every Datalog statement ends with a period (".").

Example 82 Datalog Statements

- fish (tonytuna).
• swims(X) :- fish(X).
• dog(X) :- wagtail(X), barks(X).
• private_info(X) :- info(X), + public(X).
• roommate(P1, P2) :- room(P1, R), room(P2, R), P1 \neq P2.

Facts, Rules, and Queries

A Datalog program typically consists of a combination of rules and facts. Rules are conditional (“if”) formulas. Facts are formulas without “if”.

Example 83 A Deductive Family Database

The family program (family.P, see Listing 5.1) begins with a set of facts such as “joseph is a child of george and mary”. In addition, it contains rules which define the predicates (relations) father, mother, parent, siblings, cousins, and ancestor. Note that some of these rules are recursive.

Listing 5.1: family.P, source code.

% File: family.P
% Copyright (C) 2004 Gregory D. Weber
% Family relationships.

5 % Facts
%
% child(Child, Father, Mother)
%
10 child(joseph, george, mary).
child(mark, george, mary).

child(rebecca, thomas, eleanor).
15 child(robert, joseph, susan).
child(ann, joseph, susan).
child(elizabeth, joseph, susan).

child(herbert, mark, helen).
20 child(hanna, mark, helen).

child(william, richard, rebecca).
child(margaret, richard, rebecca).

25 % Rules

father(F, C) :- child(C, F, _).
mother(M, C) :- child(C, _, M).
parent(P, C) :- father(P, C).
parent(P, C) :- mother(P, C).
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(P, B), ancestor(A, P).
siblings(A, B) :-
    child(A, F, M),
    child(B, F, M),
    A \(\neq\) B.
cousins(A, B) :-
    parent(PA, A),
    parent(PB, B),
    siblings(PA, PB).

To run the program, we start Datalog and “consult” (i.e., load) the file by entering either “consult(family).” or “[family]”. For a sample session, see see Listing 5.2. We may then type in queries. Each query must end with a period. Datalog will respond to a query by displaying the values of variables (if any) that satisfy the query; if we wish to see more solutions, we may type a semicolon “;” and ENTER, otherwise just type ENTER. When there are no more solutions, Datalog responds “no”. For simple queries with no variables to bind, Datalog just responds “yes” or “no”. The example shows a series of queries, beginning with simple “yes or no” questions that check the facts of the database, through finding variables to satisfy queries, through recursive queries.

Listing 5.2: family.sio, source code.

[gdweber@chopin src]$ xsb
[xsb_configuration loaded]
[sysoinitrc loaded]
[packaging loaded]

XSB Version 2.4 (Bavaria) of July 13, 2001
[i686-pc-linux-gnu; mode: optimal; engine: chat; gc: copy; scheduling: local]

| ?- consult(family).
[Compiling ./family]
[family compiled, cpu time used: 0.0410 seconds]
[family loaded]

yes
104 | CHAPTER 5. OTHER DATABASE LANGUAGES

| ?- child(joseph, george, mary).

yes
19 | ?- child(joseph, herbert, elizabeth).

no
| ?- child(joseph, F, M).

24 F = george
M = mary

yes
| ?- father(F, joseph).

29 F = george

yes
| ?- father(F, george).

34 no
| ?- father(george, Child).

Child = joseph;
39 Child = mark;

no
| ?- parent(P, elizabeth).

44 P = joseph;
P = susan;
49 no
| ?- parent(mark, C).

C = herbert;
54 C = hanna;

no
| ?- siblings(robert, Sib).

59 Sib = ann;
Sib = elizabeth;
Expressive Power of Datalog

Non-recursive Datalog has the same expressive power as relational algebra. Recursion allows queries to be expressed in Datalog that could not be expressed in relational algebra, or in SQL 92. (However, SQL:1999 introduces a limited form of recursive queries.)
Chapter 6
The Entity-Relationship Model

Reference: Silberschatz, Korth, and Sudarshan, Chapter 6

6.1 Introduction

The Entity-Relationship (E-R) model is a conceptual tool frequently used in the design of databases. E-R diagrams provide a graphical depiction of the data in the database and their inter-relations.

6.2 Basic Concepts

Entities

Entities are things that exist, be they animate or inanimate, personal or impersonal, abstract or concrete. Examples of entities: an automobile, an employee, a pet hamster, a loan, an insurance policy. An entity set is a set (collection) of entities of the same type, e.g., a set of automobiles, a set of employees.

Attributes

Entities are described (characterized) by their attributes, i.e., properties. Examples: attributes of employee might include name, address, social security number, wage, and date of hire. Attributes of automobile might include make, model, year, vehicle identification number (VIN), color, age, and purchase price. Some of the attributes may be sufficient to uniquely identify an entity, e.g., a social security number identifies an employee, a VIN identifies an automobile. We refer to these uniquely identifying attributes as keys.

1Accent the first syllable: “at-trib-u tes” (“at-tib-u tes” is a verb).
Each attribute has a *domain*, a set of permissible values. Examples: the purchase price of a car must be a positive number; a social security number is an unsigned 9-digit number; an employee’s name is a string; vehicle make must be one of \{“Ford”, “Chevrolet”\} (assuming, for simplicity, that there are only these two car manufacturers).

**E-R Diagram of an Entity and Its Attribute**

In E-R diagrams, an entity set is shown as a rectangular box, connected by lines to its attributes, which are enclosed in ellipses, as in Figure 6.1.

**Example 84** Figure 6.2 shows the Automobile entity set, with attributes make, model, year, VIN, and color.

**Kinds of Attributes**

Attributes can be classified as:

1. *Simple* or *composite*.
2. *Single-* or *multi-valued*.
3. *Derived* or non-derived.

**Example 85** Figure 6.3 figure shows color as a multi-valued attribute, and age as a derived attribute, of Automobile.
Null Attributes

Sometimes a “null value” is used for an attribute. Null is not really a value; it represents the absence of a value. Specifically, null may mean:

1. “Does not apply” (e.g., the pregnant attribute, true or false, for a male patient)

2. “Unknown” or “missing” (i.e., not recorded in the database), either:
   (a) Known to have a value, but the value is unknown (e.g., we know a car has a VIN but we don’t know what it is); or
   (b) we don’t even know if it has a value (e.g., date of last X-ray, when we don’t know if the patient has ever had an X-ray).

The use of null is apt to cause confusion—partly because of its ambiguity, partly because of the complicated ways it interacts with other values in queries. It is often desirable (and possible) to reorganize a database, so that null values are not used.

Relationships

A relationship is a connection or association between entities. Examples: John is married to Sue. Fred is the owner of Fred’s Corvette.

A relationship set is a set of relationships of the same type. Examples: the husband-wife relationship set, the owner-automobile relationship set.

Definition

Formally, a relationship $R$ between entity sets $E_1, E_2, \ldots, E_n$ is a set of $n$-tuples $(e_1, e_2, \ldots, e_n)$ where $e_i \in E_i (i = 1, 2, \ldots, n)$. That is, $R$ is a subset of the Cartesian product $E_1 \times E_2 \times \ldots \times E_n$. We say that these entity sets participate in the relationship. Note: $e_1, \ldots, e_n$ are entities, not attributes.
Arity of Relationships

An \( n \)-ary relationship is one that has \( n \) participating entity sets. If \( n = 2 \), we say it is a binary relationship. (To be more exact, an \( n \)-ary relationship is one consisting of \( n \)-tuples. Some of the entity sets may be used more than once in different roles. For example, a borrower-lender relationship between the entity set Person and the entity set Person is binary: although there is only one entity set, it is used in two roles.)

Descriptive Attributes

- A relationship may also have descriptive attributes.

Examples

Relationship Sets in E-R Diagrams

In E-R diagrams, a relationship set is shown in a diamond-shaped box, connected by lines to the entity sets that participate in it, as in Figure 6.4.

*Example 86* Figure 6.5 shows the relationship set Insured, between the entity sets Owner and Automobile.

Modeling Issues

- Can spouse (an entity) be an attribute of a person entity?
- Can a person’s name be an entity?
6.3. CONSTRAINTS

Rules of Thumb

1. Something can be considered an entity if it has attributes.

2. If it has no attributes of its own, it may be an attribute of an entity (or of a relationship), but it cannot be an entity itself.

3. An attribute’s value must not be an entity.

4. Anything involving two or more entities must be a relationship (one of the entities cannot be an attribute of the other).

Discussion

What is (probably) the correct way of modeling:

- A person’s owning a car?
- A person’s having a social security number?
- A car’s being made by a particular manufacturer? (What other information, if any, do we have about the maker?)

6.3 Constraints

Constraints are rules that restrict the possible contents of a database. The database designer uses constraints to help preserve the integrity of data.

There are three important kinds of constraints: data types, mapping cardinalities, and participation constraints.

Data Types

Data types are a form of constraint, although not discussed as such in the textbook. Data types limit the possible values of attributes in an entity or relationship, and the possible entities related in a relationship. An attribute value must belong to the domain of the attribute. Entities in a relationship must belong to the entity sets that participate in the relationship set. For example, data type constraints would prevent the storing of ABC-DE-FGHI as a social security number; and would prevent the storing of Sam-Fido as a husband-wife pair, where Sam is a man and Fido is a dog.

Mapping Cardinalities

Mapping cardinalities represent the allowable numbers of entities related to a particular entity in a relationship. There are four possibilities. Consider a binary relationship set \( R \subseteq A \times B \). \( R \) may be:

1. One-to-one, if each member of \( A \) is related to at most one member of \( B \), and vice-versa. Example: marriage in a monogamous society.
2. **One-to-many**, if each member of $A$ may be related to any number of members of $B$, but each member of $B$ may be related to at most one member of $A$. Examples: marriage in a polygynous society, where a man may have many wives but a woman may have only one husband; car ownership, where each person may own any number of cars, but each car has just one owner.

3. **Many-to-one**—the reverse of one-to-many. Examples: marriage in a polyandrous society, where a woman may have many husbands but a man may have only one wife; political party affiliation, if each voter can be registered as a member of only one party, but a party may have many members.

