Relational and Object-Oriented Databases

by
Willi-Hans Steeb
International School for Scientific Computing

Contents

1 What is a table? 1

1.1 Introduction . . . 1
1.2 Examples . . . 5
1.3 Tables in Programs . . . 8
1.4 Table and Relation . . . 33

2 Structured Query Language 35

2.1 Introduction . . . 35
2.2 Integrity Rules . . . 38
2.3 SQL Commands . . . 39
2.3.1 Introduction . . . 39
2.3.2 Aggregate Function . . . 40
2.3.3 Arithmetic Operators . . . 40
2.3.4 Logical Operators . . . 40
2.3.5 SELECT Statement . . . 41
2.3.6 INSERT Command . . . 45
2.3.7 DELETE Command . . . 46
2.3.8 UPDATE Command . . . 47
2.3.9 CREATE TABLE Command . . . 48
2.3.10 DROP TABLE Command . . . 51
2.3.11 ALTER TABLE Command . . . 52
2.4 Set Operators . . . 53
2.5 Views . . . 60
2.6 Primary and Foreign Keys . . . 62
2.7 Datatypes in SQL . . . 63
2.8 Joins . . . 66
2.9 Stored Procedure . . . 71
2.10 MySQL Commands . . . 72
2.11 Cursors . . . 73
2.12 PL and SQL . . . 75
2.13 ABAP/4 and SQL . . . 76
2.14 Query Processing and Optimization . . . 77

3 Normal Forms **83**
3.1 Introduction . . . 83
3.2 Anomalies . . . 87
3.3 Example . . . 89
3.4 Fourth and Fifth Normal Forms . . . 93

4 Transaction **101**
4.1 Introduction . . . 101
4.2 Data Replication . . . 107
4.3 Locks . . . 108
4.4 Deadlocking . . . 111
4.5 Threads . . . 117
4.5.1 Introduction . . . 117
4.5.2 Thread Class . . . 119
4.5.3 Example . . . 121
4.5.4 Priorities . . . 123
4.5.5 Synchronization and Locks . . . 126
4.5.6 Producer Consumer Problem . . . 131
4.6 Locking Files for Shared Access . . . 134

5 JDBC **137**
5.1 Introduction . . . 137
5.2 Classes for JDBC . . . 140
5.2.1 Introduction . . . 140
5.2.2 Classes DriverManager and Connection . . . 141
5.2.3 Class Statement . . . 144
5.2.4 Class PreparedStatement . . . 147
5.2.5 Class CallableStatement . . . 149
5.2.6 Class ResultSet . . . 151
5.2.7 Class SQLException . . . 154
5.2.8 Classes Date, Time and TimeStamp . . . 155
5.3 Data Types in SQL . . . 156
5.4 Example . . . 158
5.5 Programs . . . 159
5.6 Metadata . . . 173
5.7 JDBC 3.0 . . . 173

6 Object-Oriented Databases **177**
6.1 Introduction . . . 177
6.2 Object-Oriented Properties . . . 181
6.3 Terms Glossary . . . 183
6.4 Properties of an Object-Oriented Database . . . 186
6.5 Example . . . 188
6.6 C++ . . . 192
6.7 The Object Query Language . . . 194

6.8 SQL3 Object Model . . . 195
6.8.1 Basic Concepts . . . 195
6.8.2 Objects . . . 197
6.8.3 Operations . . . 198
6.8.4 Methods . . . 199
6.8.5 Events . . . 201
6.8.6 Binding and Polymorphism . . . 202
6.8.7 Types and Classes . . . 203
6.8.8 Inheritance and Delegation . . . 208
6.8.9 Noteworthy Objects . . . 210
6.8.10 Extensibility . . . 212
6.9 SQL3 Datatypes and Java . . . 214
6.10 Evaluations of OODBMSs . . . 219
6.11 Summary . . . 222

7 Versant **225**
7.1 Introduction . . . 225

8 FastObjects **233**
8.1 Introduction . . . 233

9 Data Mining **235**
9.1 Introduction . . . 235
9.2 Example . . . 242

Bibliography **243**

Index **243**

Preface

This book explores the use of databases and related tools in the various applications. Both relational and object-oriented databases are coverd. An introduction to JDBC is also given. It also includes C++ and Java programs relevant in databases.

Without doubt, this book can be extended. If you have comments or suggestions, we would be pleased to have them. The email addresses of the author are:

whs@na.rau.ac.za
steeb_wh@yahoo.com
willi-hans.steeb@fhso.ch

The web sites of the author are:

http://www.fhso.ch/~steeb
http://issc.rau.ac.za

Chapter 1

What is a table?

1.1 Introduction

What is a table? As a definition for a table in the Oxford dictionary we find

```
 "orderly arrangment of facts, information etc
(usually as in columns)"
```

For a database we find the definition

```
A database is a means of storing information in such a way that
information can be retrieved from it.
```

Thus a database is typically a repository for heterogeneous but interrelated pieces of information. Often a database contains more than one table. Codebooks and dictionaries can also be considered as tables. A dictionary is a reference book on any subject, the items of which are arranged in alphabetical order. A codebook is a list of replacements for words or phrases in the original message. A code is a system for hiding the meaning of a message by replacing each word or phrase in the original message with another character or set of characters. The list of replacements is contained in a codebook. An alternative definition of a code is any form of encryption which has no built-in flexbility, i.e. there is only one key, namely the codebook.

Databases contain organized data. A database can be as simple as a flat file (a single computer file with data usually in a tabular form) containing names and telephone numbers of one's friends, or as elaborate as the worldwide reservation system of a major airline. Many of the principles discussed in this book are applicable to a wide variety of database systems.

Structurally, there are three major types of databases:

```
Hierarchical
Relational
Network
```

During the 1970s and 1980s, the hierarchical scheme was very popular. This scheme treats data as a tree-structured system with data records forming the leaves. Examples of the hierarchical implementations are schemes like b-tree and multi-tree data access. In the hierarchical scheme, to get to data, users need to traverse up and down the tree structure. XML (Extensible Markup Language) is based on a tree structure. The most common relationship in a hierarchical structure is a one-to-many relationship between the data records, and it is difficult to implement a many-to-many relationship without data redundancy.

The network data model solved this problem by assuming a multi-relationship between data elements. In contrast to the hierarchical scheme where there is a parent-child relationship, in the network scheme, there is a peer-to-peer relationship. Most of the programs developed during those days used a combination of the hierarchical and network data storage and access model.

During the 90s, the relational data access scheme came to the forefront. The relational scheme views data as rows of information. Each row contains columns of data, called fields. The main concept in the relational scheme is that the data is uniform. Each row contains the same number of columns. One such collection of rows and columns is called a table. Many such tables (which can be structurally different) form a relational database. A relational database is accessed and administered using Structured Query Language (SQL) statements. SQL is a command language for performing operations on databases and their component parts. Tables are the component parts we are dealing with most often. Their column and row structure makes them look a great deal like spreadsheets, but the resemblance is only surface-level. Table elements are not used to represent relationships to other elements - that is, table elements do not hold formulas, they just hold data. Most SQL statements are devoted to working with the data in these rows and columns, allowing the user to add, delete, sort, and relate it between tables.

There are many ways to approach the design of a database and tables. The database layout is the most important part of an information system. The more design and thought put into a database, the better the database will be in the long run. We should gather information about the user's requirement, data objects, and data definitions before creating a database layout. The first step we take when determining the database layout is to build a data model that defines how the data is to be stored. For most relational databases, we create an entity-relationship diagram or model. The steps to create this model are as follows:

1. Identify and define the data objects (entities, relationship, and attributes)

2. Diagram the objects and relationship

3. Translate the objects into relational constructs (such as tables)

4. Resolve the data model

5. Perform the normalization process

First, define the entities and the relationships between them. An *entity* is something that can be distinctively identified. An example of an entity is a

specific person, an element in the periodic table, a specific book,

etc. The relationship is the association between the entities, which is described as connectivity, cardinality, or dependency. Connectivity is the occurrence of an entity, meaning the relationship to other entities is either one-to-one, one-to-many, or many-to-many. The cardinality term places a constraint on the number of times an entity can have an association in a relationship. The entity dependency describes whether the relationship is mandatory or optional. After we identified the entities we can proceed with identifying the attributes. Attributes are all the descriptive features of the entity. When defining the attributes, we must specify the constraints (such as valid values or characters) or any other features about the entity. After we complete the process of defining the entities and the relationship of the database, the next step is to display the process we designed. There are many purposes for diagramming the data objects.

1. Organize information

2. Documents for the future and give people new to the project a basic understanding of what is going on

3. Identifies entities and relationships

4. Determines the logical design to be used for the physical layout

After the diagram is complete, the next step is to translate the data objects (entities and attributes) into relational constructs. We translate all the data objects that are defined into tables, columns and rows. Each entity is represented as a table, and each row represents an occurence of that entity.

A table is an object that holds data relating to a single entity. Table design includes the following

1. Each table is uniquely named within the database

2. Each table has one or more columns

3. Each column is uniquely named within the table

4. Each column contains one data type

5. Each table can contain zero or more rows of data.

The tables contain two types of columns: keys or descriptors. A key column uniquely defines a specific row of a table, whereas a descriptor column specifies non-uniqueness in a particular row. When we create tables, we define

primary and foreign keys.

The primary key consists of one or more columns that make each row unique. Each table should have a primary key. The foreign key is one or more columns in one table that match the columns of the primary key of another table. The purpose of a foreign key is to create a relationship between two tables so we can join tables. Primary keys play a central role in the normalization process.

The next step in our data model is to resolve our relationships. The most common types of relationships are one-to-many (1:m) and many-to-many (m:n). In some cases, a one-to-one relationship may exist between tables. In order to resolve the more complex relationships, we must analyze the relational business rules, and in some instances, we might need to add additional entities.

1.2 Examples

Let us give some examples of tables.

Example 1. A table `Person` contains their id-number, their surname, their first name, sex, and birthdate

```
id#  SurName   FirstName  Sex  Birthdate
===  ========= =========  === ============
31    Miller    John        m   20.03.1945
72    Smith     Laura       f   10.10.1980
83    Cooper    Fred        m   28.12.1967
..    ......    ....       ..   ..........
=========================================
```

The id-number could play the role of the primary key.

Example 2. The ASCII table maps characters to integers and vice versa (one-to-one map)

```
char  value
====  =====
...   ...
'0'    48
'1'    49
...   ...
'9'    57
...    ...
'A'    65
'B'    66
...    ...
'a'    97
'b'    98
...   ...
===========
```

Example 3. The memory address and its contents is a table. In most hardware design we can store one byte (8 bits) at one memory location.

```
Address  Contents
=======  ========
.......  ........
0x45AF2  01100111    <- 1 byte at each address
0x45AF3  10000100    <- 1 byte at each address
.......  ........
================
```

This means that at address `0x45AF2` (memory addresses are given in hex notation, where `0x` indicates hex notation) the contents is the bitstring `01100111`, which is 103 in decimal. Obviously the contents at a memory address can change.

Example 4. Look up table for integration

integrand	variable	integral	condition
a	x	a*x	none
a*x	x	a*x*x/2	none
exp(a*x)	x	exp(a*x)/a	a not 0
sin(a*x)	x	-cos(a*x)/a	a not 0
a/x	x	a*ln(x)	x > 0

Example 5. A table for a soccer league should include the position, the name of the teams, the number of matches, the number of matches won, draw, lost, the goals, the difference of the goals, and the points. For example

Pos	Name	matches	won	draw	lost	goals	diff	points
1	FC Lugano	22	12	6	4	33:16	17	42
2	FC St. Gallen	22	11	7	4	43:18	25	40
3	Gh Zuerich	22	11	3	8	46:25	21	36
4	Lausanne Sport	22	11	2	9	37:34	3	35
5	FC Basel	22	10	4	8	42:36	6	34
6	Servette Genf	22	9	6	7	34:26	8	33
7	FC Sion	22	9	5	8	27:31	-4	32
8	FC Zuerich	22	8	7	7	36:29	7	31
9	FC Aarau	22	6	6	10	31:43	-12	24
10	FC Yverdon	22	5	6	11	27:43	-16	21
11	Xamax Neuchatel	22	6	2	14	21:53	-32	20
12	FC Luzern	22	5	4	13	27:50	-23	19

Example 6. To represent negative integers one uses the so-called two-complement of a given bitstring. For example, assume we have 8 bits. We can list this as a table

bitstring	one-complement	two-complement
00000000	11111111	00000000
00000001	11111110	11111111
00000010	11111101	11111110
.....	.....	.....
11111110	00000001	00000010
11111111	00000000	00000001

Example 7. The most common devices in a PC (COM ports, parallel ports, and floppies) and their IRQ (Interrupt Request), DMA (Direct Memory Access), and I/O addresses are listed in tables.

Device	IRQ	DMA	I/O Address (hex)
COM 1 (/dev/ttyS0)	4	N/A	3F8
COM 2 (/dev/ttyS1)	3	N/A	2F8
COM 3 (/dev/ttyS2)	4	N/A	3E8
COM 4 (/dev/ttyS3)	3	N/A	2E8
LPT 1 (/dev/lp0)	7	N/A	378-37F
LPT 2 (/dev/lp1)	5	N/A	278-27F
Floppy A (/dev/fd0)	6	2	3F0-3F7
Floppy B (/dev/fd1)	6	2	3F0-3F7

Since only two COM ports (serial ports) are usually supported by DOS, they share IRQ values. The I/O addresses are different. Both floppy disks share the same I/O addresses, IRQ, and DMA. Network cards, SCSI adapters, sound cards, video cards, and other peripherals all must have unique IRQ, DMA, and I/O addresses, which can be difficult to arrange with a fully loaded system.

1.3 Tables in Programs

Using several programs we show how to set up a table.

Example 1. We have a table `Student`. It includes the following attributes

```
studentno, surname, firstname, subject, marks
```

The table looks like this

```
studentno   surname   firstname   subject   marks
=========   =======   =========   =======   =====
101         Muller    Jack        C++        50%
102         Smith     Milton      C++        74%
103         Muller    John        C++        82%
104         Solms     Carl        C++       100%
105         Steeb     Hans        C++       100%
================================================
```

The student number (`studentno`) can be considered as a so-called primary key. In the following C++ program we declare each column in the table as an array of strings. We provide the function

```
int lookup(char* number)
```

with the student number. The function then finds the index for this student number. The index is in the range 0..4. Using this index in the `main` function we retrieve the `surname`, `firstname`, `subject` and the `marks`. If the student number is not in the list the function `lookup` returns `-1`.

```
// student.cpp

#include <iostream>
#include <string.h>   // for strcmp

using namespace std;

char* studentno[] = { "101", "102", "103", "104", "105" };
char* surname[] = { "Muller", "Smith", "Muller", "Solms", "Steeb" };
char* firstname[] = { "Jack", "Milton", "John", "Carl", "Hans" };
char* subject[] = { "C++", "C++", "C++", "C++", "C++" };
char* marks[] = { "50%", "74%", "82%", "100%", "100%" };
```

```
int lookup(char* number)
{
   int i;
   for(i = 0; i < 5; i++)
   {
   int result = strcmp(number,studentno[i]);
   if(result == 0) return i;
   }
   return -1;
}

int main()
{
   char* number = new char[4];  // allocating memory
   cout << "enter student number: ";
   cin.getline(number,4);
   int x = lookup(number);
   delete number;

   if(x != -1)
   {
   cout << "studentno = " << studentno[x] << endl;
   cout << "surname = " << surname[x] << endl;
   cout << "firstname = " << firstname[x] << endl;
   cout << "subject = " << subject[x] << endl;
   cout << "marks = " << marks[x] << endl;
   }
   if(x == -1)
   {
   cout << "StudentNo not in table ";
   }

   return 0;
}
```

Example 2. In our second example we have an employeers id-number and the name

```
Id-Number  Name
=========  ========
101        Simpson
102        Singer
103        Carter
104        Thompson
105        Ulrich
===================
```

We use the container class `map` from the Standard Template Library (STL) in C++. The STL provides the class `map`. The first argument of `map` is called the key and the second argument is called the value. In our case the key is an `int` and the value a `string`.

```
// table.cpp

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
   map<int,string> m;

   m.insert(pair<int,string>(101,"Simpson"));
   m.insert(pair<int,string>(102,"Singer"));
   m.insert(pair<int,string>(103,"Carter"));
   m.insert(pair<int,string>(104,"Thompson"));
   m.insert(pair<int,string>(105,"Ulrich"));

   int stnumber;

   cout << "enter the students number: ";
   cin >> stnumber;

   map<int,string>::iterator p;

   p = m.find(stnumber);
   if(p != m.end())
   cout << p -> second;
   else
```

```
   cout << "not a student number";
   cout << endl;

   map<string,int> n;

   n.insert(pair<string,int>("Simpson",101));
   n.insert(pair<string,int>("Singer",102));
   n.insert(pair<string,int>("Carter",103));
   n.insert(pair<string,int>("Thompson",104));
   n.insert(pair<string,int>("Ulrich",105));

   string name;

   cout << "enter the students name: ";
   cin >> name;

   map<string,int>::iterator q;

   q = n.find(name);
   if(q != n.end())
   cout << q -> second;
   else
   cout << "not a student name";

   return 0;
}
```

If we need more than two column in the table we can use

```
map<T1,map<T2,T3> >
```

The following two programs give two examples.

```
// mapmap1.cpp

#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
   map<string,string> t;

   t["Steeb"] = "Willi";
   t["de Sousa"] = "Nela";

   map<string,map<string,string> > ct;

   ct["ten"] = t;
   ct["eleven"] = t;

   cout << ct["ten"]["Steeb"] << endl;       // => Willi
   cout << ct["ten"]["de Sousa"] << endl;    // => Nela

   cout << ct["eleven"]["Steeb"] << endl;    // => Willi
   cout << ct["eleven"]["de Sousa"] << endl; // => Nela

   return 0;
}
```

```
// mapmap2.cpp

#include <iostream>
#include <string>
#include <map>

using namespace std;

typedef map<string,string> ENTITY;
typedef map<int,ENTITY*> DB;

int main()
{
   DB db;
   ENTITY* e;

   e = new ENTITY;
   (*e)["firstname"] = "Willi";
   (*e)["surname"] = "Steeb";

   db[1] = e;

   e = new ENTITY;
   (*e)["firstname"] = "Nela";
   (*e)["surname"] = "de Sousa";

   db[2] = e;

   DB::iterator it = db.begin();
   while(it != db.end())
   {
   e = it -> second;
   cout << "firstname: " << (*e)["firstname"] << "\tsurname: "
        << (*e)["surname"] << endl;
   it++;
   }
   return 0;
}

// the output is
firstname: Willi  surname: Steeb
firstname: Nela   surname: de Sousa
```

Example 3. In our third example we consider a container class from Java. A useful container class in Java for application in databases is the `TreeMap` class. The `TreeMap` class extends `AbstractMap` to implement a sorted binary tree that supports the `Map` interface. This impementation is not synchronized. If multiple threads access a `TreeMap` concurrently and at least one of the threads modifies the `TreeMap` structurally it must be synchronized externally.

The next program shows an application of the class `TreeMap`. The default constructor `TreeMap()` constructs a new, empty `TreeMap` sorted according to the keys in natural order. The constructor

```
TreeMap(Map m)
```

constructs a new `TreeMap` containing the same mappings as the given `Map`, sorted according to the key's natural order.

The method

```
Object put(Object key,Object value)
```

associates the specified value with the specified key in this `TreeMap`. The method

```
public Set entrySet()
```

in class `TreeMap` returns a `Set` view of the mapping contained in this map. The `Set`'s `Iterator` will return the mappings in ascending `Key` order. Each element in the returned set is a `Map.Entry`. The method

```
boolean containsKey(Object key)
```

returns `true` if this `TreeMap` contains a mapping for the specified key. The method

```
boolean containsValue(Object value)
```

returns `true` if this `Map` maps one or more keys to the specified value. The method

```
Object get(Object key)
```

returns the value to which this `TreeMap` maps the specified key. The method

```
Object remove(Object key)
```

removes the mapping for this key from this `TreeMap` if present.

```
// MyMap.java

import java.util.*;

public class MyMap
{
    public static void main(String[] args)
    {
    String[] sc = new String[3];
    sc[0] = new String("A");
    sc[1] = new String("B");
    sc[2] = new String("C");

    String[] sn = new String[3];
    sn[0] = new String("65");
    sn[1] = new String("66");
    sn[2] = new String("67");

    TreeMap map = new TreeMap();

    int i;
    for(i=0; i < sc.length; i++)
    {
    map.put(sc[i],sn[i]);
    }

    displayMap(map);
    }  // end main

    static void displayMap(TreeMap map)
    {
    System.out.println("The size of the map is: " + map.size());
    Collection c = map.entrySet();
    Iterator it = c.iterator();

    while(it.hasNext())
    {
    Object o = it.next();
    if(o == null)
    System.out.println("null");
    else
    System.out.println(o.toString());
    }
    } // end method displayMap
}
```

Example 4. Hashtables are *associative arrays* with key-value pairings. Presentation of the key retrieves the value. Both the key and the value must be objects, i.e. primitive values must be represented by their wrapper classes, e.g. `Integer` for an `int` value.

The methods

```
put(Object key,Object value)
get(Object key)
remove(Object key)
```

are the methods for entering new pairs, retrieving and removing them. We can test for the presence of a particular key with the method `containsKey()`. Hashtables rely on the

```
int hashCode()
```

method derived from the class `Object`. It overrides `hashCode()` in class `Object`. The return value of the `hashCode()` method is a unique numerical value derived from the object contents. The method `hashCode()` can be overriden.

```
// Hash.java

import java.util.*;

public class Hash
{
   public static void main(String[] args)
   {
   Hashtable dates = new Hashtable();
   dates.put("Birthday Willi",new String("20 March"));
   dates.put("Birthday Jan",new String("10 October"));
   dates.put("Birthday Mia",new String("8 April"));
   dates.put("Birthday Moritz",new String("16 July"));

   String jan = (String) dates.get("Birthday Jan");
   System.out.println("Jan = " + jan);

   boolean b = dates.containsKey("Birthday Moritz");
   System.out.println("b = " + b);
   dates.remove("Birthday Moritz");
   b = dates.containsKey("Birthday Moritz");
   System.out.println("b = " + b);
   }
}
```

Example 5. In C progamming the tables are implement as structures. In the following example we have two structures, one for the table `Name` and one for the table `Employee`. The table `Name` is used inside (nested) the table `Employee`.

```
// Employee.cpp

#include <iostream.h>
#include <string.h>

struct Name
{
   char firstname[20];
   char surname[20];
};

struct Employee
{
   struct Name name[2];
   int empno, salary;
};

int main()
{
   struct Name n[2];
   strcpy(n[0].firstname,"Willi");
   strcpy(n[0].surname,"Steeb");
   strcpy(n[1].firstname,"Lara");
   strcpy(n[1].surname,"Smith");

   struct Employee emp[2];
   emp[0].empno = 123;
   emp[0].salary = 1240;
   strcpy(emp[0].name[0].firstname,"Willi");
   strcpy(emp[0].name[0].surname,"Steeb");

   emp[1].empno = 134;
   emp[1].salary = 2340;
   strcpy(emp[1].name[1].firstname,"Lara");
   strcpy(emp[1].name[1].surname,"Smith");

   cout << emp[1].name[1].firstname << " "
        << emp[1].name[1].surname << " "
        << emp[1].empno << " " << emp[1].salary;
   return 0;
}
```

Example 6. The table is normally stored as a file, binary or text. In the following sixth example the table is given by

```
Name   idnumber   fee
=====  ========   ======
Willi  67890123   -45.66
Hans   12345678   -56.00
Nela   89045677   -13.45
Helen  89734444   -97.11
Audrey 99999999   -1.45
========================
```

Thus the table includes the name, the identity number and the outstanding fee. The table is stored as the text file `mydata.txt`

```
Willi  67890123 -45.66
Hans   12345678 -56.00
Nela   89045677 -13.45
Helen  89734444 -97.11
Audrey 99999999 -1.45
```

In our Java program we read the file `mydata.txt` and display the names and identity number und calculate the sum of the outstanding fees. We read each line using the method

```
String readLine()
```

in the class `BufferedReader` which reads a line of text. A line is considered to be terminated by any one of a line feed `'\n'` a carriage return `'\r'` or a carriage return followed immediately by a line feed. A `String` is returned containing the contents of the line, not including any line termination characters, or null if the end of the stream has been reached. a `String`.

Using the class `StringTokenizer` we cast the `String` into a `String` for the name, an `int` for the id-number and a `double` for the fee.

```
// DataBase.java

import java.awt.*;
import java.io.*;
import java.util.*;

public class DataBase
{
   public static void main(String args[]) throws IOException
   {
   int[] idnumber = new int[5];
   String[] names = new String[5];
   int i = 0;
   String str;
   double sum = 0.0;

   FileInputStream fin = new FileInputStream("mydata.txt");
   BufferedReader in = new BufferedReader(new InputStreamReader(fin));

   while(!(null == (str = in.readLine())))
   {
   StringTokenizer tok = new StringTokenizer(str);
   String s1 = tok.nextToken();
   names[i] = s1;

   String s2 = tok.nextToken();
   idnumber[i] = new Integer(s2).intValue();

   String s3 = tok.nextToken();
   double temp = new Double(s3).doubleValue();
   sum += temp;

   i++;
   } // end while

   in.close();

   for(i=0; i < 5; i++)
   {
   System.out.println("The name is: " + names[i] + " " +
                      "The idnumber is: " + idnumber[i]);
   }
   System.out.println("The sum is: " + sum);
   } // end main
}
```

Example 7. In a more advanced case we have a phone book with the phone number and the name. We want to insert names and phone numbers in the table, delete rows, write to the file, exit the manipulation of the database. We have the following commands:

`?name` find phone number
`/name` delete row with the given name
`!number name` insert or update a row
`*` list whole phonebook
`=` save to file (commit changes)
`#` exit phonebook (database)

For example, the command

```
!34567 Cooper_Jack
```

inserts a row into the phonebook with the phone number `34567` and the name `Cooper_Jack`. To save it to the file `phone.txt` we have to apply the command `=`. In our C++ program we use the Standard Template Library (STL). Here `less<T>` is a function object that tests the truth or falsehood of some condition. If `f` is an object of class `less<T>` and `x` and `y` are objects of class `T`, then `f(x,y)` returns true if `x < y` and false otherwise.

```
// phone.cpp

#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <map>
using namespace std;

typedef map<string,long,less<string> > directype;

void ReadInput(directype& D)
{
   ifstream ifstr;
   ifstr.open("phone.txt");
   long nr;
   string str;
   if(ifstr)
   {
   cout << "Entries read from file phone.txt:\n";
   for(;;)
   {
```

```
   ifstr >> nr;
   ifstr.get();   // skip space
   getline(ifstr,str);
   if(!ifstr) break;
   cout << setw(9) << nr << " " << str << endl;
   D[str] = nr;
   }
   }
   ifstr.close();
}

void ShowCommands()
{
   cout <<
        "Commands: ?name       : find phone number\n"
        "          /name       : delete\n"
        "          !number name: insert (or update)\n"
        "          *           : list whole phonebook\n"
        "          =           : save in file\n"
        "          #           : exit" << endl;
}

void ProcessCommands(directype& D)
{
   ofstream ofstr;
   long nr;
   string str;
   char ch;
   directype::iterator i;
   for(;;)
   {
   cin >> ch; // skip any whitespace and read ch
   switch(ch)
   {
   case'?': case '/':
   getline(cin,str);
   i = D.find(str);
       if(i == D.end()) cout << "Not found.\n";
       else
        if(ch == '?') cout << "Number: " << (*i).second << endl;
        else
          D.erase(i);
        break;
       case '!':
         cin >> nr;
```

```
          if(cin.fail())
          {
          cout << "Usage: !number name\n";
          cin.clear();
          getline(cin,str);
          break;
     }
     cin.get();
     getline(cin,str);
     D[str] = nr;
     break;
       case '*':
         for(i = D.begin(); i != D.end(); i++)
           cout << setw(9) << (*i).second << " "
                << (*i).first << endl;
           break;
       case '=':
         ofstr.open("phone.txt");
         if(ofstr)
         {
         for(i = D.begin(); i != D.end(); i++)
           ofstr << setw(9) << (*i).second << " "
                 << (*i).first << endl;
           ofstr.close();
 }
 else cout << "Cannot open output file.\n";
 break;
 case '#': break;
 default: cout << "Use: * (list), ? (find), = (save), "
          "/ (delete), ! (insert), or # (exit), \n";
 getline(cin,str);
 break;
     }
     if(ch == '#') break;
     }
}

int main()
{
   directype D;
   ReadInput(D);
   ShowCommands();
   ProcessCommands(D);
   return 0;
}
```

Example 8. For Microprocessors we also use tables in particular lookup tables. As an example we consider the PIC 16F84 Microprocessor. We store a lookup table in the program memory. To access data in program memory, a table read operation must be performed. The table consists of a series of

```
RETLW K
```

statements. The command `RETLW` returns with literal in `W` (`W` is the working register or accumulator for the PIC16F84). The 8-bit table constants are assigned to the literal `K`. The first instruction in the table

```
ADDWF PCL
```

computes the offset (counting from 0) to the table and consequently the program branches to a appropiate `RETLW K` instruction. The table contains the characters

```
'A' ASCII table 65 dec 41 hex 01000001binary
'B' ASCII table 66 dec 42 hex 01000010binary
'C' ASCII table 67 dec 43 hex 01000011binary
'D' ASCII table 68 dec 44 hex 01000100binary
```

Since we move 3 into `W` using the command

```
MOVLW 3
```

and then add it to `PCL` (Program counter low) we select the character `D` which is ASCII 68 decimal and in binary 01000100. This bit string is moved to `PORTB` (output) and displayed.