4. **Many-to-many**, if each member of $A$ may be related to any number of members of $B$, and vice-versa. Examples: marriage in a polygamous society (both men and women may have multiple spouses); student-course registration, where a student may register for multiple courses and a course may have multiple students.

Note: the term “one” in this context really means “zero or one”; the term “many” means “zero or more,” i.e., any non-negative integer.

**Mapping Cardinality Constraints in E-R Diagrams**

E-R diagrams show mapping cardinality constraints by using arrows. An arrow on the line from a relationship set to an entity set means that the entity set’s mapping cardinality is “one” (as in “one-to-many” or “one-to-one”). The absence of an arrow signifies that its mapping cardinality is “many”. Figure 6.6 shows the four possible mapping cardinalities.
6.4 KEYS

Consider, again, a binary relationship \( R \subseteq A \times B \). We say the participation of \( A \) in this relationship is total, if, for every \( a \in A \) there is at least one tuple \( (a, b) \in R \), i.e., every member of \( A \) is involved in the relationship. Otherwise, we say \( A \)’s participation in \( R \) is partial.

**Example**: if every car must be owned by somebody, but not everyone is required to own a car, then the ownership relationship between persons and cars has total participation by the set of cars and partial participation by the set of persons.

**Another Example**

In an election:
- Eligible voters vote for candidates.
- Total eligible voter participation in the vote-for relationship means . . .
- Total candidate participation means . . .

**Participation Constraints in E-R Diagrams**

E-R diagrams show total participation constraints using double lines. In Figure 6.7, entity set \( A \) participates totally in relationship \( R \), and entity set \( B \) participates partially.

6.4 Keys

The notions of superkey, candidate key, and primary key, presented earlier in the context of the relational model, carry over in the obvious way to entity sets in the E-R model: keys are sets of attributes that uniquely identify each entity in an entity set, or each relationship in a relationship set.

**Keys for Relationship Sets**

Relationship sets, as well as entity sets, need keys. Suppose we have relationship set \( R \subseteq E_1 \times E_2 \times \ldots \times E_n \). If \( \text{PK}(E_i) \) is the primary key of \( E_i(i = 1, 2, \ldots, n) \), then

\[
\text{PK}(E_1) \cup \text{PK}(E_2) \cup \ldots \cup \text{PK}(E_n)
\]
Figure 6.8: The Automobile relationship set with a primary key.

is a superkey of $R$, provided that all of the $E_i$ are distinct sets and have distinct primary keys.\(^2\)

**Example 87** Consider the Enrolled relationship set between entity sets Student and Section. If $\text{PK(Student)} = \{\text{sid}\}$, and $\text{PK(Section)} = \{\text{section.number}\}$, then $\{\text{sid, section.number}\}$ is a superkey for the relationship.

How much of this superkey is actually needed depends on the mapping cardinalities.

**Primary Keys in E-R Diagrams**

The primary key is *underlined* in E-R diagrams. If the primary key for an entity set consists of more than one attribute, all of the attributes in the primary key are underlined; and similarly for primary keys of relationship sets.

**Example 88** Figure 6.8 shows VIN as the primary key of Automobile.

### 6.5 Exercises

For discussion in class:


2. SKS page 256: problems 6.2–6.4.

### 6.6 Strong and Weak Entity Sets

A *weak entity set* is an entity set that has insufficient attributes to form a key (superkey, candidate key, or primary key)\(^2\). If an entity set is not weak, then it is *strong*.

\(^2\)If some of the $E_i$ are the same or have the same primary key, we can rename the key attributes to make this work, using the role of the entity set in the relationship, e.g., husband.ssn, wife.ssn.
Example 89 An apartment owner’s database has an entity set Apartment, with attributes unit (number), (number of) bedrooms, and (monthly) rent; and an entity set Payment, with attributes date and amount. Each apartment’s renter makes only one payment per month, but there are 100 apartments in the building, and some of them pay on the same date. Unit number is the primary key for Apartment. Payments cannot be uniquely identified by date, by amount, or by date and amount, so Payments is a weak entity set. Each payment is for a particular apartment, so there is an Apartment-Payment relationship.

How to Identify Members of a Weak Entity Set

How, then, do we identify members of a weak entity set? The weak entity set must be related by an identifying relationship to a strong entity set, which is called the owner or identifying entity set. We say that the identifying entity set owns the weak entity set, and that the weak entity set is existence dependent on its identifying entity set.

In our example, Apartment is the identifying entity set, and the identifying relationship is Apartment-Payment. If we know which apartment the payment is for, and the date of the payment, then we have uniquely identified the payment. Figure 6.9 shows the E-R diagram representing this information.

Weak Entity Sets in E-R Diagrams

In the E-R diagram, the weak entity set and its identifying relationship are shown with double-lined shapes (rectangle and diamond).

Note that the weak entity set must participate totally in the identifying relationship (why?), and that the identifying relationship must be one to many from the identifying entity set to the weak entity set (why?). Both of these requirements are shown in the diagram.

For any one apartment, the different payments are distinguished by their dates. We say that \{date\} is the discriminator for this entity set. In general, a discriminator is a set of attributes that distinguishes between weak entities owned by the same strong entity. The primary key for a weak entity set is the union of its discriminator (or one of its discriminators) with the primary key
of its identifying entity set. In our example, the primary key for the Payment entity set is \{unit, date\}, since \{unit\} is the primary key for Apartment. (Okay, give me the key and let me in!) In the E-R diagram, the key of the identifying entity set is underscored with a solid line, and the discriminator of the weak entity set is underscored with a dashed line.

**Discussion.** (Maybe) how to diagram the course-offering example, (5th ed.) page 227.

### 6.7 Specialization

Users of object-oriented programming languages are familiar with the application of classification trees to programming, i.e., the use of superclasses and subclasses. A subclass *inherits* characteristics (data or functions) from its superclass, and it usually *specializes* the superclass by adding to or overriding some of the characteristics.

The same technique of specialization can be used to develop entity sets in the E-R model. In a merchant ship database, for example, each ship has a name, a length, and a weight. In addition to these, a tanker has a tank capacity (gallons), a passenger ship has a number of cabins, and a submarine has a maximum depth and a number of torpedoes. The tanker, passenger ship, and submarine entity sets are specializations of the ship entity set. In E-R diagrams, the specialization is shown as a triangle labeled “ISA” (since, e.g., a tanker is a ship). See Figure 6.10.

What is inherited by the subclasses of ship? What do they not inherit?

**Design Strategies**

Design by specialization flows from the top down, from general concepts to specific. *Generalization* is the reverse: the design begins with specific entity sets (such as computer and printer); and as the designers recognize that they have common features, they add a more general entity set (such as product). Either way works!

**Discussion.** (5th ed.) Page 260, number 6.24, motor vehicles

### 6.8 Reducing an E-R Schema to Tables

We know that E-R is one kind of data model. We also know that the dominant model in database systems today is the relational model. If we ultimately want to develop a relational database, aren’t the E-R diagrams a waste of time?

No, because it is straightforward to derive a relational model from an E-R model. We do this by reducing the E-R model to tables. (The relations in the relational model are, informally speaking, tables.) We show how to do this for each of the major components of an E-R model.

---

\(^3\text{C++: “base classes” and “derived classes”}\)
Figure 6.10: Specialization.
Strong Entity Sets

Example 90 The Automobile entity set from an earlier example (see Figure 6.2) has the attributes make, model, year, VIN, color; for the time being, we assume all of the attributes are single-valued.

The table for the entity set has a column for each of the attributes, e.g.:

<table>
<thead>
<tr>
<th>make</th>
<th>model</th>
<th>year</th>
<th>VIN</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford</td>
<td>Taurus</td>
<td>1995</td>
<td>123</td>
<td>black</td>
</tr>
<tr>
<td>Toyota</td>
<td>Camry</td>
<td>2000</td>
<td>125</td>
<td>blue</td>
</tr>
<tr>
<td>Volkswagen</td>
<td>Beetle</td>
<td>1973</td>
<td>041</td>
<td>green</td>
</tr>
<tr>
<td>Buick</td>
<td>Century</td>
<td>2002</td>
<td>139</td>
<td>white</td>
</tr>
</tbody>
</table>

The schema for the table is \( automobile = (\text{make}, \text{model}, \text{year}, \text{VIN}, \text{color}) \); underlining indicates primary key.

In general, a strong entity set with single-valued attributes \( A_1, \ldots, A_n \) becomes a table with columns \( A_1, \ldots, A_n \).

Relationship Sets

Example 91 See SKS, Figure 6.31, p. 260.

Written-by Table

<table>
<thead>
<tr>
<th>author-name</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Silberschatz, A.</td>
<td>0-07-228363-7</td>
</tr>
<tr>
<td>Korth, H. F.</td>
<td>0-07-228363-7</td>
</tr>
<tr>
<td>Sudarshan, S.</td>
<td>0-07-228363-7</td>
</tr>
<tr>
<td>Date, C. J.</td>
<td>0-201-38590-2</td>
</tr>
<tr>
<td>Ramakrishnan</td>
<td>0-07-050775-9</td>
</tr>
</tbody>
</table>

Table schema: \( \text{written-by} = (\text{author-name}, \text{ISBN}) \).

In general, the table for a relationship set has a column for each attribute of the primary keys of the entity sets that participate in it. In addition, if the relationship has any descriptive attributes, there is a column for each of them. (There are exceptions involving weak entity sets, described below.)

Example 92 See the same figure.

Contains Table

<table>
<thead>
<tr>
<th>ISBN</th>
<th>BasketID</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-07-228363-7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0-201-38590-2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0-201-38590-2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0-07-050775-9</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table schema: \( \text{contains} = (\text{ISBN}, \text{BasketID}, \text{number}) \).
Weak Entity Sets

The table for a weak entity set has a column for each attribute of the weak entity set and a column for each primary key attribute of the identifying entity set. Recall that the primary key of a weak entity set is the union of its discriminator with the primary key of the identifying entity set. Therefore, the table contains all the primary key attributes, whether they are attributes of the weak entity set or of the owner set, as well as any non-key attributes of the weak entity set.