The program counter `PC` in the PIC16F84 is 13-bits wide. The low 8-bits (`PCL`) are mapped in SRAM (static RAM) at location `02h` and are directly readable and writeable. Let `k` be a label. Then

```
CALL k
```

calls a subroutine, where `PC + 1 -> TOS` (top of stack) and `k -> PC`.

The upper byte of the program counter is not directly accessible. `PCLATH` is a slave register for `PC<12:8>`. The contents of `PCLATH` can be transferred to the upper byte of the program counter, but the contents of `PC<12:8>` is never transferred to `PCLATH`.

```
     PROCESSOR 16f84
     INCLUDE "p16f84.inc"

     ORG H'00'

Start
     BSF STATUS, RP0
     MOVLW B'11111111'
     MOVWF PORTA
     MOVLW B'00000000'
     MOVWF PORTB
     BCF STATUS, RP0

     MOVLW 3
     CALL Table

     MOVWF PORTB

Table:
     ADDWF PCL

     RETLW 'A'
     RETLW 'B'
     RETLW 'C'
     RETLW 'D'
;
     END
```

Example 9. For Web application of databases JavaScript is a good choice, since the program is embedded in the HTML file. Using a `FORM` the user enters the surname of the person and is provided with the complete information about the person (first name, street, and phone number).

```
<HTML>
<HEAD>
<TITLE>Names Database</TITLE>
<SCRIPT LANGUAGE="JavaScript">

Names = new Object()
Names[0]=10
Names[1]="cooper"
Names[2]="smith"
Names[3]="jones"
Names[4]="michaels"
Names[5]="avery"
Names[6]="baldwin"

Data = new Object()
Data[1]="Olli|Cooper|44 Porto Street|666-000"
Data[2]="John|Smith|123 Main Street|555-1111"
Data[3]="Fred|Jones|PO Box 5|555-2222"
Data[4]="Gabby|Michaels|555 Maplewood|555-3333"
Data[5]="Alice|Avery|1006 Pike Place|555-4444"
Data[6]="Steven|Baldwin|5 Covey Ave|555=5555"

function checkDatabase()
{
   var Found = false;  // local variable
   var Item = document.testform.customer.value.toLowerCase();
   for(Count = 1; Count <= Names[0]; Count++)
   {
   if(Item == Names[Count])
   {
   Found = true;
   var Ret = parser(Data[Count], "|");
   var Temp = "";
   for(i = 1; i <= Ret[0]; i++)
   {
   Temp += Ret[i] + "\n";
   }
   alert(Temp);
   break;
   }
```

```
   }
   if(!Found)
   alert("Sorry, the name '" + Item +"' is not listed in the database.")
}  // end function checkDatabase()

function parser(InString,Sep)
{
   NumSeps=1;
   for(Count=1; Count < InString.length; Count++)
   {
   if(InString.charAt(Count)==Sep)
   NumSeps++;
   }
   parse = new Object();
   Start=0; Count=1; ParseMark=0;
   LoopCtrl=1;
   while(LoopCtrl==1)
   {
   ParseMark = InString.indexOf(Sep,ParseMark);
   TestMark = ParseMark+0;
   if((TestMark==0) || (TestMark==-1))
   {
   parse[Count]= InString.substring(Start,InString.length);
   LoopCtrl=0;
   break;
   }
   parse[Count] = InString.substring(Start,ParseMark);
   Start=ParseMark+1;
   ParseMark=Start;
   Count++;
   }
   parse[0]=Count;
   return (parse);
}  // end function parser
</SCRIPT>
<FORM NAME="testform" onSubmit="checkDatabase()">
Enter the customer's name, then click the "Find" button:
<BR>
<INPUT TYPE="text" NAME="customer" VALUE="" onClick=0> <P>
<INPUT TYPE="button" NAME="button" VALUE="Find"
onClick="checkDatabase()">
</FORM>
</BODY>
</HTML>
```

Example 10. The class `JTable` in Java provides a very flexible capability for creating and displaying tables. The `JTable` class is another Swing component that does not have an AWT analog. The `JTable` class is used to display and edit regular two-dimensional tables of cells. It allows tables to be constructed from arrays, vectors of objects, or from objects that implement the `TableModel` interface. The `DefaultTableModel` is a model implementation that uses a `Vector` of `Vectors` of Objects to store the cell values.

The following program gives an example. The table is given by

```
First Name  Last Name  Sport         # of Years  Vegetarian
==========  =========  ============  ==========  ==========
Mary        Lea        Snowboarding   5          no
Alison      Humi       Rowing         3          yes
Kathy       Wally      Tennis         2          no
Mark        Andrews    Boxing        10          no
Angela      Lih        Running        5          yes
=============================================================
```

```
// MyTable.java

import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.JOptionPane;
import java.awt.*;
import java.awt.event.*;

public class MyTable extends JFrame
{
   private boolean DEBUG = true;

   public MyTable()
   {
   super("MyTable");

   MyTableModel myModel = new MyTableModel();
   JTable table = new JTable(myModel);
   table.setPreferredScrollableViewportSize(new Dimension(400,70));

   JScrollPane scrollPane = new JScrollPane(table);
```

```
getContentPane().add(scrollPane,BorderLayout.CENTER);

addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});

} // end MyTable

class MyTableModel extends AbstractTableModel
{
final String[] columnNames =
{ "First Name", "Last Name", "Sport", "# of Years", "Vegetarian" };

final Object[][] data =
{
{ "Mary", "Lea", "Snowboarding", new Integer(5), new Boolean(false) },
{ "Alison", "Humi", "Rowing", new Integer(3), new Boolean(true) },
{ "Kathy", "Wally", "Tennis", new Integer(2), new Boolean(false) },
{ "Mark", "Andrews", "Boxing", new Integer(10), new Boolean(false) },
{ "Angela", "Lih", "Running", new Integer(5), new Boolean(true) }
};

public int getColumnCount()
{ return columnNames.length; }

public int getRowCount()
{ return data.length; }

public String getColumnName(int col)
{ return columnNames[col]; }

public Object getValueAt(int row,int col)
{ return data[row][col]; }

public Class getColumnClass(int c)
{ return getValueAt(0,c).getClass(); }

public boolean isCellEditable(int row,int col)
{
if(col < 2) { return false; }
else
```

```
{ return true; }
}

public void setValueAt(Object value,int row,int col)
{
if(DEBUG)
{
System.out.println("Setting value at " + row + "," + col
                                        + " to " + value
                                        + " (an instance of "
                                        + value.getClass() + ")");
}

if(data[0][col] instanceof Integer)
{
try
{
data[row][col] = new Integer((String) value);
fireTableCellUpdated(row,col);
}
catch(NumberFormatException e)
{
JOptionPane.showMessageDialog(MyTable.this,
      "The \"" + getColumnName(col)
      + "\" column accepts only integer values.");
}
}
else
{
data[row][col] = value;
fireTableCellUpdated(row,col);
}

if(DEBUG)
{
System.out.println("New value of data:");
printDebugData();
}

}

private void printDebugData()
{
int numRows = getRowCount();
int numCols = getColumnCount();
```

```
   for(int i=0; i < numRows; i++)
   {
   System.out.print("  row " + i + ".");
   for(int j=0; j < numCols; j++)
   {
   System.out.print("  " + data[i][j]);
   }
   System.out.println();
   }
   System.out.println("--------------------------------");
   }
   }

   public static void main(String[] args)
   {
   MyTable frame = new MyTable();
   frame.pack();
   frame.setVisible(true);
   }
}
```

Example 11. An XML document is a database only in the strictest sense of the term. That is, it is a collection of data. In many ways, this makes it no different from any other file. All files contain data of some sort. As a database format, XML has some advantages. For example, it is self-describing (the markup describes the data), it is portable (Unicode), and it describes data in tree format. Every well-formed XML document is a tree.

A tree is a data structure composed of connected nodes beginning with a top node called the *root*. The root is connected to its child nodes, each of which is connected to zero or more children of its own, and so forth. Nodes that have no children of their own are called *leaves*. A diagram of a tree looks much like a genealogical descendant chart that lists the descendants of a single ancestor. The most useful property of a tree is that each node and its children also form a tree. Thus, a tree is a hierarchical structure of trees in which each tree is built out of samller trees.

Fo the purpose of XSLT, elements, attributes, namespaces, processing instructions, and comments are counted as nodes. Furthermore, the root of the document must be distinguished from the root element. Thus, XSLT processors model an XML document as a tree that contains seven kinds of nodes:

1) the root

2) elements

3) text

4) attributes

5) namespaces

6) processing instructions

7) comments

An example for the periodic table is given below (we only give the first two elements of the periodic table)

```
<?xml version="1.0"?>
<!-- Periodic_Table.xml -->

<periodic_table>
 <atom>
   <name> hydrogen </name>
   <symbol> H </symbol>
   <atomic_number> 1 </atomic_number>
   <atomic_weight> 1.00794 </atomic_weight>
   <atom_phase> gas </atom_phase>
 </atom>

 <atom>
   <name> helium </name>
   <symbol> He </symbol>
   <atomic_number> 2 </atomic_number>
   <atomic_weight> 4.0026 </atomic_weight>
   <atom_phase> gas </atom_phase>
 </atom>

</periodic_table>
```

1.4 Table and Relation

The table is isomorphic to the mathematical relation, which puts the relational model of data onto firm theoretical foundations that allow the development of theorems and proofs.

Consider two finite sets A and B. For example

$$A := \{ a, b\}, \qquad B := \{ u, v, w, x \} .$$

The *Cartesian product* of the sets A and B is

$$A \times B := \{ (a,u), (a,v), (a,w), (a,x), (b,u), (b,v), (b,w), (b,x)\} .$$

Thus $A \times B$ contains $2 \cdot 4 = 8$ elements. A relation R is a subset of the Cartesian product of the sets on which it is defined. For example

$$R = \{ (a,u), (a,x), (b,u), (b,v) \} .$$

Here R is a subset of $A \times B$, which is another way of saying that R is a set of couples (2-tuples) with the first element taken from the set A and the second element taken from the set B. As table we have

```
R
==  ==
a   u
a   x
b   u
b   v
======
```

The relational algebra (and any equivalent languges) is closed: algebraic operators take relations as operands and return relations as result, allowing the nesting of expressions to arbitrary depths.

The Cartesian product can be extended to $S_1 \times S_2 \times \ldots \times S_n$ of n sets S_1, S_2, $\ldots S_n$ is the set of all ordered n-tuples $(x_1, x_2, \ldots, x_n)$ in which $x_1 \in S_1$, $x_2 \in S_2$, $\ldots x_n \in S_n$.

Since a relation is a set of tuples, we can apply the set theoretical operators

```
UNION
INTERSECTION
MINUS
TIMES (Cartesian product)
```

However, note that the operators `UNION`, `INTERSECTION` and `MINUS` can only be applied to pairs of relations that share the same attributes.

Let R be a relational scheme with a set of attributes X, where $\{A_1, \ldots, A_k\} \subseteq X$. The projection of R onto $\{A_1, \ldots, A_k\}$ is expressed by

```
SELECT DISTINCT A1, ... , Ak FROM R
```

Let R be a relational scheme with a set of attributes X, where $A, B \in X$. The selection of R with respect to condition $A = a$ is expressed by

```
SELECT DISTINCT * FROM R WHERE A = a
```

Analogously, the selection with respect to condition $A = B$ is expressed by

```
SELECT DISTINCT * FROM R WHERE A = B
```

Let R and S be relational schemata with equal sets of attributes. The union of R and S is expressed by

```
SELECT DISTINCT * FROM R
UNION SELECT DISTINCT * FROM S
```

Analogously, the difference between R and S is expressed by

```
SELECT DISTINCT * FROM R
EXCEPT SELECT DISTINCT * FROM S
```

Let R be a relational schema with attributes $A_1, \ldots, A_n, \quad B_1, \ldots, B_m$ and S a relational schema with attributes $B_1, \ldots, B_m, \quad C_1, \ldots, C_l$. Then the natural join of R and S, which joins those tuples from R and S which have equal B-values, is expressed by

```
SELECT DISTINCT A1, ... , Am, R.B1, ... , R.Bm, C1, ... , Cl
FROM R, S
WHERE R.B1 = S.B1 AND ... AND R.Bm = S.Bm
```

Since attributes from different relational schemata may have identical names, the dot-notation `R.B` is used to specify which occurrence of the respective attribute is meant. With respect to the natural join, of course, we could have used `S.B` as well. The `WHERE` clause then explicitly states that tuples from the different relations must have identical values with respect to the shared attributes. A simpler formulation of the natural join, which is possible in SQL2, is

```
SELECT * FROM R NATURAL JOIN S
```

Functional dependency is defined as follows: Consider a relation R that has two attributes A and B. The attribute B of the relation is functionally dependent on the attribute A if and only if for each value of A no more than one value of B is associated.

Chapter 2

Structured Query Language

2.1 Introduction

A database is a means of storing information in such a way that information can be retrievd from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation, which explains the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used here is a general term that includes RDBMS.

Structured Query Language (SQL) is a relational database language. Amongst other things the language consists of

select, insert, update, delete, query and protect data.

SQL allows users to access data in relational database mangement systems such as mySQL, Oracle, Sybase, Informix, DB2, Microsoft SQL Server, Access and others, by allowing users to describe the data the user wishes to see. Additionally SQL also allows users to define the data in a database and manipulate that data, for example update the data. SQL is the most widely used relational database language. Others are QBE and QUEL.

SQL is a nonprocedural language. We can use SQL to tell what data to retrieve or modify without telling it how to do its job. SQL does not provide any flow-of-control programming constructs, function definitions, do-loops, or if-then-else statements. However, for example Oracle provides procedural language extensions to SQL through a product called PL/SQL. ABAP/4 contains a subset of SQL (called Open SQL) that is an integral part of the language.

SQL provides a fixed set of datatypes in particular for strings of different length

`char(n), varchar(n), longvarchar(n)`

We cannot define new datatypes. Unlike object-oriented programming language that allows to define a new datatype for a specific purpose, SQL forces us to choose from a given set of predefined datatypes when we create or modify a column.

At the highest level, SQL statements can be broadly categorized as follows into three types:

Data Manipulation Language (DML), which retrieves or modifies data

Data Definition Language (DDL), which defines the structure of the data

Data Control Language (DCL), which defines the privileges granted to database users.

The category of DML contains four basic statements:

`SELECT`, which retrieves rows from a table. The `SELECT` statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are `SELECT` statements.

`INSERT`, which adds rows to a table. `INSERT` is used to populate a newly-created table or to add a new row (or rows) to an already-existing table.

`UPDATE`, which modifies existing rows in a table. In other words it changes an existing value in a column of a table.

`DELETE`, which removes a specified row or a set of rows from a table.

These statements are used most often by application developers. DDL and DCL statements are commonly used by a database designer and database administrator for establishing the database structures used by an application.

The most common DDL commands are:

`CREATE TABLE`, creates a table with the column names the user provides. The user also needs to specify the data type for each column. Unfortunately, data types vary slighly from one RDBMS to another, so that user might need metadata to establish the data types used for a particular database. The command `CREATE TABLE` is normally used less often than the data manipulation commands because a table is created only once, whereas inserting and deleting rows or changing individual values generally occurs more frequently.

`DROP TABLE`, deletes all rows and removes the table definition from the database.

`ALTER TABLE`, adds or removes a column from a table. This command is used in connection with `ADD`, `MODIFY` and `DROP`.

For most systems (for example mySQL, Oracle), every SQL statement is terminated by a semicolon. An SQL statement can be entered on one line or split accross several lines for clarity. Most of the examples given here are split into readable portions.

For most systems SQL is not case sensitive. We can mix uppercase and lowercase when referencing SQL keywords (such as `SELECT` and `INSERT`), tables names, and column names. However, case does matter when referring to the contents of a column.

Before we can create a table we have to create a database. For example in mySQL we have the command

```
CREATE DATABASE [IF NOT EXISTS] db_name;
```

`CREATE DATABASE` creates a database with the given name. An error occurs if the database already exists and we did not specify `IF NOT EXISTS`. Since there are no tables in a database when it is initially created, the `CREATE DATABASE` statement only creates a directory under the mySQL data directory.

2.2 Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible.

First, the rows in a relational table should all be distinct. If, there are duplicates rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicates an existing row.

A second integrity rule is that column values cannot be repeating groups or arrays.

A third aspect of data integrity involves the concept of a `null` value. A database has to take care of situations where data may not be available: a null value indicates that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a *primary key*. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

Thus some of an entity's attributes uniquely identify each row in that entity. This set of attributes is called the primary key.

From a managerial design perspective, it is a good idea to assume that the RDBMS in use does not enforce *referential integrity*. Further, it is a good idea to assume that the programmer/analyst with whom we are working is not necessarily going to design the system to enforce referential integrity. It is our responsibility to define procedures that enforce referential integrity.

Example. We must specifically state in our system design that the system cannot permit the enrollment of a student into a specific course until both the student information and the course information have been previously established in their appropriate files. An error message should be displayed if an attempt to enroll a non-existent student into a non-existent course.

2.3 SQL Commands

2.3.1 Introduction

SQL (Structered Query Language) is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the `SELECT` statement. In this section we list the SQL commands and give a number of examples. In the following we assume the following tables are given. The name of the tables are `Employees` and `Cars`.

table: `Employees`

Employee_No	First_Name	Last_Name	Date_of_Birth	Car_Number
10001	John	Smith	28-AUG-1943	5
10083	Axel	Sharma	24-DEC-1954	null
10120	Jonas	Goldberg	01-JAN-1956	null
10005	Florence	Wojokowski	04-JUL-1971	12
10099	Sean	Smith	21-SEP-1966	null
10035	Liz	Yamaguchi	24-DEC-1967	null

In this table of employees, there are five columns: `Employee_No`, `First_Name`, `Last_Name`, `Date_of_Birth`, and `Car_Number`. There are six rows, each representing a different employee. The primary key for this table would generally be the employee number because each one is guaranteed to be different. A number is more efficient than a string for making comparisons. It would also be possible to use `First_Name` and `Last_Name` together because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of `Smith`. In the particular case the first name are all different, so one would conceivably use that column as a primary key, but when we add new names to the table the same first name could appear twice. If we would use the first name as the primary key and add a new employee with the name `Sean Connery` to the table the RDMS will not allow his name to be added.

table: `Cars`

Car_Number	Make	Model	Year	Price
5	BMW	Z3	1999	62000
12	Volkswagen	Lupo	2000	30000

2.3.2 Aggregate Function

SQL system allow five aggregate functions, `SUM`, `AVG`, `MAX`, `MIN`, and `COUNT`. They are called aggregate functions because they summarize the result of a query.

`SUM()` gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.

`AVG()` gives the average (arithmetic mean) of the given column (does floating point division)

`MAX()` gives the largest number in the selected column

`MIN()` gives the smallest number in the selected column

`COUNT(*)` gives the number of rows satisfying the conditions

2.3.3 Arithmetic Operators

The arithmetic operators used in SQL are similar to those in C.

```
Description            Operator
===========            ========
Addition                 +
Subtraction              -
Multiplication           *
Division                 /
```

There are six relational operators in SQL.

```
Less than                <
Less than or equal to    <=
Greater than             >
Greater than or equal to >=
Equal to                 =
Not equal to             !=
```

2.3.4 Logical Operators

The logical operators are `NOT`, `AND` and `OR`. The `AND` operator joins two or more conditions, but only returns a row if all of the conditions listed hold true. The `OR` operator joins two or more conditions. It returns a row if any of the conditions listed hold true.

2.3.5 SELECT Statement

The `SELECT` statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection.

Example 1. We want to get all columns and rows from the table `Cars`.

```
SELECT Car_Number, Make, Model, Year, Price
FROM Cars;
```

This displays the whole table. To get all columns and rows of a table without typing all columns we can use `*`. Thus

```
SELECT *
FROM Cars;
```

The `WHERE` clause (conditional selection) is used to specify that only certain rows of the table are displayed, based on the criteria described in that `WHERE` clause.

Example 2. We select from table `Employee` the `Last_Name` and `Car_Number` when the `First_Name` is `Sean`

```
SELECT Last_Name, Car_Number
FROM Employees
WHERE First_Name = 'Sean';
```

The result set (output) is

```
Last_Name   Car_Number
---------   ----------
Smith       null
----------------------
```

Example 3. The command

```
SELECT Make
FROM Cars
WHERE Price > 60000;
```

gives the output

```
Make
----
BMW
----
```

Example 4. The command

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL;
```

gives the output

```
First_Name   Last_Name
----------   ---------
John         Smith
Florence     Wojokowski
-----------------------
```

Example 5. An application of the arithmetic operation `+` is as follows

```
SELECT Model, Price + 5000
FROM Cars;
```

The output is

```
Model  Price + 5000
-----  ------------
Z3     67000
Lupo   35000
-------------------
```

Example 6. An application of the logical `AND` is

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10010 AND Car_Number IS NULL;
```

The result set is empty.

The wildcards are

```
%  zero or more characters
```

and

```
_  only one character
```

Example 7. Consider the command

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Smit%';
```

Then `Smit` and `Smith` match, but also `Smithling` etc. This means `%` stands for plus zero or more additional characters. The result set is

```
First_Name   Last_Name
----------   ---------
John         Smith
Sean         Smith
---------------------
```

Example 8. Consider the command

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'S_ith';
```

The other wildcard used in `LIKE` clauses is an underbar `_`, which stands for any one character. It matches `Smith`, but also `Snith`, `Skith` etc.
The result set is

```
First_Name   Last_Name
----------   ---------
John         Smith
Sean         Smith
----------------------
```

Example 9. The query

```
SELECT COUNT(*)
FROM Employees
WHERE Car_Number IS NULL;
```

counts the number of employees which have no car.

Example 10. The `IN` operator has to be understood in the set theoretical sense. For example

```
SELECT First_Name
FROM Employees
WHERE Last_Name IN ('Smith', 'Sharma');
```

The `NOT IN` operator list all rows excluded from the `IN` list.

Example 11. Using the `BETWEEN` operator we can give a numerical range. For example

```
SELECT Model
FROM Cars
WHERE Price NOT BETWEEN 30000 AND 50000;
```

Example 12. To sort data we use the `ORDER BY` clause. It sorts rows on the basis of columns. The `ORDER BY` clause is optional. By default the database system orders the rows in ascending order.

```
SELECT Last_Name, First_Name, Date_of_Birth
FROM Employees
ORDER BY Last_Name, First_Name;
```

To order the rows in descending order, we must add the keyword `DESC` after the column name.

A `SELECT` statement can be nested inside of another `SELECT` statement which is nested of another `SELECT` and so on. When a `SELECT` is nested it is refered to as a *subquery clause*. An example is

```
SELECT first_name AS name,
       last_name AS surname
FROM Employees
WHERE car_number IN
(SELECT DISTINCT car_number FROM
 cars WHERE car_number > 4);
```

The `DISTINCT` predicate is used to omit duplicate values just in a field. For example, if we have the last name `Smith` repeated numerous times in the table `Employees` then the code

```
SELECT DISTINCT Last_Name
FROM Employees;
```

returns only one `Smith`.

2.3.6 INSERT Command

The `INSERT` command adds rows to a table. We supply literal values or expressions to be stored as rows in the table. The term `INSERT` leads some new SQL users to think that they can control where a row is inserted in a table. Recall that a large reason for the use of relational databases is the logical data independence they offer - in other words, a table has no implied ordering. A newly inserted row simply goes into a table at an arbitrary location.

The `INSERT` commands takes the form

```
INSERT INTO table_name
[(column_name[,column_name]...[,column_name])]
VALUES
(column_value[,column_value]...[,column_value])
```

where

```
table_name    table in which to insert the row
column_name   column belonging to table_name
column_value  literal value or an expression
              whose type matches the corresponding column_name
```

Notice that the number of columns in the list of column names must match the number of literal values or expressions that appear in the parentheses after the keyword `VALUES`. If it does not match the database system returns an error message. The column and value datatype must match. Inserting an alphanumeric string into a numeric column, for example, does not make any sense. Each column value supplied in an `INSERT` command must be one of the following:

1) A `null`, 2) A literal value, such as `3` or `'Swiss'`.

An expression containing operators and functions, such as `SUBSTR(Last_Name,1,4)`. We notice that the column list is an optional element. If we do not specify the column names, the database system uses all the columns. In addition, the column order that the database system uses is the order in which the columns were specified when the table was created.

Example. As an example we insert a new row into the table `Cars`.

```
INSERT INTO Cars
(Car_Number, Make, Model, Year, Price)
VALUES
(15, 'AUDI', 'A6', 2001, 46000);
```

2.3.7 DELETE Command

The `DELETE` statement removes rows from a table. We do not need to know the physical ordering of the rows in a table to perform a `DELETE`. The database system uses the criteria in the `WHERE` clause to determine which rows to delete. The database engine determines the internal location of the rows.

The `DELETE` commamd has the simplest syntax of the four DML statements

```
DELETE FROM table_name
[WHERE condition]
```

The variables are defined as follows

```
table_name  is the table to be updated
condition   is a valid SQL condition
```

Example 1. In the table `Cars` we delete one row with the `Make BMW`.

```
DELETE FROM Cars
WHERE Make = 'BMW';
```

Example 2.

```
DELETE FROM Employees
WHERE Car_Number IS NULL;
```

If we want to delete all rows the `DELETE` statement is quite inefficient. The

```
TRUNCATE TABLE table_name
```

deletes rows much faster than the `DELETE` command does. Notice that a

```
TRUNCATE TABLE
```

statement is not a DML statement. Therefore, if we issue a `TRUNCATE TABLE` statement, we cannot change our mind and perform a rollback to recover the lost rows.

2.3.8 UPDATE Command

If we want to modify existing data in an SQL database, we need to use the `UPDATE` command. With this statement, we can update zero or more rows in a table.

The `UPDATE` statement has the following syntax.

```
UPDATE table_name
SET column_name = expression [, column = expression] ...
                             [, column = expression]
[WHERE condition]
```

The variables are defined as follows.

```
table_name   table to be updated
column_name  column in the table being updated
expression   valid SQL expression
condition    valid SQL condition
```

Example 1.

```
UPDATE Cars
SET Price = 70000
WHERE Make = 'BMW';
```

Example 2.

```
UPDATE Cars
SET Car_Number = 7
WHERE Car_Number = 5;
```

2.3.9 CREATE TABLE Command

To generate a table we use the `CREATE TABLE` statement. Each table must have a unique name. A table name cannot be an SQL reserved word. A table name should be descriptive. Within a single table, a column name must be unique. A column name cannot be an SQL reserved word. We also have to provide the datatype for each column. Furthermore, we can supply the primary key constraints and the `NOT NULL` constraint.

The `CREATE TABLE` syntax is

```
CREATE TABLE table_name  (
column_name1 data_type [NOT NULL],
.......
column_nameN datatype [NOT NULL],
[Constraint constraint_name]
[Primary key (column_nameA, column_nameB, ... column_nameN)]);
```

The variables are defined as follows

```
table_name   name for the table
column_name1 through column_nameN  valid colun names
datatype     valid datatype specification
constraint_name  optional name that identifies the
                 primary key constraint
column_nameA through column_nameN  table's columns that compose
                                   the primary key
```

Constraints include the following possibilities: `NULL` or `NOT NULL`, `UNIQUE` enforces that no two rows will have the same values for this column, `PRIMARY KEY` tells the database that this column is the primary key.