Example 93 Recall the apartment-payment example shown in Figure 6.9.

```
<table>
<thead>
<tr>
<th>unit</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-25-02</td>
<td>500.00</td>
</tr>
<tr>
<td>1</td>
<td>2-20-02</td>
<td>500.00</td>
</tr>
<tr>
<td>2</td>
<td>12-30-01</td>
<td>500.00</td>
</tr>
<tr>
<td>2</td>
<td>1-25-02</td>
<td>500.00</td>
</tr>
<tr>
<td>3</td>
<td>2-17-02</td>
<td>550.00</td>
</tr>
</tbody>
</table>
```

Table schema: \(\text{payment} = (\text{unit}, \text{date}, \text{amount})\).

Identifying Relationship of Weak Entity Set

If a weak entity set and its identifying relationship are both converted to tables, the table for the identifying relationship contains only redundant information. So, we do not create a table for the identifying relationship.

Example 94 See the weak entity set Payment in the figure above. What is the identifying relationship? If it were converted to a table, what columns would the table have? Where are these columns already found?

Multi-Valued Attributes

If an entity set contains multi-valued attributes, they are treated differently from single-valued attributes. Instead of putting a multi-valued attribute in a column of the table for the strong entity set (why would this not work?), we create an additional table for each multi-valued attribute. Each such table has column(s) for the primary key of the strong entity set and a column for the multi-valued attribute.

Example 95 We revisit the automobile example (see Figure 6.3), this time treating \textit{color} as a multi-valued attribute.
Automobile-Color Table

<table>
<thead>
<tr>
<th>VIN</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>red</td>
</tr>
<tr>
<td>123</td>
<td>black</td>
</tr>
<tr>
<td>125</td>
<td>blue</td>
</tr>
<tr>
<td>041</td>
<td>green</td>
</tr>
<tr>
<td>139</td>
<td>white</td>
</tr>
<tr>
<td>139</td>
<td>blue</td>
</tr>
</tbody>
</table>

Table schema:  
automobile = (make, model, year, VIN);  
automobile-color = (VIN, color).

Discussion. Reduce some of Figure 6.31, on page 260, to tables.

6.9 Unified Modeling Language (UML)

UML, a graphical language for object-oriented design, provides another way to diagram entities and relationships. (BUS-S 310 Systems Analysis and Design)
Chapter 7

Relational Database Design


7.1 Outline

- Functional Dependencies
- Normal Forms

7.2 Overview

The theory of normal forms is an aid to designing effective databases. Normalizing a database, that is, converting it to an appropriate normal form, reduces redundancy while still allowing all the information in the original form to be retrieved. There is a series of normal forms, 1NF (first), 2NF, 3NF, BCNF (Boyce-Codd), 4NF, and 5NF; each of which increases the degree of normalization. The most important of these are 3NF and BCNF. Most of the normal forms are based on the recognition of functional dependencies (FDs). FDs generalize the concept of superkey.

7.3 Functional Dependency

Definition 6 Let $R$ be a relation, $Z$ the set of attributes of $R$, $A \subseteq Z, B \subseteq Z$. There is a functional dependency (FD)

$$A \rightarrow B$$

if it is necessary that for any tuples $t_1, t_2 \in R$, if $t_1[A] = t_2[A]$, then $t_1[B] = t_2[B]$. Equivalently, if $t_1[B] \neq t_2[B]$, then $t_1[A] \neq t_2[A]$. The notation $A \rightarrow B$ is also read as $A$ (functionally) determines $B$. 

121
Example 96 In the relation product \((\text{pid}, \text{pname}, \text{length})\), \{\text{pid}\} \rightarrow \{\text{pid}, \text{pname}, \text{length}\}\), since \{\text{pid}\} is the primary key.

In general, if \(K\) is any superkey of a relation with attributes \(Z\), then \(K \rightarrow Z\).

The concept of functional dependency generalizes the concept of superkey.

Example 97 Consider the relation employee \((\text{name}, \text{rank}, \text{salary})\). Suppose that \(\text{domain}(\text{rank}) = \{\text{programmer}, \text{senior-programmer}, \text{analyst}\}\) and that, as a matter of company policy, all programmers are paid $20,000 per year, all senior programmers $30,000, and all analysts $40,000. Then the FD \(\text{rank} \rightarrow \text{salary}\) holds, as well as the FD \(\text{salary} \rightarrow \text{rank}\).

Consider some ways in which one or both of these FDs would not hold.

Definition 7 If \(B \subseteq A\), then \(A \rightarrow B\) is called a trivial FD.

Example 98 The following are trivial dependencies:

1. \(\{A\} \rightarrow \{A\}\)
2. \(\{A, B\} \rightarrow \{B\}\)

Trivial FDs are so called because they are logically necessary. Any relation must have the trivial dependencies among its sets of attributes.

Notation Convention 1 If \(A, B, C, D\) are sets of attributes, we may write \(A \cup B \rightarrow C \cup D\) as \(AB \rightarrow CD\).

Properties of Functional Dependencies

Armstrong’s axioms: Let \(X, Y, Z\) be subsets of the set of attributes of a relation. Then:

Axiom 1 (Reflexivity) If \(Y \subseteq X\), then \(X \rightarrow Y\) (trivial FDs).

Axiom 2 (Augmentation) If \(X \rightarrow Y\), then \(XZ \rightarrow YZ\).

Axiom 3 (Transitivity) If \(X \rightarrow Y\) and \(Y \rightarrow Z\), then \(X \rightarrow Z\).

The axioms are complete: all functional dependencies implied by a set of FDs can be derived from them; and sound: no incorrect FD statements can be derived from them.

The following laws can be proved from Armstrong’s axioms, for any sets of attributes \(A, B, C, D\):

Theorem 1 (Self-determination) \(A \rightarrow A\).

Proof: Since \(A \subseteq A\), the Reflexivity Axiom implies that \(A \rightarrow A\).
Theorem 2 (Union) If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.

Proof:
1. Assume $A \rightarrow B \wedge A \rightarrow C$
2. $A \rightarrow B$ (from 1)
3. $AA \rightarrow AB$ (Augmentation, 2)
4. $A \rightarrow AA$ (Reflexivity, since $AA \subseteq A$)
5. $A \rightarrow AB$ (Transitivity, 4, 3)
6. $A \rightarrow C$ (1)
7. $AB \rightarrow BC$ (Augmentation, 6)
8. $A \rightarrow BC$ (Transitivity, 5, 7)
\[\square\]

Theorem 3 (Decomposition) If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$.

(Decomposition is the converse of union.)

Theorem 4 (Pseudo-transitivity) If $A \rightarrow B$ and $BC \rightarrow D$, then $AC \rightarrow D$.

Theorem 5 (Composition) If $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$.

Closures

Definition 8 Let $F$ be a set of FDs. The closure of $F$, written $F^+$, is the set of all FDs implied by $F$.

Example 99 Let $F = \{A \rightarrow B, B \rightarrow C\}$. Then: $F^+$ includes $A \rightarrow C$, by the transitivity axiom. $F^+$ also includes $A \rightarrow B$, since it is included in $F$. $F^+$ does not include $B \rightarrow A$.

Definition 9 Let $F$ be a set of FDs and $Z$ be a set of attributes. The closure of $Z$ under $F$ is the set of attributes that are functionally determined by $Z$ through the FDs in $F$.

A Closure Algorithm

It is useful to be able to compute the closure of a set of attributes under a set of FDs. Among other things, the computation can tell us whether a set of attributes is a superkey for its relation (the closure contains all of the attributes of the relation). The following algorithm performs this computation:

Algorithm closure ($S$, $F$)
result $\leftarrow S$
repeat
for each FD $X \rightarrow Y \in F$ do
if $X \subseteq$ result then
result $\leftarrow$ result $\cup Y$
end if
\endrepeat
end for
until result does not change
return result

Example 100 Let $S = \{A, B, C\}$ and $F = \{A \rightarrow C, A \rightarrow D, AE \rightarrow FG, BC \rightarrow E, AGF \rightarrow H\}$.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{A, B, C}</td>
</tr>
<tr>
<td>1</td>
<td>{A, B, C, D, E}</td>
</tr>
<tr>
<td>2</td>
<td>{A, B, C, D, E, F, G, H}</td>
</tr>
</tbody>
</table>

Another application of the closure algorithm is, given a set $F$ of FDs, determine whether a proposed FD $X \rightarrow Y$ holds, by computing closure($\{X\}, F$). ($X \rightarrow Y$ holds if $Y$ is in the result.)

Covers

Definition 10 Let $C$ and $S$ be sets of FDs. $C$ is a cover of $S$ if every FD implied by $S$ is implied by $C$, i.e., $S^+ \subseteq C^+$. If $S$ is also a cover for $C$, we say that $C$ and $F$ are equivalent.

If $C$ and $S$ are equivalent sets of FDs, then when a database mechanism enforces the FDs of $C$ as constraints, it automatically does so for the FDs of $S$, and vice-versa. We would like to find a minimal set of FDs that is equivalent to a given set, so that the database can do this as efficiently as possible.

Definition 11 A set $F$ of FDs is minimal if, for every FD $X \rightarrow Y \in F$:

1. $Y$ consists of a single attribute;
2. for every $X' \subset X$, $F - \{X \rightarrow Y\} \cup \{X' \rightarrow Y\}$ is not equivalent to $F$; i.e., we cannot replace $X$ by a proper subset of $X$ without losing equivalence to $F$; i.e., no attribute in $X$ is superfluous; and
3. $F - \{X \rightarrow Y\}$ is not equivalent to $F$; i.e., no FD in $F$ is superfluous.

SKS present a different idea of a minimal set of FDs, which they call a canonical cover, in section 7.3.4.1

1 Their idea is this: given a set $F$ of FDs, reduce it to an equivalent set of FDs in which there are no extraneous attributes. An attribute is extraneous in an FD if it can be removed from the FD without affecting the closure of the set.