Example 1. Oracle style (for mySQL replace `NUMBER(8)` by `INT` for Integers (4 bytes)).

```
CREATE TABLE Square
(x  NUMBER(6),
 x2 NUMBER(6));
```

Example 2. Oracle style (for mySQL replace `VARCHAR2(10)` by `VARCHAR(10)` and `NUMBER(5)` by `INT`).

```
CREATE TABLE Employees
      (EMPLOYEE_NUMBER NUMBER(5) NOT NULL,
        FIRST_NAME VARCHAR2(10),
        LAST_NAME VARCHAR2(10),
        DATE_OF_BIRTH VARCHAR2(10),
        CAR_NUMBER NUMBER(5));
```

We can also write our create statement in a SQL file. Then we read this file (Oracle style).

```
-- exam1.sql

DROP TABLE Employees;
DROP TABLE Cars;

CREATE TABLE Employees
       (EMPLOYEE_NUMBER NUMBER(5) NOT NULL,
        FIRST_NAME VARCHAR2(10),
        LAST_NAME VARCHAR2(10),
        DATE_OF_BIRTH VARCHAR2(10),
        CAR_NUMBER NUMBER(5));

INSERT INTO Employees VALUES
   (10001,'John','Smith','28-AUG-43',5);
INSERT INTO Employees VALUES
   (10083,'Arvid','Sharma','24-NOV-54',null);
INSERT INTO Employees VALUES
   (10035,'Sean','Washington','22-FEB-81',12);

CREATE TABLE Cars
       (CAR_NUMBER NUMBER(2),
        MAKE VARCHAR2(14),
        MODEL VARCHAR2(14),
        YEAR NUMBER(4));

INSERT INTO Cars VALUES
       (5,'BMW','Z3',1998);
INSERT INTO Cars VALUES
       (12,'VW','POLO',1999);
```

In mySQL the command

```
LOAD DATA INFILE 'file_name.txt'
INTO TABLE tbl_name
```

reads rows from a text file into a table. If the `LOCAL` keyword is specified, the file is read from the client host.

In the following SQL file we also include a primary key and foreign key (Oracle style).

```
-- exam2.sql

DROP TABLE Employees;
DROP TABLE Cars;

CREATE TABLE Cars
       (CAR_NUMBER NUMBER(5),
        MAKE VARCHAR2(14),
        MODEL VARCHAR2(14),
        YEAR NUMBER (4),
        CONSTRAINT PK_CARS
Primary Key (CAR_NUMBER));

CREATE TABLE Employees
       (EMPLOYEE_NUMBER NUMBER(5) NOT NULL,
        FIRST_NAME VARCHAR2(20),
        LAST_NAME VARCHAR2(20),
        DATE_OF_BIRTH VARCHAR2(10),
        CAR_NUMBER NUMBER (5),
        CONSTRAINT PK_EMPLOYEES
        Primary Key (EMPLOYEE_NUMBER,CAR_NUMBER),
        CONSTRAINT FK_EMPLOYEES_CAR_NUMBER
        Foreign Key (CAR_NUMBER) REFERENCES Cars (CAR_NUMBER));

INSERT INTO Cars VALUES
       (5,'BMW','Z3',1998);
INSERT INTO Cars VALUES
       (12,'VW','Polo',1999);
INSERT INTO Cars VALUES
       (6,'MERCEDES','BENZ',1999);

INSERT INTO Employees VALUES
       (10001,'John','Smith','28-Aug-43',5);
INSERT INTO Employees VALUES
       (10083,'Arvid','Sharma','24-Nov-54',6);
INSERT INTO Employees VALUES
       (10035,'Sean','Washington','28-Feb-81',12);
```

2.3.10 DROP TABLE Command

The SQL `DROP TABLE` command is used if we decide that a base relation in the database is not needed any longer. We can then delete the relation and its definitions using the `DROP TABLE` command. This also removes a tuple from the `SYSTABLES` and system catalog table. This means `DROP TABLE` deletes all rows and removes the table definition from the database.

The syntax is as follows

```
DROP TABLE table_name;
```

Example.

```
DROP TABLE Cars;
```

The `DROP` command is also used together with the `ALTER` command.

We may not drop a table if it is referenced by another table.

2.3.11 ALTER TABLE Command

Sometimes it is necessary to modify a table's definition. The `ALTER TABLE` statement serves this purpose. This statement changes the structure of a table, not its contents. Using `ALTER TABLE`, the changes we can make to a table include the following

1) Adding a new column to an existing table

2) Increasing or decreasing the width of an existing column

3) Changing an existing column from mandatory to optional or vice versa

4) Specifying a default value for an existing column

5) Specifying other constraints for an existing column

Here are the four basic forms of the `ALTER TABLE` statement:

```
ALTER TABLE table_name
ADD (column_specification | constraint , ...
     column_specification | constraint);

ALTER TABLE table_name
MODIFY (column_specification | constraint , ...
        column_specification | constraint);

ALTER TABLE table_name DROP PRIMARY KEY;

ALTER TABLE table_name DROP CONSTRAINT constraint;
```

The variables are defined as follows:

```
table_name  name of the table
column_specification  valid specification for a column
(column name and datatype)
constraint  column or table constraint
```

The first form of the statement is used for adding a column, the primary key, or a foreign key to a table. The second form of the statement is used to modify an existing column. Among other things, we can increase a column's width or transform it from mandatory to optional. The third and fourth forms of the `ALTER TABLE` statement are used for dropping a table's primary key and other constraints.

Example.

```
ALTER TABLE Cars
DROP Make;
```

2.4 Set Operators

In this section we consider the set operators in the `SELECT` statement. The SQL language is a partial implementation of the model as envisioned by Codd, the father of relational databases. As part of that implementation SQL provides three set operators

`INTERSECT, UNION, MINUS`

The mathematical symbols are $\cap$, $\cup$, $\setminus$. Let A and B be two sets. The definitions are

$$A \cap B := \{\, x \in A \text{ and } x \in B \,\}$$

$$A \cup B := \{\, x \in A \text{ or } x \in B \,\}$$

$$A \setminus B := \{\, x \in A \;\; x \notin B \,\}\,.$$

The symmetric difference Δ of the sets A and B is defined as

$$A \triangle B := (A \cup B) \setminus (A \cap B)\,.$$

Example. Let

$$A = \{\, \texttt{"Olten"}, \texttt{"Johannesburg"}, \texttt{"Singapore"} \,\}$$

$$B = \{\, \texttt{"Johannesburg"}, \texttt{"Stoeckli"} \,\}\,.$$

Then

$$A \cap B = \{\, \texttt{"Johannesburg"} \,\}$$

$$A \cup B = \{\, \texttt{"Olten"}, \texttt{"Johannesburg"}, \texttt{"Singapore"}, \texttt{"Stoeckli"} \,\}$$

$$A \setminus B = \{\, \texttt{"Olten"}, \texttt{"Singapore"} \,\}$$

If the set is finite with n-elements, then the number of subsets (including the empty set and the set itself) is 2^n.

The `INTERSECT` Operator

The `INTERSECT` operator returns the rows that are common between two sets of rows. The syntax for using the `INTERSECT` operator is

```
SELECT stmt1
INTERSECT
SELECT stmt2
[ORDER BY clause]
```

The variables are defined as follows:

`SELECT stmt1`

and

`SELECT stmt2`

are valid `SELECT` statements. The `ORDER BY` clause references the columns by number rather than by name.

Requirements and considerations for using the `INTERSECT` operator are:

The two `SELECT` statements may not contain an `ORDER BY` clause. However, we can order the result of the entire `INTERSECT` operation.

The number of columns retrieved by `SELECT stmt1` must be equal to the number of columns retrieved by `SELECT stmt2`.

The datatypes of the columns retrieved by `SELECT stmt1` must match the datatypes of the columns retrieved by `SELECT stmt2`.

The optional `ORDER BY` clause differs from the usual `ORDER BY` clause in a `SELECT` statement because the columns used for ordering must be refereneced by number rather than by name. The reason is that SQL does not require that the column names retrieved by `SELECT stmt1` be identical to the column names retrieved by `SELECT stmt2`. Therefore, we must indicate the columns to be used in ordering results by their position in the select list.

The UNION Operator

To combine the rows from similar tables or produce a report or to create a table for analysis we apply the UNION operator. The syntax is

```
SELECT stmt1
UNION
SELECT stmt2
[ORDER BY clause]
```

The variables are defined as follows

```
SELECT stmt1
```

and

```
SELECT stmt2
```

are valid SELECT statements. The ORDER BY clause references the columns by number rather than by name.

The MINUS Operator

The syntax is

```
SELECT stmt1
MINUS
SELECT stmt2
[ORDER BY clause]
```

The variables are defined as follows

```
SELECT stmt1
```

and

```
SELECT stmt2
```

are valid SELECT statements. The ORDER BY clause references the columns by number rather than by name.

Both C++ using the STL with the class `set` and Java with the class `TreeSet` allow set-theoretical operations. We can find the union, intersection and difference of two finite sets. We can also get the cardinality of the finite set (i.e. the number of elements). Furthermore, we can find out whether a finite set is a subset of another finite set and whether a finite set contains a certain element.

In C++ the class `set` is a sorted associative container that stores objects of type Key. The class `set` is a simple associative container, meaning that its value type, as well as its key type, is key. It is also a unique associative container meaning that no two elements are the same. The C++ class `set` is suited for the set algorithms

```
includes,
set_union,  set_intersection,
set_difference,  set_symmetric_difference
```

The reason for this is twofold. First, the set algorithms require their arguments to be sorted ranges, and, since the C++ class `set` is a sorted associative container, their elements are always sorted in ascending order. Second, the output range of these algorithms is always sorted, and inserting a sorted range into a `set` is a fast operation. The class `set` has the important property that inserting a new element into a `set` does not invalidate iterators that point to existing elements. Erasing an element from a set also does not invalidate any iterator, except of course, for iterators that actually point to the element that is being erased. Other functions in the C++ class `set` are

```
bool empty()
```

which returns true if the container is empty,

```
int size()
```

which returns the number of elements in the container.

The following program shows an application of this class.

```
// setstl.cpp

#include <iostream>
#include <set>
#include <algorithm>
#include <string.h>

using namespace std;

struct ltstr
{
```

```
   bool operator() (const char* s1,const char* s2) const
   {
   return strcmp(s1,s2) < 0;
   }
};

int main()
{
  const int N = 3;
  const char* a[N] = { "Steeb", "C++", "80.00" };
  const char* b[N] = { "Solms", "Java", "80.00" };

  set<const char*,ltstr> S1(a,a+N);
  set<const char*,ltstr> S2(b,b+N);
  set<const char*,ltstr> S3;

  cout << "union of the sets S1 and S2: " << endl;
  set_union(S1.begin(),S1.end(),S2.begin(),S2.end(),
            ostream_iterator<const char*>(cout," "),ltstr());
  cout << endl;

  cout << "intersection of sets S1 and S2: " << endl;
  set_intersection(S1.begin(),S1.end(),S2.begin(),S2.end(),
                   ostream_iterator<const char*>(cout," "),ltstr());
  cout << endl;

  set_difference(S1.begin(),S1.end(),S2.begin(),S2.end(),
                 inserter(S3,S3.begin()),ltstr());
  cout << "Set S3 difference of S1 and S2: " << endl;
  copy(S3.begin(),S3.end(),ostream_iterator<const char*>(cout," "));
  cout << endl;

  // S2 subset of S1 ?
  bool b1 = includes(S1.begin(),S1.end(),S2.begin(),S2.end(),ltstr());
  cout << "b1 = " << b1 << endl;

  // S4 subset of S2 ?
  const char* c[1] = { "Solms" };
  set<const char*,ltstr> S4(c,c+1);
  bool b2 = includes(S2.begin(),S2.end(),S4.begin(),S4.end(),ltstr());
  cout << "b2 = " << b2;

  return 0;
}
```

In Java the interface `Set` is a `Collection` that cannot contain duplicates elements. The interface `Set` models the mathematical set abstraction. The `Set` interface extends `Collection` and contains no methods other than those inherited from `Collection`. It adds the restriction that duplicate elements are prohibited. The JDK contains two general-purpose `Set` implementations. The class `HashSet` stores its elements in a hash table. The class `TreeSet` stores its elements in a red-black tree. This guarantees the order of iteration.

The following program shows an application of the `TreeSet` class.

```
// SetOper.java

import java.util.*;

public class SetOper
{
   public static void main(String[] args)
   {
   String[] A = { "Steeb", "C++", "80.00" };
   String[] B = { "Solms", "Java", "80.00" };

   TreeSet S1 = new TreeSet();
   for(int i=0; i < A.length; i++)
   S1.add(A[i]);
   System.out.println("S1 = " + S1);

   TreeSet S2 = new TreeSet();
   for(int i=0; i < B.length; i++)
   S2.add(B[i]);
   System.out.println("S2 = " + S2);

   // union
   TreeSet S3 = new TreeSet(S1);
   boolean b1 = S3.addAll(S2);
   System.out.println("S3 = " + S3);
   System.out.println("S1 = " + S1);

   // intersection
   TreeSet S4 = new TreeSet(S1);
   boolean b2 = S4.retainAll(S2);
   System.out.println("S4 = " + S4);
   System.out.println("S2 = " + S2);

   // (asymmetric) set difference
```

```
   TreeSet S5 = new TreeSet(S1);
   boolean b3 = S5.removeAll(S2);
   System.out.println("S5 = " + S5);

   // test for subset
   TreeSet S6 = new TreeSet(S1);
   boolean b4 = S6.containsAll(S2);
   System.out.println("b4 = " + b4);

   // is element of set (contains)
   boolean b = S1.contains("80.00");
   System.out.println("b = " + b);
   b = S2.contains("Steeb");
   System.out.println("b = " + b);
   }
}
```

The output is

```
S1 = [80.00, C++, Steeb]
S2 = [80.00, Java, Solms]
S3 = [80.00, C++, Java, Solms, Steeb]
S1 = [80.00, C++, Steeb]
S4 = [80.00]
S2 = [80.00, Java, Solms]
S5 = [C++, Steeb]
b4 = false
b = true
b = true
```

Other methods are

```
int size()
```

which returns the number of elements in this set (its cardinality) and

```
boolean isEmpty()
```

returns true if this `set` contains no elements.

2.5 Views

We use the `CREATE VIEW` statement to create a new *view* with its definition based upon a current table or another view of the database. A view is also known as a synthetic table. We can query or update the view as if it is a table. The table data is a view of what is stored in the real table. The view does not actually contain or store its own data. The only storage that a view actually requires is the `SELECT` statement that defines it. Thus a view is a stored query based on a query of one or more tables. Views are created for many purposes.

1) Restrict users to specific rows or columns of tables

2) Control the insert and update data of the table

3) Avoid redundancy of data

A view acts just like any other table and consists of all rows and columns that are the result of the `SELECT` statement used at the creation time. The data attributes for each column are derived from the original table. The user must have the select privilege on the table in order to create the view.

The syntax for creating a view is

```
CREATE VIEW view-name
(column1,...,columnN)
AS
SELECT statement
```

where `view-name` is the name of the view subject to the same requirements as other SQL object names. We can use a view in all SQL statements that deal with data viewing and manipulation, such as

`SELECT, INSERT, DELETE, UPDATE` .