Examples:

1. Let $F = \{A \rightarrow B, A \rightarrow BC\}$. Since $F' = \{A \rightarrow B, A \rightarrow C\}$ is equivalent to $F$, attribute $B$ is extraneous in $A \rightarrow BC$.
2. Let $F = \{AB \rightarrow C, B \rightarrow C\}$. Since $F' = \{B \rightarrow C\}$ is equivalent, the attribute $A$ is extraneous in $AB \rightarrow C$. 


Example 101 Let $A, B, C, D, E$ be single attributes, and let $F = \{ A \rightarrow BC, C \rightarrow D, BC \rightarrow D, D \rightarrow E, C \rightarrow E \}$. $F$ is not a minimal set of FDs. We can derive an equivalent minimal set of FDs as follows:

1. Replace $A \rightarrow BC$ by $A \rightarrow B, A \rightarrow C$.
2. Remove $BC \rightarrow D$, since it is implied by $C \rightarrow D$.
3. Remove $C \rightarrow E$, since it is implied by $C \rightarrow D$ and $D \rightarrow E$.

7.4 Normal Forms

There are 6 commonly studied normal forms: first (1NF) through fifth (5NF), and Boyce-Codd (BCNF), an “improved” 3NF. The normal forms are hierarchically organized, so that every 5NF relation is 4NF, every 4NF relation is BCNF, every BCNF relation is 3NF, every 3NF relation is 2NF, and every 2NF relation is 1NF.

Normalization to only 1NF or 2NF is not adequate; the first two normal forms are mere stepping stones on the way to higher normal forms, usually 3NF or BCNF. Normal forms above BCNF (4NF, 5NF) are less commonly used.

The goals of normalization (i.e., converting relations to higher normal forms) are to reduce redundancy and certain anomalies associated with redundancy.

First Normal Form (1NF)

Definition 12 A domain is atomic if its elements are treated as indivisible units.

Numbers and strings are usually atomic. Sets and composite attributes, e.g., name = (first, last), are not. The distinction is not intrinsic to the domain but depends on how it is used: if we apply bit-level operations to integers, they are no longer atomic.

Definition 13 A relation schema $R$ is in first normal form (1NF) if all of its attributes have atomic domains. Equivalently, if each tuple has just one value for each attribute.

We have assumed all relations are in 1NF since beginning our study of the relational model in chapter 3, so 1NF is really nothing new.

Example 102 Converting to 1NF Relations having non-atomic domains are easily converted to 1NF.

1. A multi-valued attribute in the E-R model, represented as an attribute with some type of set as its domain:
2. A composite attribute:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Woods, Peter)</td>
<td>19</td>
</tr>
<tr>
<td>(Stuart, Dolores)</td>
<td>21</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
A_1 & A_2 \\
1 & \{2, 3, 4\} \\
5 & \{6, 7\}
\end{array} \Rightarrow
\begin{array}{c|c|c}
A_1 & A_2 & A_3 \\
1 & 2   &   \\
1 & 3   &   \\
1 & 4   &   \\
5 & 6   &   \\
5 & 7   &   
\end{array}
\]

**Second Normal Form (2NF)**

The higher normal forms 2NF, 3NF, and BCNF (Boyce-Codd Normal Form) are based on functional dependencies involving candidate keys. For the development of these concepts, define *key* to mean “candidate key”, *key attribute* to mean an attribute in a candidate key, and *non-key attribute* to mean an attribute which is not in any candidate key.\(^2\)

**Definition 14** A FD \( X \rightarrow Y \) is a *partial FD* if some attribute \( A \in X \) can be removed, i.e., \( X - \{A\} \rightarrow Y \); otherwise, it is a *full FD*.

**Definition 15** A relation \( R \) is in *second normal form (2NF)* if:

1. \( R \) is in 1NF; and
2. for every key \( K \) and every non-key attribute \( A \) in \( R \), \( K \rightarrow A \) is a full FD (i.e., no non-key attribute is determined by anything less than a candidate key).

Note that if \( K \) consists of a single attribute, \( K \rightarrow A \) is necessarily a full FD.

**Example 103** Let \( R \) be the relation student-course(sid#, course#, grade, stud-name, course-title, course-instructor). Assume that \( \{\text{sid#}, \text{course#}\} \) is the only candidate key, and that a minimal set of FDs is:

1. \( \text{sid#} \rightarrow \text{grade} \)
2. \( \text{sid#} \rightarrow \text{stud-name} \)
3. \( \text{course#} \rightarrow \text{course-title} \)
4. \( \text{course#} \rightarrow \text{course-instructor} \)

\(^2\)A simplified presentation is often given by starting with *key* meaning “primary key”, *key attribute* meaning an attribute in the primary key, and *non-key attribute* meaning an attribute which is not in the primary key, defining the normal forms for the special case of primary keys, and then defining the generalized normal forms using candidate keys.
7.4. NORMAL FORMS

\( R \) is not in 2NF because of the FDs \( \text{sid}\# \text{ course}\# \rightarrow \text{stud-name}, \text{sid}\# \text{ course}\# \rightarrow \text{course-title}, \text{sid}\# \text{ course}\# \rightarrow \text{course-instructor} \) are partial. The relation contains redundant data, and it suffers from anomalies when a modification (insert, update, or delete) to the database makes a student with no grade, or a course without a student.

To convert \( R \) to 2NF, we split it into three relations:

1. student-course(sid\#, course\#, grade)
2. student(sid\#, stud-name)
3. course(course\#, course-title, course-instructor)

It is easy to see that the FDs of the three relations have no attributes that are partially dependent on their candidate keys. The conversion to 2NF removes redundancy and avoids the anomalies.

### Third Normal Form (3NF)

**Definition 16** A relation \( R \) is in **third normal form (3NF)** if:

1. \( R \) is in 2NF; and
2. for every attribute \( A \), for every non-trivial FD \( X \rightarrow A \), either \( X \) is a superkey of \( R \), or \( A \) is key attribute of \( R \).

**Example 104** Let \( R \) be the relation student(sid\#, stud-name, dept\#, dept-name, dept-chair). Assume that \( \{ \text{sid}\# \} \) is the only candidate key, and that a minimal set of FDs is:

1. \( \text{sid}\# \rightarrow \text{student-name} \)
2. \( \text{sid}\# \rightarrow \text{dept}\# \)
3. \( \text{dept}\# \rightarrow \text{dept-name} \)
4. \( \text{dept}\# \rightarrow \text{dept-chair} \)

\( R \) is not in 3NF, because the third and fourth dependencies have the non-superkey dept\# on the left hand side and has the non-key attributes dept-name and dept-chair on the right-hand side. \( R \) contains redundant data and is subject to anomalies (a department with no student).

It also illustrates *transitive dependencies*: \( \text{sid}\# \rightarrow \text{dept}\# \rightarrow \text{dept-name}, \text{sid}\# \rightarrow \text{dept}\# \rightarrow \text{dept-chair}. \)

---

\(^3\)An FD \( X \rightarrow Y \) for a relation \( R \) is a **transitive dependency** if, for some set \( Z \) of attributes of \( R \), \( X \rightarrow Z \) and \( Z \rightarrow Y \), and \( Z \) is neither a candidate key nor a subset of a candidate key for \( R \).

An alternative definition of third normal form is this. A relation \( R \) is in **third normal form (3NF)** if and only if:

1. \( R \) is in 2NF; and
2. no non-key attribute of \( R \) is transitively dependent on any key of \( R \).

It can be shown that this definition is equivalent to Definition 16.
We can convert \( R \) to 3NF by making \( \text{dept} \) a separate relation with the offending “man in the middle” (\( \text{dept#} \)) as its primary key:

1. student(sid#, stud-name, dept#)
2. dept(dept#, dept-name, dept-chair)

**Boyce-Codd Normal Form (BCNF)**

**Definition 17** A relation \( R \) is in **Boyce-Codd Normal Form (BCNF)** if:

1. \( R \) is in 2NF; and
2. for every non-trivial FD \( X \rightarrow A \), \( X \) is a superkey of \( R \).

Note that all BCNF relations are in 3NF, but not 3NF relations are BCNF. However, most relations arising in practice which are in 3NF are also in BCNF.

**Example 105** Let \( R \) be the relation student-course(sid#, course#, sec#). Assume that \( \{ \text{sid#}, \text{course#} \} \) is the only candidate key, and that a minimal set of FDs is:

1. \( \text{sid#} \text{ course#} \rightarrow \text{sec#} \)
2. \( \text{sec#} \rightarrow \text{course#} \)

The relation is not in BCNF, because the second FD is a non-trivial one in which the left-hand side is not a superkey. (However, since the right-hand side is a key attribute, the relation is in 3NF.) The relation contains redundancy and suffers from anomalies (a course or section without a student).

We can convert it to BCNF by splitting into two relations, with the offending left-hand side attribute sec# becoming the key of one:

1. student-section(sid#, sec#)
2. section-course(sec#, course#)

Note that in doing so, we lose the FD \( \text{sid#} \text{ course#} \rightarrow \text{sec#} \), since the three attributes are no longer in any one relation. This is undesirable, but it is offset by the elimination of redundancy. Whether 3NF or BCNF is better here is a matter of judgment.

**Fourth Normal Form**

**Fifth Normal Form**
Chapter 8

Application Design and Development

Reference: Silberschatz, Korth, and Sudarshan, Chapter 8.

8.1 Outline

- Application development tools for easily creating forms and reports.
- Web interfaces are an important way of delivering database applications.
- Triggers can be used to alert humans or take actions when certain changes to the database occur.
- Security measures protect against malicious damage to data by humans, and, in a broad sense, also from events beyond human control, such as equipment failures and natural disasters.
- Ethical responsibilities concerning privacy.

8.2 Form and Report Generator Tools

Most database users do not interact directly with the database, using query languages such as SQL, QBE, or Datalog; but indirectly, using graphical interfaces for data input (forms), or as recipients of printed reports. In this section, we consider some of the ways of producing such forms and reports.