A view cannot be used in database layout SQL statements such as the following:

1) Alter Index

2) Create Index

3) Drop Index

4) Alter Table

5) Create Table

6) Drop Table

7) Lock Table

8) Rename Table

9) Unlock Table

There are additional restrictions when using views. Because a view is not a real table, we cannot create an index, and the `INTO TEMP`, `UNION`, and `ORDER BY` functions cannot be processed. A view is based on one or more tables, so it should always reflect the most current changes of the tables. We can perform updates and inserts on a view with certain restrictions. First, we can update a view only if the columns are not derived from the `SELECT` statement that creates the view. An `INSERT` statement follows the same rules, and the view must be modifiable for any function to complete. The privileges on a view are checked at the time the `CREATE VIEW` statement is processed. The authorization is checked from the table from which we want to create our view. When a table is created, the grant to public is automatic, but that is not the case with a view. If the view is modifiable, the database server grants `DELETE`, `INSERT`, and `UPDATE` privileges.

2.6 Primary and Foreign Keys

A *primary key* is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the table `Employees`, the `Employee_No` column uniquely identifies that row. This means two things: no two rows can have the same `Employee_No`, and, even if two employees have the same first and last names, the `Employee_No` column ensures that the two will not be confused with each other, because the unique `Employee_No` column will be used through the database to track the employees, rather than the names.

A *foreign key* is a column in a table where that column is the primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where the column is the primary key. This correspondence is known as *referential integrity*. In the table `Employees` the `Car_Number` is a foreign key. It is the primary key in the table `Cars`.

When we define a foreign key, the database system verifies the following

1) A primary key has been defined for the table by the foreign key.

2) The number of columns composing the foreign key matches the number of primary key columns.

3) The datatype and width for each foreign key column matches the datatype and width of each primary key columm.

For example in mySQL we set up a primary key as follows:

```
> DROP DATABASE db;
> CREATE DATABASE db;
> USE db;
> CREATE TABLE customers (
> customers_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
> INDEX customers_index(customers_id),
> companyname VARCHAR(80) NOT NULL,
> address VARCHAR(80) NOT NULL,
> PRIMARY KEY(customers_id)
> )
```

Assume we have a lot of data (i.e. a lot of rows), and we want to try and improve our performance. The trick is to use an `INDEX` on commonly used columns in our queries. If we have a `WHERE` clause in our SQL statement, then the columns we are selecting on may benefit from indexes.

2.7 Datatypes in SQL

Datatypes in SQL can be classified as follows:

```
Numbers
Strings
Date and Time information
Large strings and BLOBs (Binary Large OBjects)
```

Datatypes slightly differ from system to system. Database systems offer several datatypes for storing numbers; each suited for a different purpose.

```
NUMBER   stores general numbers
DECIMAL  stores fixed-point numbers
FLOAT    stores floating-point numbers
         (equivalent to double in C++ and Java)
```

The `NUMBER` datatype (specific to Oracle) offers the greatest flexibility for storing numerical data. It accepts positive and negative integers and real numbers, and has from 1 to 38 digits of precision. The syntax is

```
NUMBER(precision,scale)
```

The variables are defined as follows: `precision` is the maximum number of digits to be stored, `scale` is used to indicate the position of the decimal point number of digits to the right (positive) or left (negative) of the decimal point. The scale can range from -84 to 127. In mySQL we have `INT` for signed and unsigned integers (4 bytes).

One can store from 1 to 38 digits of precision for a number. The number of bytes required to store a number in a database depends on how many digits are used to express the number. If we limit the precision, the database system limits the values that can be stored in the column to the defined precision.

The `DECIMAL` type is used for values for which it is important to preserve exact precision, for example with monetary data. When declaring a column the precision and scale can be (and usually is) specified. For example

```
salary DECIMAL(9,2)
```

In this example, 9 (precision) represents the number of significant decimal digits that will be stored for values, and 2 (scale) represents the number of digits that will be stored following the decimal point. In this case, therefore, the range of values that can be stored in the `salary` column is from `-9999999.99` to `9999999.99`.

To store strings in a database system, we can choose from several datatypes.

```
CHAR
VARCHAR
VARCHAR2 (Oracle)
LONG or LONGVARCHAR
```

The bulk of many databases is character data. The `CHAR` datatype stores fixed-length character strings of up to 255 characters, but specifying the `n` in `CHAR(n)`. If we do not specify a length, a `CHAR` column stores a single character. Since the `CHAR` datatype stores fixed-length columns, we use it when we are defining columns that will contain a single character. Using the `CHAR` datatype to store larger strings is not efficient because we waste storage space. The size for `VARCHAR(n)` differs from system to system. For example, Sybase 11.0 has $n <= 255$ and IBM DB2 has $n <= 4000$. In Oracle `VARCHAR(n)` is replaced by `VARCHAR2` which can store up to 2000 characters in a single column. In Oracle `VARCHAR2` is the prefered datatype for storing strings, because it stores variable-length strings. If we need more storage we choose the `LONG` or `LONGVARCHAR` datatype. We can store 2GB of characters in a `LONG` column. We face a number of restrictions on the use of `LONG` columns in SQL. We cannot use functions or operators to search or modify the contents of a `LONG` column. We can think of a `LONG` column as a large container into which we can store or retrieve data - but not manipulate it. In mySQL the `CHAR` and `VARCHAR` types are similar, but differ the way they are stored and retrieved. The length of a `CHAR` column is fixed to the length that we declare when we create the table. The length may be 0 to 255. Values in `VARCHAR` are variable-length strings. We can declare a `VARCHAR` column to be any length between 1 and 255.

Each database system also provides the datatypes for date and time. The `DATE` datatype provides storage for both date and time information. For example, Oracle always allocates a fixed 7 bytes for a `DATE` column, even if we are using a `DATE` column to store date information only or time information only. Database systems have quite a few built-in functions specifically for manipulating `DATE` values and expressions. The `DATE` datatype can store dates in the range of January 1, 4712 B. C. to December 31, 4712 A. D. Most database system use the default format

```
DD-MO-YY
```

for entering and displaying dates. Most systems allow the format to be changed

In mySQL the date and time data types are `DATETIME`, `DATE`, `TIMESTAMP` and `YEAR`. Dates must be given in the year-month-day order. The `DAYTIME` type is used when we need values that contain both date and time information. MySQL retrieves and displays `DATETIM` values in `'YYYY-MM-DD' HH:MM:SS`. The `TIMESTAMP` column provides a type that we use to automatically mark `INSERT` and `UPDATE` operations with the current date and time.

Most database systems provide for the storage of binary large objects (BLOBs). BLOBs include documents, graphics (for example `jpeg` files), sound, video - actually, any type of binary file we can have. For example in Oracle `LONG RAW` datatype is designed for BLOB storage. When we want to associate a BLOB with a normal row, two choices are available: store the BLOB in an operating system file and store the directory and filename in the associated table or store the BLOB itself in a `LONG RAW` column.

Some database system also provide the datatype `RAW`. It can accommodate up to 255 bytes of binary data. Owing to this storage restriction, a `RAW` column is less useful than a `LONG RAW` column.

Newer database systems also have the datatype `CLOBs` to store character data, `NCLOBs` to store character data to support Asian languages, and `BFILEs` to reference binary files in the file system.

In mySQL we have four BLOB types. The four BLOB types `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB` differ only in the maximum length of the values they can hold.

mySQL also has four TEXT types `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT` correspond to the four BLOB types and have the same maximum length and storage requirements. The only difference between BLOB and TEXT types is that sorting and comparison is performed in case-sensitive fashion for `BLOB` values and case-insensitive fashion for TEXT values. In other words, a TEXT is a case-insensitive BLOB. Since `BLOB` and `TEXT` values may be extremely long, we may run up against constraints when using them. If we want to use `GROUP BY` or `ORDER BY` on a BLOB or TEXT column, we must convert the column value into a fixed-length object.

2.8 Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a *join*. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, the year of car and the price. This information is stored in the table `Cars`. There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In the present case, the column that appears in two tables is `Car_Number`, which is the primary key for the table `Cars` and the foreign key for the table `Employees`. If the 1999 BMW Z3 were wrecked and deleted from the `Cars` table, then `Car_Number 5` would also be have to be removed from the `Employees` table in order to maintain what is called *referential integrity*. Otherwise, the foreign key column `(Car_Number)` in `Employees` would contain an entry that did not refer to anything in table `Cars`. A foreign key must either be `null` or equal to an existing primary key value in the table to which it refers. This is different from a primary key, which may not be `null`. There are several `null` values in the `Car_Number` column in the table `Employees` because it is possible for an employee not to have a company cars.

Example. The following command aks for the first and last names of employees who have company cars for the make and model. Note that the `FROM` clause lists both `Employees` and `Cars` because the requested data is contained in both tables. Using the table name and the *access operator* . before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name, Cars.Make,
       Cars.Model, Cars.Price
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number;
```

The result set is

```
First_Name    Last_Name     Make        Model    Price
----------    ----------    --------    -----    -----
John          Smith         BMW         Z3       62000
Florence      Wojokowski    Volkswagen  Lupo     30000
-----------------------------------------------------
```

Thus a join is a process in which two or more tables are combined to form a single table. The join can be dynamic, where two tables are merged to form a virtual table, or static, where two tables are joined and saved for future reference. A static join is usually a stored procedure which can be invoked to refresh and then query the saved table. Joins are performed on tables that have a column of common information. Conceptually, there are many types of joins, which are discussed later in this section.

Consider the following example with three tables. We have a table

Students

```
Student_ID    Student_Name
==========    ============
1              John
2              Mary
3              Jan
4              Jack
==========================
```

a table Courses

```
Course_ID  Course_Title
=========  =============
S1           Math
S2           English
S3           Computer
S4           Logic
=======================
```

and a table StudentCourses

```
Student_ID    Course_ID
==========    =========
2              S2
3              S1
4              S3
======================
```

MySQL supports the following `JOIN` syntaxes for use in `SELECT` statements

```
SELECT ... CROSS JOIN table_reference
SELECT ... INNER JOIN table_reference
SELECT ... LEFT OUTER JOIN table_reference
SELECT ... NATURAL JOIN table_referece
```

Inner Join

A simple join called the *inner join* with the Students and StudentCourses tables give us the following table (Inner Join Table)

```
Student_ID      Student_Name     Course_ID
==========      ============     =========
2               Mary              S2
3               Jan               S1
4               Jack              S3
=========================================
```

That is, we get a new table which combines the Students and StudentCourses tables by adding the `Student_Name` column to the StudentCourses table.

Just because we are using the `Student_ID` to link the two tables does not mean that we should fetch that column. We can exclude the key field from the result table of an inner join. The SQL statement for this inner join is as follows:

```
SELECT Students.Student_Name, StudentCourses.Course_ID
FROM Students, StudentCourses
WHERE Students.Student_ID = StudentCourses.Student_ID
```

Outer Join

An *outer join* between two tables (say Table1 and Table2) occurs when the result table has all the rows of the first table and the common records of the second table. The first and second table are determined by the order in the SQL statement. There is a `LEFT OUTER JOIN` and a `RIGHT OUTER JOIN`.

If we assume a SQL statement with the

```
SELECT FROM Table1, Table2
```

clause, in a left outer join, all rows of the first table (Table1) and common rows of the second table (Table2) are selected. In a right outer join, all records of the second table (Table2) and common rows of the first table (Table1) are selected. A left outer join with the Students table and the StudentCourses table creates

```
Student_ID    Student_Name     Course_ID
==========    ============     =========
1              John              null
2              Mary              S2
3              Jan               S1
4              Jack              S3
=======================================
```

This join is useful if we want the names of all students, regardless of whether they are taking any subjects this term, and the subjects taken by the students who have enrolled in this term. Some people call it an if-any join, as in,

```
"Give me a list of all students and the subjects they are taking, if any".
```

The syntax is

```
SELECT column_list FROM table_reference
  LEFT | RIGHT | FULL | [OUTER] JOIN table_reference
  ON predicate
  [LEFT | RIGHT | FULL [OUTER] JOIN table_reference
  ON predicate ... ]
```

We use an `OUTER JOIN` to join two tables, a source and joining table, that have one or more columns in common. One or more columns from each table are compared in the `ON` clause for equal values. The primary difference between inner and outer joins is that, in outer joins, rows from the source table that do not have a match in the joining table are not excluded from the result set. Columns from the joining table for rows in the source table without matches have `NULL` values.

The `LEFT` modifier causes all rows from the table on the left of the `OUTER JOIN` operator to be included in the result set, with or without matches in the table to the right. If there is no matching row from the table on the right, its columns contain `NUL` values.

The `RIGHT` modifier causes all rows from the table on the right of the `OUTER JOIN` operator to be included in the result set, with or without matches. If there is no matching row from the table on the left, its columns contain `NUL` values.

Full Outer Join

The *full outer join* returns all the records from both the tables merging the common rows.

```
Student_ID    Student_Name   Course_ID
==========    ============   =========
1             John            null
2             Mary            S2
3             Jan             S1
4             Jack            S3
null          null            S4
====================================
```

What if we want only the students who haven't enrolled in this term or the subjects who have no students (the tough subjects or professors)? Then, we resort to the subtract join. In this case, the join returns the rows that are not in the second table. Remember, a subtract join has only the fields from the first table. By definition, there are no records in the second table.

There are many other types of joins, such as the self join, which is a left outer join of two tables with the same structure. An example is the assembly/parts explosion in a Bill of Materials application for manufacturing. The join types that we have discussed so far are enough for normal applications. In all of these joins, we were comparing columns that have the same values; these joins are called equi-joins. Joins are not restricted to comparing columns of equal values. We can join two tables based on column value conditions (such as the column of one table being greater than the other). For equi-joins, as the column values are equal, we retrieved only one copy of the common column. Then, the joins are called natural joins. When we have a non equi-join, we might need to retrieve the common columns from both tables.

2.9 Stored Procedure

A *stored procedure* is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as the name implies, store it in the database. A DBMS compiles a stored procedure before storing it, so it does not have to be recompiled each time it is called, cutting down on execution time.

The following code is an example of how to create a very simple stored procedure, but since each DBMS has its own way of creating stored procedures, it is meant to give only an idea of how it might be done and is not meant to be used in actual code.

This example creates a stored procedure called `Assign_Car_Num`, which updates the `Employees` table. It takes two input arguments, the number of the car to be added and the number of the employee to whom the car is being assigned. The type of each argument must be `integer`, as specified in the first line.