Forms and Graphical Interfaces

By forms, we mean something graphically displayed having some or all of the following features: boxes for text input, menus, buttons, etc.
HTML forms: A program run by the web server may write HTML, including forms, to be displayed on a web browser, accept the form inputs and formulate them as an SQL query, send the query to the database (via embedded SQL, dynamic SQL, ODBC, or the like), receive the query results from the database, and format them in HTML for display in the browser.

Programming languages (e.g., C++, Java, COBOL): the program uses a graphical toolkit (such as Tk, GTK+, or the Microsoft Windows API) to produce forms, queries the database based on the form inputs, and displays the results of the query to the user.

Form editors Commercial database products typically provide productivity tools such as a “form editor,” allowing forms to be declaratively specified, automating much of the code generation.

Formated Reports

Tools:

Programming languages The program queries the database and prints the query results in the format specified by the programmer.

Report generators Database vendors supply productivity tools such as a “report generator,” allowing reports to be declaratively specified, automating much of the code generation.

Automatic Programming

Form and report generators are automatic programming tools for two common types of well-understood programming problems. The immediate output of a report generation tool, for example, might be a COBOL or C program, which is then compiled and run to produce the report. There are no standards for these productivity tools. Each database vendor supplies their own. In evaluating a database product, it is important to consider the quality of these productivity tools, as well as the SQL implementation and the efficiency and reliability of the database. High-quality tools for generating forms and reports can save many hours of programmer time. Besides saving their employer money directly, they can increase morale in the IT department, by freeing programmers from drudgery and enabling them to concentrate their efforts on non-routine tasks.

8.3 Databases and the World-Wide Web

User interfaces to databases through the World-Wide Web are covered in detail in INFO-I 320 Distributed Computing.
8.4 Triggers

The create trigger Statement. A trigger specifies that when some kind of event occurs, some kind of action will be performed. Also called active rules.

Syntax (simplified):

```
cREATE TRIGGER name [BEFORE | AFTER] EVENT event ON table
FOR EACH ROW | STATEMENT
WHEN condition
ACTION action
```

where `name` is the name of the trigger; `event` is `insert`, `update`, or `delete`; `table` is the name of a relation; `condition` is a boolean expression; and `action` is an SQL statement or sequence of statements.

Semantics: Before (or after) each `event` on `table`, check `condition`; if it is true, perform `action`.

Example 106 Triggers.

- The overdraft-trigger (SKS, p. 330, Figure 8.8) creates an automatic loan when a bank account is overdrawn.
- The reorder-trigger (SKS, p. 333, Figure 8.9) creates an automatic reorder entry in the database for an inventory item when its quantity falls below the minimum desired level.
- Note that each trigger’s direct effects are limited to the database. A trigger, by itself, cannot notify humans, sound an alarm, or send an order to a supplier.
- If external effects are desired, an external program must be run periodically to query the database and perform the external actions when the query results indicate they are needed. In Unix-like systems, such a program can be run as a “daemon,” always present and ready to act, or it can be run at specified intervals (daily, weekly, monthly) as a cron job.

Triggers and Production Rules

Triggers or active rules are similar to the production rules used in many knowledge-based expert systems. Like production rules, they can combine in chain reactions: trigger $T_1$ causes trigger $T_2$ to fire, which in turn fires trigger $T_3$, and so on. Cycles and recursion are possible.

Triggers and Referential Integrity

Triggers are usually not a good way to preserve referential integrity, because of their procedural nature. By comparison, declaratively specified “referential actions” such as `ON DELETE CASCADE` are easier to comprehend and verify (C. J. Date).
Support for Triggers

Triggers were widely used in database systems but not standardized until SQL:1999. Many database products do not conform totally to the standard.

PostgreSQL (as of version 7.2.1) supports a subset of the SQL:1999 `create trigger` statement. Unfortunately, instead of allowing the action to be written as SQL commands, it requires calling a procedure in another language (C, Tcl, or PL/pgSQL).

PostgreSQL also provides a `create rule` statement, which is similar in spirit to `create trigger` but totally nonstandard. It does allow an SQL action, however.

8.5 Security

Security measures are designed to protect the database against malicious actions or intrusions, such as the unauthorized reading, writing, or destruction of data.

No system is 100% secure. Even if we locked up our database system in a bank vault, it could be stolen by a determined criminal or destroyed by a hostile military force or natural disaster. Reasonable security measures are designed to deter malicious access by making it very difficult. System and database administrators must judge the level of security they desire, according to the sensitivity and value of the data and the cost of protecting it.

Levels of Security

Security is needed at many levels:

**Physical**: Is the database server located in a locked machine room? How hard is it for an unauthorized person to get in?

Physical security, in a wider sense, extends to measures taken against non-human threats. Fire, flood, and disk drive crashes can be countered by routinely backing up essential data and storing the backups in a secure off-site location (why?). Backup (“uninterruptible”) power supplies allow the system to continue operating or shut down safely in case of loss of electrical power.

What the security consultant found about locks and passwords . . . .

Professor C’s password . . . .

**Human**: Are employees honest? Can they be bribed? Well treated?

**Operating system**: Deficiencies in OS security may open the system to unauthorized access of the database. Are passwords easy to crack? Does the OS run obsolete, insecure versions of essential services? Does it run unnecessary system services?
8.5. SECURITY

**Network:** Deficiencies here too may open a path to the database. Does the server run unnecessary and insecure services? Does a firewall restrict incoming connections? Are the data communication channels (e.g., wires or open air) secure?

(In 2003, intrusions had increased 300,000% since 1990. On the average, a new host is on the Internet for one minute before being probed. Firewalls are useful, since 40% of intrusions are external, 60% internal.)

In addition to a company firewall, which decides the fate of all packets trying to come in from outside, the database server typically can be configured to control what connections it accepts. For example, in PostgreSQL, rules can determine the acceptance or rejection of a connection based on: the client’s IP address; which database; which user; with or without SSL (Secure Socket Layer); and methods of authentication, such as password, MD5 (an encrypted challenge/response password method), Kerberos.

**Database system security:** DBA should authorize individual users and groups to only the privileges they need. Audit trails may be used to log all changes to the database and identify the responsible user.

Any one of these levels of security could easily be described at book length. We confine our attention, from now on, to database system security. Topics: creating users and groups (or roles), granting and revoking privileges, and audit trails.

**Creating and Modifying Users and Groups**

To create and modify users and groups, PostgreSQL provides the statements

**create user, alter user, drop user, create group, alter group, and drop group.** SQL standards do not include these statements. However, SQL:1999 roles are similar to groups.

The psql `du` command lists database users.

**Example 107** See see Listing 8.1.

Listing 8.1: users.sql, source code.

```sql
create user david;
create user dwoods password 'axmsbtz89';
```

---

**File:** users.sql

**Create, alter, and drop users and groups.**

**Only a PostgreSQL database 'superuser' can create, alter, or drop users and groups.**
create group accounting
  user david, dwoods;

create group personnel;

--- rforrest may create new databases and new users
--- (superuser)

create user rforrest
  password '077ak'
in group personnel
  createdb
  createuser;

--- hpomp is a member the accounting and personnel
--- user groups; account expires at the start of

create user hpomp
  password 'circ!!umstance'
in group accounting, personnel
  valid until 'May 1, 2003';

--- Change some privileges

alter user david password 'vx<<guay'
createdb;

alter user rforrest nocreateuser;

--- Change group membership
--- 'drop user' only drops from group

alter group accounting
  drop user hpomp, dwoods;

alter group personnel
  add user dwoods;

--- Clean up

drop group accounting;
drop group personnel;

--- Clean up

drop user david;
drop user dwoods;
drop user rforrest;
drop user hpomp;

Controlling Privileges

The grant and revoke statements can be used to control privileges on a table. Privileges can be classified as “read” (select) or “write” (insert, update, delete):

select: user may “read” the table with the select and copy from statements.

insert: user may modify the table with the insert and copy to statements.

update: user may update tuples in the table.

delete: user may delete tuples from the table.

all privileges: all of the above.

Each privilege is extended to one or more objects, such as a table or view. SQL92 also allows privileges on individual columns; PostgreSQL does not, but if only the select privilege is needed, the same result can be achieved by creating a view, and granting the privilege on the view; however, we need triggers otherwise, since PostgreSQL views do not allow insert, update, or delete.

The privileges are extended to a named user or group or to public (all users). Privileges for an object can be granted and revoked only by object’s creator (owner) and by database superusers. An object’s owner normally has all privileges for that object, but can revoke any of them for safety. Database superusers have all privileges on all objects.

The psql commands dp and z list user privileges(e.g., z to list privileges on all tables, z loan to list privileges for the loan table).

Example 108 See see Listing 8.2.

Listing 8.2: privileges.sql, source code.

--- File: privileges.sql
2 --- Grant and revoke privileges to tables.
2 --- Only a PostgreSQL database 'superuser' can create,
2 --- alter, or drop users and groups.

7 create group loanofficer;
create group personnel;

create user david in group loanofficer;
create user dwoods in group loanofficer;
12 create user rforrest in group personnel;
--- A manager:
create user hpomp in group personnel;
CHAPTER 8. APPLICATION DESIGN AND DEVELOPMENT

−− The general manager:
create user lmonk in group loanofficer, personnel;
create user jkonk;

−− Create employee table and privileges

cREATE TABLE employee ( 
    ssn INTEGER PRIMARY KEY,
    check (ssn > 0 and ssn <= 999999999),
    name VARCHAR(40),
    address VARCHAR(40),
    hired DATE,
    department VARCHAR(5),
    salary NUMERIC(8, 2)
);

GRANT SELECT ON employee TO group personnel;
GRANT INSERT, UPDATE, DELETE ON employee TO hpomp;

−− Create loan table and privileges

cREATE TABLE loan ( 
    cid CHAR(5) PRIMARY KEY, -- customer ID
    init DATE, -- date loan opened
    due DATE, -- to be paid in full
    init_balance NUMERIC(8,2), -- opening balance
    cur_balance NUMERIC(8,2) -- current balance
);

GRANT ALL PRIVILEGES ON loan TO group loanofficer;
GRANT SELECT ON loan, employee TO jkonk;
REVOKE INSERT, DELETE, UPDATE ON loan FROM dwoods;

−− Clean up

drop user david;
drop user dwoods;
drop user rforrest;
drop user hpomp;
drop user lmonk;
drop user jkonk;

drop group loanofficer;
drop group personnel;
drop table loan;

62 drop table employee;

**Limitations:** The SQL grant statement extends privileges for an entire table or for selected attributes of a table. It does not allow row-level control of privileges. But sometimes row-level access is needed. For example, students using the university registrar’s database should be able to view (but not change!) their own grades, but not those of other students. Since SQL does not provide this level of control, it must be provided by another agency, either human (people who work in the registrar’s office) or software (a user interface for the database)—where the software, not the student, is the real database user and has the privileges.

**Audit Trails**

An audit trail is a log file which records all changes (insertions, updates, deletions) to a database, who made the changes, and when. It can be used to trace errors back to the person who made them, either willfully or unintentionally, and thence to other changes made by the same person, which also may need to be corrected. How to set up an audit trail varies widely from one database system to another. One way to do it is with triggers, but many database systems provide a more direct and convenient mechanism.

### 8.6 Ethical Responsibilities

Security measures are required not only to protect a company’s interests, but to ensure the privacy rights of its customers, employees, and others about whom it stores personal data.

Database administrators/developers have a duty to safeguard the privacy of personal information collected in their databases. In addition, users of database systems should take care that the personal data they enter is accurate. Developers should design the system to support and encourage accuracy.

**ACM Code of Ethics**

The Association for Computing Machinery\(^1\), in its *Code of Ethics\(^2\)*, calls on members to “respect the privacy of others.”

- Access to personal data should be limited to those with a legitimate need to know. E.g., doctors in a hospital need to know their patients’ past medical histories. The hospital’s billing and maintenance departments do not.

\(^1\)URL: http://www.acm.org/

\(^2\)URL: http://www.acm.org/constitution/code.html
• Personal data should not be collected for one purpose and then used for another, without the permission of the persons it describes. For example, a prospective home buyer provides employment, salary, bank account, and loan data to the mortgage lending company for the purpose of credit evaluation. The mortgage company should not use this information to identify people who might want to buy a new car, nor should it provide the information to other firms for such use.

• Individuals should be given opportunities to review their records and correct errors.

• Personal data should be stored only as long as needed and then destroyed. The database administrator should set a “lifetime” (retention period) for personal data. Data should be regularly destroyed at the end of its lifetime.

Discussion

Experiences with privacy of personal information in databases.
What personal data is retained in NSMI’s database? What privacy safeguards should NSMI establish? Should it publicize its privacy policy, and if so, how?
Chapter 9

Physical Design of Databases

Reference: Silberschatz, Korth, and Sudarshan, Chapters 11–12.

The logical design of a database is concerned with what data are to be stored; the physical design is concerned with how the data are to be stored. Two important aspects of physical design are storage and file structures, and indexes.

9.1 Storage Hierarchy

- See SKS Figure 11.1, page 444.
- Speed
- Cost per bit of storage
- Size
- Levels:
  - Cache
  - Main memory
  - Flash memory
  - Magnetic disk
  - Optical disk
  - Magnetic tape

9.2 Disk Organization

- See SKS Figure 11.2, page 445.
• Platters and surfaces
• Read-write head
• Arm
• Tracks
• Cylinders
• Sectors

Disk Blocks
• A block is a contiguous sequence of sectors on one track of one surface.
• Block sizes are 512 bytes or higher.
• A block is the normal unit of transfer between disk and main memory. I.e., the operating system will not (normally) read or write less than one block of a disk. I/O buffers in main memory make this possible.

9.3 Time for Disk I/O
• Access time is the time between a request for I/O and the time data transfer begins.
• Access time = seek time + rotational latency
• Data transfer time is the time between the first and last bit of data in or out.
• Response time = access time + data transfer time

Typical Performance Characteristics
• Seagate Barracuda ES.2, 2009, SATA interface, 7200 RPM, 4 platters, 512 bytes/sector
• Average access time, milliseconds:

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seek time</td>
<td>8.5</td>
<td>9.5</td>
</tr>
<tr>
<td>Rotational latency</td>
<td>4.16</td>
<td>4.16</td>
</tr>
<tr>
<td>Access time</td>
<td>12.66</td>
<td>13.66</td>
</tr>
</tbody>
</table>

• Data transfer rates (MB/second):
9.4 INDEXING OVERVIEW

<table>
<thead>
<tr>
<th>Maximum</th>
<th>1287</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sustained</td>
<td>105</td>
</tr>
</tbody>
</table>

Example 109 Calculating Disk I/O Time

1. Estimate the time to read two consecutive 512-byte blocks of data, from the time of the request to the time the last byte is transferred. Assume the disk performance statistics on the previous slide.

2. How long does it take to repeat this 1000 times?

9.4 Indexing Overview

- Motivation
- Tree indexes
- Hash indexes
- SQL statements

9.5 Motivation

An index for a relation—like the index of a book—can make some queries much more efficient, because the DBMS does not have to examine every tuple in the relation. A query optimizer is a component of the DBMS that tries to find the most efficient ways to evaluate a query.

An indexed attribute or set of attributes is called a search key. Note that a search key is not necessarily a primary key, candidate key, or superkey.

Example 110 Speedup from Indexing

Consider the relation part(part#, name, color, cost) and the query

```sql
select * from part where color = 'red';
```

Suppose there are 200,000 part tuples, including 50,000 with color “red”. If color is not indexed, the DBMS must examine the color of every tuple. If it is indexed, the DBMS can look up “red” in the color index and immediately find the tuples that are red. Reducing the number of disk accesses speeds up the query. Similarly, an index has the potential to speed up update and delete statements with “where” conditions.
Which Keys to Index?

The primary key of a relation is normally indexed (automatically, in PostgreSQL). The database developer may also choose to create additional indices.

Why not just index every attribute of every relation? Having an index is not free. Table operations (insert, update, delete) require extra time to update the index. The index also requires disk space.

Use judgment to weigh costs and benefits of indexes.

9.6 Kinds of Indexes

There are two major kinds of indexes. Tree indexes have search time \( O(\log N) \) to find a single value. Hash indexes have search time \( O(1) \) to find a single value. Tree indexes support efficient sorting and finding a range of values; hash indexes do not.

Most of the material on tree indexes and hash indexes should be familiar to those who have taken CSCI-C 243, Introduction to Data Structures.

9.7 Tree Indexes

Balanced binary search trees guarantee time is \( O(\log N) \) to find a single value. For large indexes that need to be stored on disk, trees with higher degree (branching factor) are better. Each node stores as many branches as fit in a disk block. This minimizes the number of disk accesses. Search time is still \( O(\log N) \), but with a lower constant factor. B-trees and \( B^+ \)-trees are examples of high-degree, balanced search trees.

Example 111 A B-Tree Index

See SKS, fig. 12.19, p. 501. How does it work?
How can a tree index be used to sort records?
How can a tree index be used to find records based on inequality conditions, e.g., search key values less than “E”?

9.8 Hash Indexes

- Hash function
- Buckets (array or random-access disk blocks)
- Chaining
- With dynamic hashing, a hash index can automatically readjust its size.
- Search time in a hash index of size \( N \) is \( O(1) \), but that does not guarantee that it is faster than a B-tree.

Why not?
Example 112 A Hash Index

- See SKS, figure 12.23, p. 511.
- Can the hash index be used to sort records or to find records based on inequality conditions?

9.9 SQL Statements

SQL standards do not provide statements to create indexes. The standards deal only with the logical schema of a database. Indexes belong to the physical schema. Most SQL implementations provide such statements.

The syntax for PostgreSQL is typical.

```sql
create index name on table
[using access-method] (column [, . . .]);
```

where `name` is the name of the index, `column` is the search key in `table`, and `access-method` is the type of index, such as tree or hash index.

An index can be removed by the `drop index` statement.

Index Types in PostgreSQL

PostgreSQL supports four types of index, or access methods:

1. `btree`
2. `hash`
3. `rtree`
4. `gist`

The default index type is `B-tree`. GIST stands for “Generalized Index Search Tree”. The B-tree and GIST methods allow multi-column search keys. R-tree is an extension of B⁺-trees suitable for geographic data types. All of the indexing methods are dynamic; they do not require periodic adjustment.

Example 113 Indexing the Product–Distributor Database

See see Listing 9.1.

Listing 9.1: `index.sql`, source code.

```sql
File: index.sql
2 Create some indexes on the product – distributor database
   These indexes are totally useless, because our relations are so small.
   See what indexes we’ve got already
```
The following index in a larger database, could speed up queries on product with "where pname = 'motorboat', etc."

```sql
create index product_name_index on product(pname);
```

The following index, in a larger database, could speed up queries on product with "where length < 10 and length < 33", etc.

```sql
create index product_length_index on product(length);
```

The following index, in a larger database, could speed up queries on distributor with "where dcity = 'Chicago', etc.

```sql
create index distributor_city_index on distributor using hash (dcity);
```

Clean up

```sql
drop index product_name_index,
    product_length_index,
    distributor_city_index;
```

Query Optimization in PostgreSQL

The query optimizer may consider using indexes. The query optimizer will consider using a B-tree index when the query makes a comparison of an indexed attribute using any of the operators $\lt, \leq, \gt, \geq, \eq$. The query optimizer will consider using a hash index when the query makes a comparison of an indexed attribute using the operator $\eq$.

The query optimizer uses statistics to estimate the running times of each query execution plan it considers.

The (non-standard) `analyze` and `explain` statements can be used for “tuning” the query optimizer and/or indexes. The `analyze` statement updates statistics used by the query optimizer. The statistics describe the distribution of values in each table. The query optimizer uses the statistics to estimate the execution costs (number of rows returned by subqueries) of an execution plan for a query. It cannot do a good job without accurate statistics. The `explain` statement describes the optimizer’s execution plan for a query. This can show whether the plan uses an index or not, which may suggest why the index is or is not useful. For details, see the PostgreSQL User’s Guide: chapter 7, “Indexes”; chapter 11, “Performance Tips.”
Chapter 10

Transactions

Reference: Silberschatz, Korth, and Sudarshan, Chapter 15.