```
create procedure Assign_Car_Num(Car_No integer,Emp_No integer)
as begin
UPDATE Employees
SET Car_Number = Car_No
WHERE Employeee_Number = Emp_No
end
```

2.10 MySQL Commands

The main MySQL commands are:

The command

```
> SHOW DATABASES;
```

lists the databases managed by the server.

To find out which database is currently selected, we use the command

```
> SELECT DATABASE();
```

The

```
> USE db_name;
```

statement tells mySQL to use the `db_name` database as the default database for subsequent queries. The database remains current until the end of the session or until another `USE` statement is issued.

We can find out about the structure of the table with the command

```
> DESCRIBE table_name;
```

To find out what tables the current database contains (for example when we are not sure about the names of a table), we use the command

```
> SHOW TABLES;
```

The

```
LOAD DATA INFILE 'file_name.txt'
INTO TABLE tbl_name
```

statement reads rows from a text file into a table at a very high speed. If the `LOCAL` keyword is specified, the file is read from the client host.

With the command

```
> SOURCE sql_file
```

we can load an sql-file after the commands `CREATE DATABASE db_name;`
and `CONNECT db_name;`.

2.11 Cursors

A relational database query normally returns many rows of data. But an application program usually deals with one row at a time. Even when an application can handle more than one row-for example, by displaying the data in a table or spreadsheet format-it can still handle only a limited number of rows. Also, updating, modifying, deleting, or adding data is done on a row basis. This is where the concept of cursors come in the picture. In this context, a cursor is a pointer to a row. It is like the cursor on the CRT-a location indicator. Different types of multi-user applications need different types of data sets in terms of data concurrency. Some applications need to know as soon as the data in the underlying database is changed. Such as the case with reservation systems, the dynamic nature of the seat allocation information is extremely important. Others such as statistical reporting systems need stable data; if data is in constant change, these programs cannot effectively display any results. The different cursor designs support the need for the various types of applications.

A cursor can be viewed as the underlying data buffer. A fully scrollable cursor is one where the program can move forward and backward on the rows in the data buffer. If the program can update the data in the cursor, it is called a scrollable, updatable cursor. An important point to remember when we think about cursors is the transaction isolation. If a user is updating a row, other users might be viewing the row in a cursor of their own. Data consistency is important here. Worse, the other users also might be updating the same row!

The `ResultSet` in JDBC API is a cursor. But it is only a forward scrollable cursor-this means we can move only forward using the `getNext()` method.

ODBC cursors are very powerful in terms of updatability, concurrency, data integrity, and functionality. The ODBC cursor scheme allows positioned delete and update and multiple row fetch (called a rowset) with protection against lost updates. ODBC supports static, keyset-driven, and dynamic cursors.

In the static cursor scheme, the data is read from the database once, and the data is in the snapshot recordset form. Because the data is a snapshot (a static view of the data at a point of time), the changes made to the data in the data source by other users are not visible. The dynamic cursor solves this problem by keeping live data, but this takes a toll on network traffic and application performance.

The keyset-driven cursor is the middle ground where the rows are identified at the time of fetch, and thus changes to the data can be tracked. Keyset-driven cursors are useful when we implement a backward scrollable cursor. In a keyset-driven cursor, additions and deletions of entire rows are not visible until a refresh. When we do a backward scroll, the driver fetches the newer row if any changes are made.

ODBC also supports a modified scheme, where only a small window of the keyset is

fetched, called the mixed cursor, which exhibits the keyset cursor for the data window and a dynamic cursor for the rest of the data. In other words, the data in the data window (called a RowSet) is keyset-driven, and when we access data outside the window, the dynamic scheme is used to fetch another keyset-driven buffer.

Static cursors provide a stable view of the data, because the data does not change. They are good for data mining and data warehousing types of systems. For these applications, we want the data to be stable for reporting executive information systems or for statistical or analysis purposes. Also, the static cursor outperforms other schemes for large amounts of data retrieval.

On the other hand, for online ordering systems or reservation systems, we need a dynamic view of the system with row locks and views of data as changes are made by other users. In such cases, we will use the *dynamic cursor*. In many of these applications, the data transfer is small, and the data access is performed on a row-by-row basis. For these online applications, aggregate data access is very rare.

Bookmark is a concept related to the cursor model, but is independent of the cursor scheme used. Bookmark is a placeholder for a data row in a table. The application program requests a bookmark for a row from the underlying database management system. The DBMS usually returns a 32-bit marker which can be later used by the application program to get to that row of data. In ODBC, we use the SQLExtendedFetch function with `SQL_FETCH_BOOKMARK` option to get a bookmark. The bookmark is useful for increasing performance of GUI applications, especially the ones where the data is viewed through a spreadsheet-like interface.

This is another cursor-related concept. If a cursor model supports positioned update/delete, then we can update/delete the current row in a result set without any more processing, such as a lock, read, or fetch. In SQL, a positioned `update` or `delete` statement is in the form of

```
UPDATE/DELETE Field or Column values etc. WHERE CURRENT OF cursor name;
```

The positioned `update` statement to update the fields in the current row is

```
UPDATE table SET field = value WHERE CURRENT OF cursor name;
```

The positioned `delete` statement to delete the current row takes the form of

```
DELETE table WHERE CURRENT OF cursor name;
```

Generally, for this type of SQL statement to work, the underlying driver or the DBMS has to support updatability, concurrency, and dynamic scrollable cursors. But there are many other ways of providing the positioned update/delete capability at the application program level.

2.12 PL and SQL

SQL is a language without procedural capabilities. However, Oracle offers procedural language extensions to SQL through the PL/SQL language. PL/SQL is a block-structered language with a syntax similar to PASCAL. In addition to supporting embedded SQL statements, PL/SQL offers standard programming constructs such as procedure and function declarations, control statements such as `IF ... ELSE` and `LOOP`, and declared variables. An example is given below.

```
> drop table test_table;
Table dropped.
>
> create table test_table (
> record_number int,
> current_date date);
Table created.
>
> DECLARE
>
> max_records CONSTANT int := 100;
> i int := 1;
>
> BEGIN
>
> FOR i IN 1..max_records LOOP
>
>  INSERT INTO test_table
>    (record_number,current_date)
>  VALUES
>    (i,SYSDATE);
>
> END LOOP;
>
> COMMIT;
> END;
```

2.13 ABAP/4 and SQL

ABAP/4 contains a subset of SQL (called Open SQL) that is an integral part of the language. ABAP/4 programs using Open SQL can access data from all database systems that are supported by the R/3 system. Internal tables and database tables work together. The contents of a database table can be mapped into an internal table at runtime, so that the internal table is a snapshot of a database table. For example, to create a list containg all the entries in a database table, we can read the table contents into an internal table with the same structure and display each line as follows:

```
tables customers.
data all_customers like customers occurs 100
                  with header line.
select * from customers into table all_customers.
loop at all_cusomers.
write: / all_customers-name.
endloop.
```

Using a `where` clause of a `SELECT` statement, the set of selected records can be restricted according to a logical condition. We can also specify a subset of all table fields and use aggregate functions, such as the number of table entries satisfying a certain condition.

ABAP/4 also provides several commands for changing database table:

```
insert, update, modify, delete .
```

Using these commands we can change a single entry as well as a set of entries. The tables of a relational database always have a flat structure (i.e., a structure cannot contain another table).

2.14 Query Processing and Optimization

Queries in a high level language such as SQL must be:

1. Scanned and Parsed (SCANNER-PARSER). The Scanner identifies the tokens or language elements, and the Parser check for syntax or grammar validity.

2. Validated (VALIDATOR). The Validator check for valid names and semantic correctness.

3. Converted to internal representation (usually a QUERY TREE)

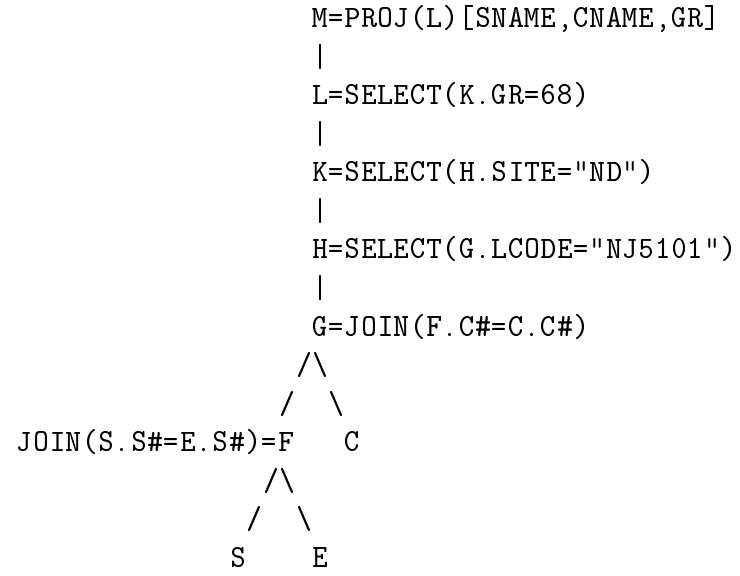
For example given the database with the three tables called `S`, `C` and `E`

```
S
   |S#|SNAME |LCODE |
   ===|======|======|
   |25|CLAY  |NJ5101|
   |32|THAISZ|NJ5102|
   |38|GOOD  |FL6321|
   |17|BAID  |NY2091|
   |57|BROWN |NY2092|
   ==================

C
   |C#|CNAME|SITE|
   |==|=====|====|
   |8 |DSDE |ND  |
   |7 |CUS  |ND  |
   |6 |3UA  |NJ  |
   |5 |3UA  |ND  |
   ===============

E
   |S#|C#|GR|
   |==|==|==|
   |32|8 |89|
   |32|7 |91|
   |25|7 |68|
   |25|6 |76|
   |32|6 |62|
   ==========
```

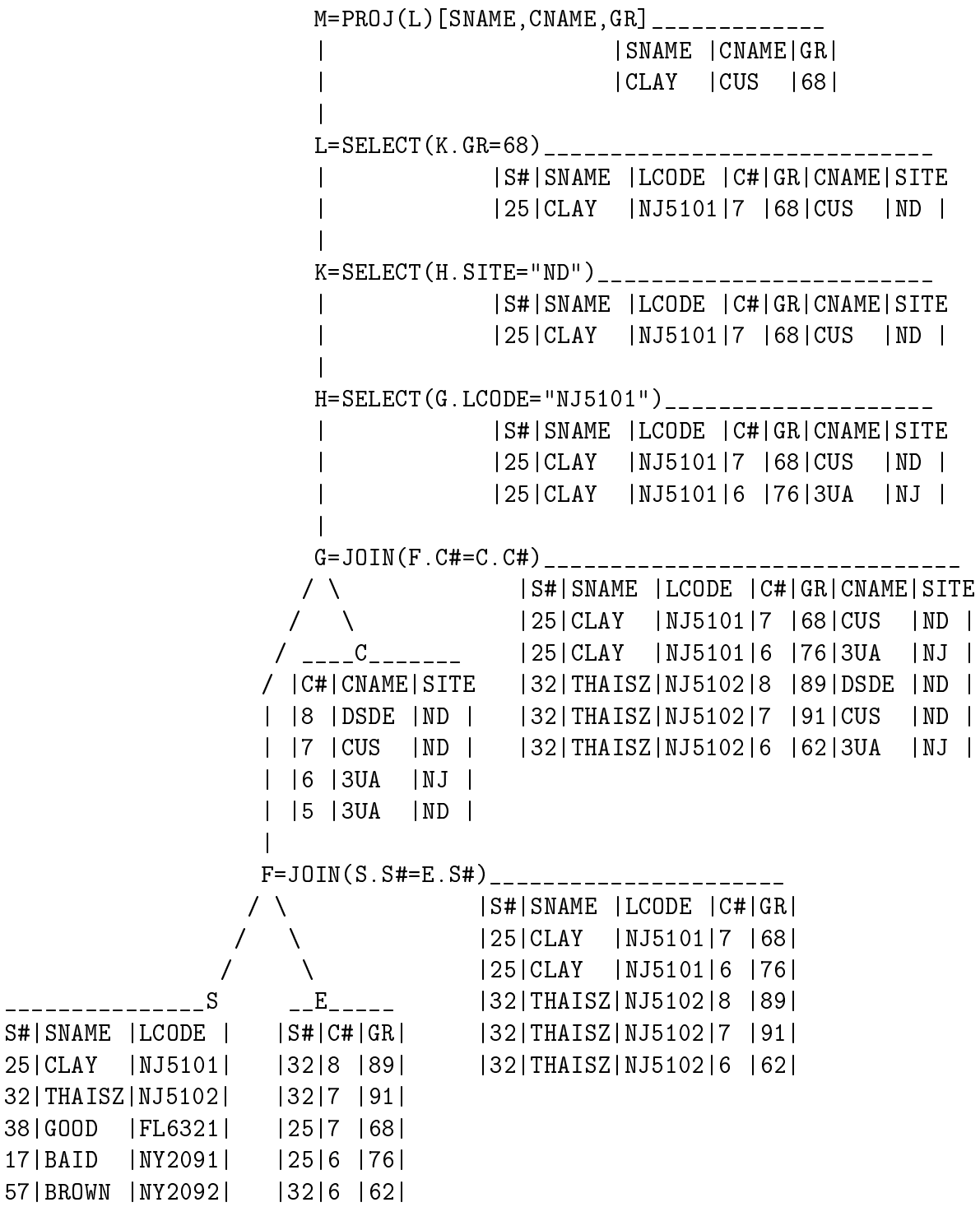
The SQL command