10.1 Outline

• The concept of transaction
• States of a transaction
• Transactions in SQL

10.2 The Concept of Transaction

A transaction is a sequence of operations that form an indivisible whole: either they all should succeed, or none. A transfer of funds from one account to another, in a bank, is a typical transaction. The amount of money must be debited from one account and credited to the other. If one, but not both, of the account updates succeeds, the transfer has not been made correctly.

ACID Properties

Transaction management in a (good) DBMS is characterized by the ACID properties: Atomicity, Consistency, Isolation, Durability.

Atomicity: Either all operations of the transaction, or none, succeed. A transaction is not supposed to end halfway through, leaving the database in an inconsistent state. Responsibility of the DBMS (transaction management subsystem). Achieved by writing a log file; if the transaction cannot complete successfully, the log file is consulted to find changes that have taken place in the transaction and the DBMS automatically undoes them.
Consistency: A transaction executed in isolation (i.e., with no concurrent other transaction) leaves the database in a consistent state. Responsibility of the programmer.

Isolation: Every execution of a pair of concurrent transactions \( T_1, T_2 \) is equivalent to the execution of one before the other (i.e., either \( T_1 \) finishes before \( T_2 \) starts, or \( T_2 \) finishes before \( T_1 \) starts). Furthermore, each of the concurrent transactions is oblivious to the other. A select query executed within \( T_1 \) will not reveal any changes made by \( T_2 \), or vice-versa. Responsibility of the DBMS (concurrency control subsystem, details chapter 16).

Durability: Changes made to the database persist. We do not want to have a system crash and find that changes that were buffered in RAM have evaporated before being written to disk. Either the changes themselves, or sufficient information to infer them, must be written to disk before the transaction completes. Responsibility of DBMS (recovery management subsystem).

Example 114 Isolated Transactions

Consider the transactions

\[
T_1 : (A \leftarrow A - 1; B \leftarrow B + 1)
\]

and

\[
T_2 : (C \leftarrow A + B)
\]

Assume that initially \( A = B = 100 \). The following “schedules” are possible:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>( T_1 )</td>
<td>( T_1 )</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>( T_2 )</td>
<td>( T_2 )</td>
</tr>
<tr>
<td>( A = 99 )</td>
<td>( C = 200 )</td>
<td>( A = 99 )</td>
</tr>
<tr>
<td>( B = 101 )</td>
<td>( A = 99 )</td>
<td>( C = 199 )</td>
</tr>
<tr>
<td>( C = 200 )</td>
<td>( B = 101 )</td>
<td>( B = 101 )</td>
</tr>
</tbody>
</table>

However, schedule 3 is not allowed, because it violates the isolation principle.

10.3 States of a Transaction

Atomicity requires that a transaction terminate in one of two ways:

- It ends successfully (the transaction is committed), or

- It aborts, and all changes made to the database by the transaction so far have to be undone (it is rolled back).

A transaction passes through three of five possible states (see SKS, Figure 15.1, p. 614):

\[
\text{Schedule 1} \quad \text{Schedule 2} \quad \text{Schedule 3}
\]

| \( T_1 \) | \( T_1 \) | \( T_1 \) |
| \( T_2 \) | \( T_2 \) | \( T_2 \) |
| \( A = 99 \) | \( C = 200 \) | \( A = 99 \) |
| \( B = 101 \) | \( A = 99 \) | \( C = 199 \) |
| \( C = 200 \) | \( B = 101 \) | \( B = 101 \) |
10.4 Transactions in SQL

1. According to the standard, a transaction begins with the start transaction statement, or implicitly with any statement.

2. Either the commit (or commit work) statement or the rollback (or rollback work) statement terminates a transaction.

The next transaction begins with the next statement.
Autocommit

Many database systems, including PostgreSQL, provide an “autocommit” feature: by default, a single SQL statement is implicitly followed by a commit. To avoid autocommit in PostgreSQL, use start transaction or begin to start the transaction.

Example 115 An SQL Transaction

--- (PostgreSQL, non-standard)
start transaction;
update A . . ;
update B . . ;
commit; — terminates transaction

Example 116 Transfer of Funds

Execute the transactions in see Listing 10.1: (a) in isolation; (b) concurrently, with two login sessions; (c) with rollback or disconnect.

Listing 10.1: transaction.sql, source code.

--- File: transaction.sql
--- Creates tables to illustrate transfer of funds
--- transaction. Try executing the transactions:
--- (a) in isolation; (b) concurrently, with two logons;
--- (c) with a rollback or disconnect.

--- Create and populate account table

create table account (acct_num integer primary key,
    name varchar(40) not null,
    balance numeric(10, 2) not null default 1
    check (balance >= 0),
    interest_earned numeric(10, 2) not null default 0
    check (interest_earned >= 0)
);

insert into account(acct_num, name, balance)
values(1, 'James Smith', 100);

insert into account(acct_num, name, balance)
values(2, 'James Smith', 100);

select *
from account;
10.4. TRANSACTIONS IN SQL

--- Transaction to transfer $1 from account 1 to account 2.

start transaction;

update account
  set balance = balance - 1
  where acct_num = 1;

update account
  set balance = balance + 1
  where acct_num = 2;

commit;

--- Transaction to credit interest to all accounts

--- Interest rate is 1% per month
--- Smith's total interest should be $2.00

start transaction;

update account
  set interest_earned = interest_earned + (balance / 100),
  balance = balance + (balance / 100);

commit;

--- Clean up

start transaction;
  update account
  set balance = 100,
  interest_earned = 0;

commit;

drop table account;
Chapter 11

Data Analysis and Mining

Reference: Silberschatz, Korth, and Sudarshan, Chapter 18.

11.1 Introduction

• Contrast routine transaction processing vs. decision support.

• Routine:
  – Purchases.
  – Customer billing.
  – Payroll, etc.

• Decision support:
  – Where to build new factory or warehouse.
  – What products to stock, in what quantities.
  – How to attract new customers.

Decision Support Systems (DSS)

• Kinds include:
  – Online Analytical Processing (OLAP).
  – Statistical analysis.
  – Knowledge discovery in databases (KDD, “data mining”).
  – Data warehouses.
CHAPTER 11. DATA ANALYSIS AND MINING

Outline
1. OLAP
2. Data warehouses
3. Data mining

11.2 Online Analytical Processing (OLAP)

- Goal: reasonably efficient processing of analytical queries typical of decision support. Typically, these queries involve aggregation with grouping. For example, what are the sales volumes of our stores, grouped by region? If there are gigabytes of data, that could take a long time. We want a response within a few seconds.

- SKS Figure 18.1, p. 726, shows a cross-tabulation ("crosstab") of sales data.
  - *Dimensional attributes*, used for grouping: item-name, color.
  - *Measure attribute(s)*, quantities aggregated: number (sold).
  - Summary is derived from table sales(item-name, color, size, number, location, date-time, sales-price, ...)
  - Illustrates *multi-dimensional data*, which consists of dimension attributes and measure attributes.
  - What is the practical value of this crosstab?

OLAP Implementation

- How to make such queries efficient.
- Essential method: *precompute* aggregations with grouping. Are these materialized views?
- Complexity:
  - $n$ dimensions $\Rightarrow 2^n$ possible subsets (combinations) for summaries.
  - Impractical to compute all if $n$ is large.

\[
\begin{align*}
2^{10} &= 1024 \\
2^{20} &= 1,048,576 \\
2^{30} &= 1,073,741,824 \\
2^{40} &= 1,099,511,627,776 \\
2^{50} &\approx 1.13 \times 10^{15} \\
2^{100} &\approx 1.27 \times 10^{30}
\end{align*}
\]

- So, just precompute some of the combinations; derive others as needed from those already computed.
MOLAP and ROLAP

- Early *multidimensional OLAP (MOLAP)* systems stored data summaries as multidimensional arrays.
  
  - See data cube, \( n = 3 \), in SKS Figure 18.3, p. 728

- Later *relational OLAP (ROLAP)* systems stored data summaries as relations.
  
  - See SKS Figure 18.2, p. 727
  - *all* indicates a total or subtotal.

Drilling Down and Up

- Another feature of OLAP is the ability to present summaries of the same data at different levels of granularity.

- Examples:
  
  - State, county, city, neighborhood.
  - Century, year, month, day, hour, h:m, h:m:s.

  - See SKS Figure 18.4, p. 729

- *Drill down:* view moves to finer level of detail.

- *Drill up:* move to more abstract view (coarser level of detail).

Skip SQL:1999 Features

- Extended Aggregation

- Ranking

- Windowing

11.3 Data Warehousing

A *data warehouse* is a collection of data—

- Gathered from multiple sources;

- Stored at one site;

- Stored for “a long time.”
Advantages of a Data Warehouse

- Gives corporate decision makers access to data from the entire organization. A large organization is spread out both geographically (e.g., thousands of stores or local offices) and functionally (i.e., different departments such as manufacturing and sales). The DW “pulls in” data from both geographical and functional areas.

- Access to historical data. Historical data are valuable for predicting trends.

- Divides the computing load. If the data warehouse is housed in a separate computing system, then DSS processing does not “hog the machine” which is also used for routine operations.

Data Warehouse Issues

- How to update the DW. It can be updated continuously, or in “batch” mode during the night, for example. Either way, the DW will be slightly out of date, but not so much as to be a problem. The DW can “pull” data from the sources, or the sources can “push” data into the DW.

- Integrating different schemata used by different sources for similar data.

- Which data to store in summary form only? Typically, raw data is too voluminous to be stored in its entirety for long periods of time.

11.4 Data Mining

- What is it?
  - Knowledge Discovery in Databases (KDD)
  - Large-scale machine learning. SKS write, p. 740: “However, data mining differs from machine learning and statistics in that it deals with large volumes of data, stored primarily on disk.” It would be better to say that machine learning and statistics have *customarily* dealt with smaller sets of data; but recently they have been scaling up to deal with very large data sets, and the result is KDD.

- Tasks of KDD:
  - Learning to predict.
    - Classification is prediction of a symbolic category, such as healthy or diseased.
    - Regression is prediction of a numerical value, such as sales volume.
  - Learning associations: things that go together.
* Amazon.com’s book recommendations.

- This presentation will focus on the classification task.

**Examples of Real KDD Tasks**

- A credit card company looks for signs of unauthorized transactions (stolen card).

- Data Mining Cup 2007: Help retailer identify customers who can be drawn back with coupons, and what type of coupons they will respond to.

- KDD Cup 2006: Identify pulmonary embolisms from 3D computed tomography (CT scan) data.

- KDD Cup 2004 (particle physics) distinguish between two types of particles in high energy collider experiments, 78 attributes, 50,000 training examples, 100,000 test examples.

- Predict alcoholism from EEG data. See “plots of normal and alcoholic subjects.”\(^1\)

- Predict protein function from bacteria genomes (E. coli and M. tuberculosis).

- The University of California at Irvine maintains a repository of KDD problems and data (*UCI KDD Archive*\(^2\)).

**Classification Tasks**

**“Batch” Classification Learning**

- The traditional approach.

- Separates training from application:
  1. Divide data into *training set* and *test set*.
  2. Train with training set.
  3. Test with test set.

- Concept is fully formed at end of training and can’t be\(^3\) used before then.

- Problems:
  1. Bad for autonomous agents (robots, softbots).
  2. Bad for KDD: doesn’t train an all available data. Memory can store only a small number of training examples. Even if there’s room for all of it in memory, don’t train on the test data—that would be considered cheating.

---

\(^1\)URL: http://kdd.ics.uci.edu/databases/eeg/eeg.data.html

\(^2\)URL: http://kdd.ics.uci.edu/

\(^3\)Or isn’t.
Incremental Classification Learning Task

- Process one example at a time.
- With each example, first test, then train.
- Concept develops gradually, is never fully formed, can be used at any time.
- Advantages:
  - Lower memory requirements.
  - Uses all data for training.
  - 'Bots can start to use results of learning immediately.
- Disadvantages:
  - Concept learned can vary with order of presenting examples.
  - Concept learned may be less good than batch learning on same training data.

Methods of Classification Learning

- Knowledge representation
- Algorithms

Knowledge Representation

- Decision trees
- Rules
- Graphs:
  - Neural networks
  - Bayesian networks

Example 117 A Decision Tree
See Figure 11.1

Example 118 A Rule Set
See Table 11.1

Algorithms for Classification Learning

- Learning as search
- Decision trees: “divide and conquer”
- Rule learning: “separate and conquer”
- Graph learning (neural, Bayesian)
- Genetic algorithms: evolutionary computation.
1. If size = small and shape = round, then yes.
2. If size = small and shape = square, then no.
3. If size = large, then no.

Table 11.1: A rule set for the concept “small round thing.” If more than one rule applies, the classifier may let them “vote” or use some other means of conflict resolution.
Learning Decision Trees

A greedy (hill-climbing) algorithm to induce a decision tree from a set $X$ of training examples:

- Find best attribute $A$ for splitting.
- “Best” refers to some measure of quality such as:
  - Accuracy: $\max(p_i)$
  - Entropy: $\sum_{i=1}^{K} p_i \log p_i$
  where $p_i$ = probability of class $i$ and $K$ = number of classes. When $K = 2$, entropy and accuracy give the same rankings.

- Split tree on $A$ and recursively apply algorithm to resulting subtrees.

- Base case: if $X$ is “good enough,” then return a leaf node with the most frequent class of $X$.

**Algorithm IDT($X$)**

```plaintext
if $X$ is good enough
    return leaf labeled with $X$’s most common class.
else
    $A \leftarrow$ best attribute for splitting $X$
    $T \leftarrow$ new tree
    $\text{Root}(T) \leftarrow A$
    for each $A_i$ in Domain($A$) do
        $\text{Branch}(T, i) \leftarrow A_i$
        $\text{Subtree}(T, i) \leftarrow \text{IDT}($\{ $x \in X : x[A] = A_i$\}$)
    end for
    return $T$
end if
```

The name IDT stands for “Induce Decision Tree”; it is not the “official” name of any well-known algorithm, but is based on the algorithms of ID3, ASSISTANT, etc.

**Example 119 Learning a Decision Tree**

Apply the IDT algorithm to the problem of learning a decision tree for the “small ball” concept, given the training examples in Figure 11.2.

There are three attributes: size (large, small); shape (round, square); filled (yes, no).
Use accuracy as the measure of quality, and define “good enough” to mean quality $\geq 0.90$.

**First Level Splits**

- Not good enough (quality = 3/10).
- Consider splits for level one:

  **Size**

  ![Size examples](image)

  Quality = $\frac{3}{5} \cdot \frac{5}{10} + \frac{5}{5} \cdot \frac{5}{10} = \frac{8}{10}$

  **Shape**

  ![Shape examples](image)

  Quality = $\frac{3}{5} \cdot \frac{5}{10} + \frac{5}{5} \cdot \frac{5}{10} = \frac{8}{10}$

  **Filled**

  ![Filled examples](image)

  Quality = $\frac{3}{5} \cdot \frac{5}{10} + \frac{4}{5} \cdot \frac{5}{10} = \frac{7}{10}$

- Pick size, breaking tie arbitrarily.

**Second Level Splits**

- “Large” subset is good enough (quality = 1).
- “Small” subset is not; it must be split. Consider:

  **Shape**

  ![Shape examples](image)

  Quality = $\frac{3}{3} \cdot \frac{3}{5} + \frac{2}{2} \cdot \frac{2}{5} = 1$

  **Filled**

  ![Filled examples](image)

  Quality = $\frac{2}{3} \cdot \frac{3}{5} + \frac{1}{2} \cdot \frac{2}{5} = \frac{3}{5}$

- Pick shape.
- Both “round” and “square” subsets are good enough (quality = 1). Stop.

**Summary of the Process**
The Resulting Decision Tree

Learning Rules for Classification

A simplification of the CN2 rule learning system. Given a set of training examples $X$:

**Algorithm LearnAllRules** ($X$)

```
RuleSet ← ∅
for each class $C$ do
    RuleSet ← RuleSet ∪ RulesForClass($X, C$)
end for
M ← most frequent class of $X$
DefaultRule ← (if true then class = $M$)
return RuleSet ∪ DefaultRule
```

The default rule guarantees that every instance can be classified. Even if no other rule applies, the default rule does, because it has no conditions (equivalently, its condition is “true”).

**Algorithm RulesForClass** ($X, C$)

```
Rules ← ∅
progressing ← true
while $X$ contains examples of $C$ and progressing do
    $R$ ← BestRule($X, C$)
    if $R$ is nil then
        progressing ← false
    else
```

...
11.4. DATA MINING

\[
\text{Covered} \leftarrow \{ x \in X : x \text{ is an example of } C \text{ and } R \text{ covers } x \} \\
X \leftarrow X \setminus \text{Covered} \\
\text{Rules} \leftarrow \text{Rules} \cup \{ R \}
\]

end if
end while
return Rules

A rule covers an example if the example satisfies the conditions of the rule.

The BestRule algorithm, hill-climbing version. CN2 actually uses a more sophisticated beam search, which allows it to “back up” to some extent in case of going down the wrong path.

Algorithm BestRule \((X, C)\)

\[
\begin{align*}
\text{InitialRule} & \leftarrow (\text{if true, then class } = C) \\
\text{BestRule} & \leftarrow \text{InitialRule} \\
\text{finished} & \leftarrow \text{false} \\
\text{while not finished do} \\
R & \leftarrow \text{best one-step refinement of } \text{BestRule} \\
\text{if } R \text{ is better than } \text{BestRule} \text{ then} \\
& \quad \text{BestRule} \leftarrow R \\
\text{else} \\
& \quad \text{finished} \leftarrow \text{true} \\
\text{end if} \\
\text{end while} \\
\text{if } \text{BestRule} \neq \text{InitialRule} \text{ and } \text{BestRule} \text{ is statistically significant then} \\
& \quad \text{return } \text{BestRule} \\
\text{else} \\
& \quad \text{return nil} \\
\text{end if}
\end{align*}
\]

Example 120 Learning Rules

- Learn one rule for the positive class from the training data of Figure 11.2.
- Initial rule: (if true then class = +).
- Evaluate candidate conditions for first refinement:

  \begin{align*}
  \text{size} = \text{small} & \quad \bullet \quad \Box \quad \Diamond \quad \bigcirc \\
  \text{Quality} & \quad 3/5 \\
  \text{size} = \text{large} & \quad \bigcirc \quad \Box \quad \bullet \quad \bigcirc \quad \bigbox \\
  \text{Quality} & \quad 0/5 \\
  \text{shape} = \text{round} & \quad \bigcirc \quad \bullet \quad \bullet \quad \bigcirc \quad \bigbox \\
  \text{Quality} & \quad 3/5
  \end{align*}
Learning Rules, Continued

- Best rule so far is (if size = small, then class = +).
- Candidate additional conditions for second refinement:

  - shape = round is best.
  - Best rule now: (if size = small and shape = round, then class = +).
  - Best rule has 100% accuracy; cannot be improved.

Learning Rules, Concluded

- Now remove from training set the examples covered by the rule just learned.
- Remaining in training set: 

  - No positive examples remain.
  - Rule learning for class + is finished.
  - Similarly, learn rules for class −.
  - Add default rule.
11.4. DATA MINING

Overfitting and Pruning

- Overfitting can result in good predictions on the training set, poor predictions on other examples.
- Affects both tree and rule learning techniques.
- Various forms of pruning are added to reduce overfitting.
- CN2’s statistical test is an example of pruning.

ILP and (Multi-)Relational Learning

- Algorithms as presented apply to flat files or single relation.
- Multi-relational forms also exist:
  - FOIL and many others (rules).
  - TILDE (decision trees).
- Derived from inductive logic programming, originally aimed at automatic creation of logic programs.