```
SELECT S.SNAME, C.CNAME, E.GR FROM S,C,E
  WHERE
    S.S#=E.S# and C.C#=E.C# and S.LCODE=NJ5101
              and C.SITE="ND" and E.GR=68;
```

would get SCANNED, PARSED, VALIDATED and then CONVERTED to a query tree which follows the `WHERE`-clause sequencing:

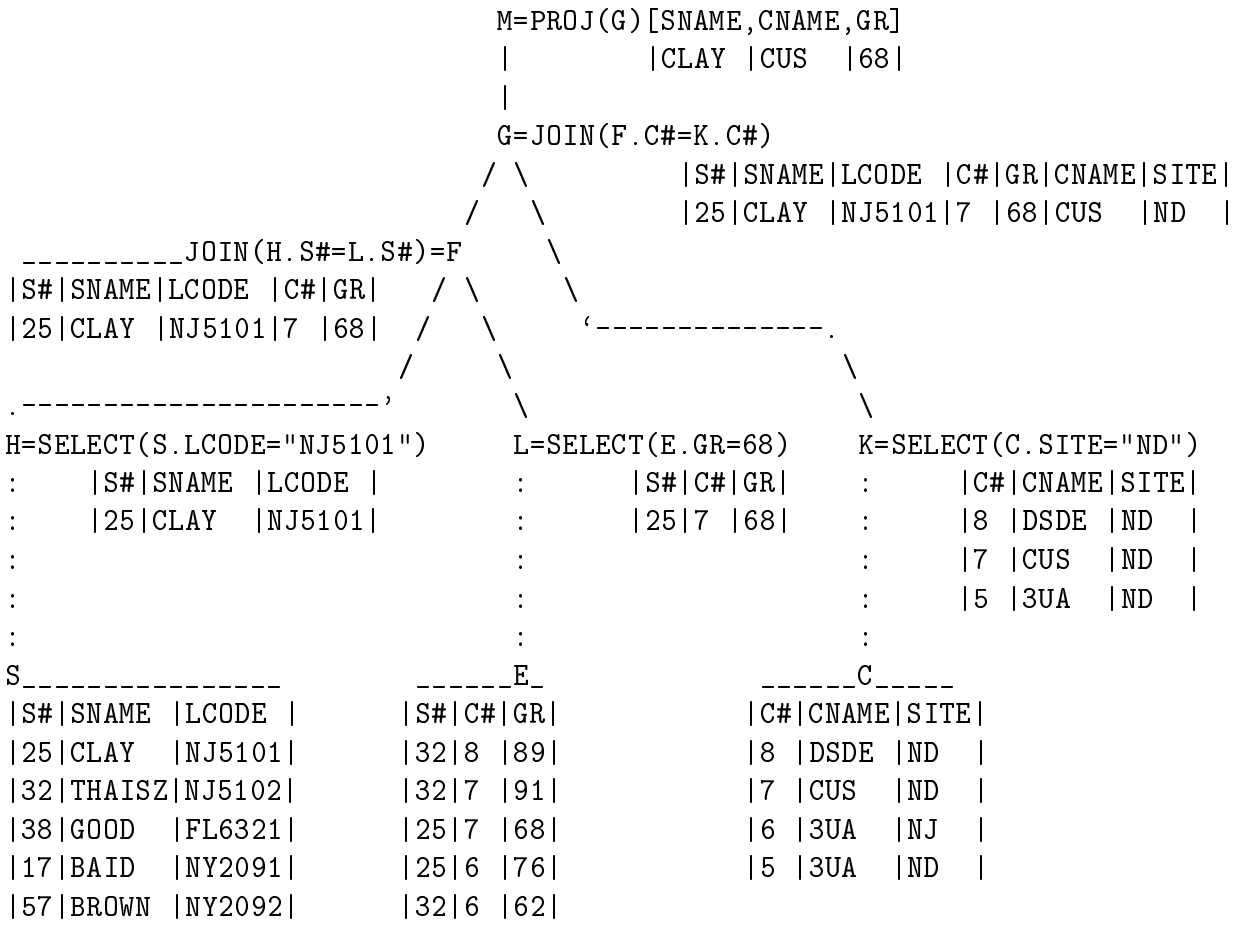


The results at each step (starting from the bottom of the tree) is



4. Optimized: (QUERY OPTIMIZER) Execution stategy must be devised (plan for executing, accessing data, storing intermediate results). We have to choose a execution strategy (query tree?)

Is the query tree above efficient? Is the tree below better?



Is this tree really optimal? The following could be done:

i. The SITE attribute can be projected off of `K` (does not require elimination of duplicates because it is not part of the key)

ii. The LCODE attribute can be projected off of `H` (does not require elimination of duplicates because it is not part of the key)

iii. `S#` could be projected off of `F` (it is part of the key but duplicate elimination could be deferred until `M` since it will have to be done again there anyway - thus this projection can be a "non duplicate-eliminating" projection too. These projections take no time, whereas duplicate eliminating projections take a lot of time).

iv. `C#` can be (non-duplicate-eliminating) projected off of `G`.

The rules for query optimization are:

a. Always do SELECTS first (push to the bottom of the tree).

b. Always do the attribute elimination part of PROJECT as soon as possible (push down).

c. Only do duplicate elimination once (at the top-most PROJECT only).

Chapter 3

Normal Forms

3.1 Introduction

When designing relational database, the tables must be in normal form. When the database is in normal form, data can be retrieved, changed, added and deleted without anomalities. The process of RDBMS normalization is an exercise in logic - from a managerial data access and usage perspective. It requires only a little thought and examination of how data are most logically used. When thinking about the logical normalization process we first look at all of the data required to accomplish a task.

Normalisation process looks at the data from application viewpoint. Thus we can not do normalisation without detailed application knowledge. Several current systems analysis methods produce normalised or near-normalised database schemas.

There are differing opinions how far the normalisation process should go, but in most practical cases Boyce-Codd normal form is the limit. Practical situations also exists, where normal form rules will be violated for application reasons or processing efficiency.

There are also database systems ignoring the first normal form. These are called Non-First Normal Form (NF2) systems. Automatic database conversion from Non-First Normal Form into fully normalised form is possible.

We recall that keys are the columns, whose value must be known in order to select proper rows of the table or to join tables. Whole primary key means all component columns of the primary key.

Initially Codd (1972) presented the first three normal forms (1NF, 2NF, 3NF) all based on functional dependencies among the attributes of a relation. Later Boyce and Codd proposed another normal form called the Boyce-Codd normal form. The fourth and fifth normal forms are based on multivalue and join dependencies and were proposed later.

Let us recall several definitions that are used in the world of database administration and normalization.

entity. The word *entity* as it relates to databases can simply be defined as the general name for the information that is to be stored within a single table.

Example. If we were interested in storing information about the schools students, then student would be the entity. The student entity would likely be composed of several pieces of information, for example: student identification number, name, and email address. These pieces of information are better known as attributes.

functional dependency. It addresses the concept that certain data fields are dependent upon other data fields in order to uniquely define and access them.

primary key. A primary key uniquely identifies a row of data found within a table.

Example. Referring to a school system, the student identification number would be the primary key for the student table since an ID would uniquely identify each student. A primary key might not necessarily correspond to one specific attribute. It could be the result of a combination of several components of the entity. For example, while a location could not be a primary key for a class, since there might be several classes held there throughout the day, the combined time and location would make a primary key, since no two classes could be held at the same time in the same location. When multiple attributes are used to derive a primary key, this key is known as a concatenated primary key.

relationship. Understanding of the various relationships both between the data items forming the various entities and between the entities themselves forms the crux of database normalization. There are three types of data relationships:

one-to-one (1:1). A one-to-one relationship signifies that each instance of a given entity relates to exactly one instance of another entity.

Example. Each student would have exactly one grade record, and each grade record would be specific to one student.

one-to-many (1:M). A one-to-many relationship signifies that each instance of a given entity relates to one or more instances of another entity.

Example. One professor entity could be found teaching several classes, and each class could in turn be mapped to one professor.

many-to-many (M:N). A many-to-many relationship signifies that many instances of a given entity relate to many instances of another entity.

Example. A schedule could be comprised of many classes, and a class could be found within many schedules.

foreign key. A foreign key forms the basis of a 1:M relationship between two tables. The foreign key can be found within the M table, and maps to the primary key found in the 1 table.

Example. The primary key in the professor table (probably a unique identification number) would be introduced as the foreign key within the classes entity, since it would be necessary to map a particular professor to several classes.

Entity-relationship diagram (ERD). An ERD is essentially a graphical representation of the database structure. These diagrams, regardless of whether they are built using the latest design software or scrawled on a napkin with a crayon, are immensely useful towards attaining a better understanding of the dynamics of the various database relationships.

Normal forms

1NF: Each row has equal number of columns and each column contains undivisible (atomic) data. Reduce entities to first normal form (1NF) by removing repeating or multi-valued attributes to another, child entry (table).

In other words a relation is in 1NF if and only if all underlying domains contain atomic values only.

2NF: The table must have a primary key. No variable column exists, which were specifiable with only some, but not all key columns. Reduce first normal form entities to second normal form (2NF) by removing attributes, that are not dependent on the whole primary key.

In other words a relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.

3NF: No variable columns may be specifiable with other variable columns. Reduce second normal form entities to third normal form (3NF) by removing attributes that depend on other, non-key attributes (other than alternate keys).

BCNF: Reduce third normal form entities to Boyce-Codd Normal Form (BCNF) by ensuring, that they are in third normal form for any feasible choice of candidate key as primary key.

4NF: Reduce Boyce-Codd Normal Form entities to fourth normal form (4NF) by removing any independently multivalued components of the primary key to two new parent entities. Retain the original (now child) entity only if it contains other, non-key attributes.

5NF: Reduce fourth normal form entities to fifth normal form (5NF) by removing pairwise cyclic dependencies (appearing within composite primary keys with three or more component attributes) to three or more new parent entities.

PJNF: (Alagic) A relation R is in the projection/join normal form if and only if every nontrivial join dependency, which holds in R is the result of keys.

Overnormalisation rule: In general, do not split fully normalised entities into smaller entities.

3.2 Anomalies

The normalization rules are designed to prevent update and delete anomalies and data inconsistencies.

Upate Anomaly. If data appears more than once in the database, then if one item of the data is altered, we must alter all the instances of that data.

Insertion Anomaly. If new information is available we may not be able to enter the data until we have data for all fields.

Deletion Anomaly. If we delete some information from the database, we may accidentally remove additional data which we wish to keep.

As an example let us consider a table `Student`.

sno	sname	address	cno	cname	instructor	office_instructor
101	Smith	1, Main	302	Database	Steeb	102
101	Smith	1, Main	303	C++	Hardy	102
101	Smith	1, Main	304	Java	Solms	105
105	Jones	12, 7th	302	Database	Steeb	102

The above table satisfies the properties of a relation and is said to be in first normal form (or 1NF). Conceptually it is convenient to have all the informationin one relation since it is then likely to be easier to query the database. However the table has the following undesirable features.

1. Repetition of information – A lot of information is being repeated. Student name, address, course name, instructor name and office number of the instructors are repeated often. Every time we wish to insert a student enrolment, say, for cno 302 we must insert the name of the course cno 302 as well as the name and office number of its instructor. Also every time we insert a new enrolment for, say Smith, we must repeat his name and address. Repetition of information results in wastage of storage as well as other problems.

2. Update Anomalies – Redundant information not only wastes storage but makes updates more difficult since, for example, changing the name of the instructor of cno 302 would require that all tuples containing cno 302 enrolment information be updated. If for some reason, all tuples are not updated, we might have a database that gives two names of instructor for subject cno 302. This difficulty is called the update anomaly.

3. Insertional anomalies – Inability to represent certain information – Let the primary key of the above table be `(sno,cno)`. Any new tuple to be inserted in the relation must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL. However, if one wanted to insert the number and name of a new course in the database, it would not be possible until a student enrols in the course and we are able to insert values of `sno` and `cno`. Similarly information about a new student cannot be inserted in the table until the student enrols in a subject. These difficulties are called insertion anomalies.

4. Deletion anomalies – Loss of useful information – In some instances, useful information may be lost when a tuple is deleted. For example, if we delete the tuple corresponding to student 101 doing cno 304, we loose relevant information about course cno 304 (viz. course name, instructor, office number) if the student 101 was the only student enrolled in that course. Similarly deletion of course cno 302 from the table may remove all information about the student named `Jones`. This is called deletion anomalies.

The above problems are due to the fact that the table `Student` has information about students as well as subjects. One solution to deal with the problem is to decompose the relation into two or more smaller tables.

Decomposition may provide further benefits, for example, in a distributed database different tables may be stored at different sites if necessary. Of course, decomposition does increase the cost of query processing since the decomposed relations will need to be joined in some cases.

The above table may be easily decomposed into three tables to remove most of the above undesirable properties:

```
table Student(sno,sname,address)
table Course(cno,cname,instructor,office)
table Student_Course(sno,cno)
```

Such a decomposition is called *normalization*.

3.3 Example

We consider the following table, named ENROLL, that contains the data fields (attributes) required to enroll us in a class at a University.

```
ENROLL(Class_Code,Class_Description,Student_Number,Student_Name,
       Address,City,State,Zip,Major_Code,Major_Description,Course_Grade,
       Class_Start_Date,Class_Start_Time,Class_Room_Number,
       Building_Number,Building_Name,Building_Address,
       Professor_Number,Professor_Name)
```

An object is said to be in 1NF if there are no repeating groups of attributes (fields). This object is said to be in First Normal Form (1NF) if it is in the format illustrated above with no "gaps" or repeating groups. It is simply a collection of data fields necessary to complete the job of enrolling, with each record in the file containing all data necessary for the enrollment. The problem with 1NF is that there is redundancy with respect to entering all of the data into a database for each and every class in which we enroll. For example, our name, address, etc., will have to be entered for each class that we take. If we take four classes, our name will have to be entered four times, not to mention the opportunities to incorrectly enter it. Developing a logical method of eliminating the entry of our name four times leads us to the definition of what is called Second Normal Form (2NF).

We must next introduce the concept of a (primary) key field. A key field is one (or more logically joined) field(s) that is used to uniquely identify each record in a data file. For example, the Student_Number field can be used to uniquely identify each student's record in a student data file. However, since one student may be enrolled in more than one class each quarter, the Student_Number field alone is not sufficient to uniquely identify each record in the ENROLL table illustrated above. The combination of the Student_Number field and the Class_Code field forms a unique combination and can therefore be considered as the key field for the ENROLL table.

A relation is in 2NF if, and only if, it is in 1NF and every nonkey attribute (field) is fully functionally dependent upon the key field. This means that all data attributes (fields) that are not used to uniquely identify records (tuples or rows) in a table should not appear more than once in the entire database and should never have to be entered into the database more than once. Any non-identifying data fields should be placed into separate objects (files). For example, we could remove the name, address, etc. fields into an table named STUDENT and remove them from the ENROLL table. The result yields two tables:

```
STUDENT(Student_Number,Student_Name,Address,City,State,Zip,Major_Code)

ENROLL(Student_Number,Class_Code,Major_Description,Class_Description,
       Course_Grade,Class_Start_Date,Class_Start_Time,Class_Room_Number,
       Building_Number,Building_Name,Building_Address,
       Professor_Code,Professor_Name)
```

Here we see that the Student_Name, Address, etc., are functionally dependent upon the Student_Number in the STUDENT table. The class description, Class_Start_Date, Building_Name, etc., are functionally dependent upon the Student_Number and the Class_Code in the ENROLL table. The relation between these objects (files) is said to be in 2NF. The relation is the logical linkage between the files so that all data necessary to enroll students in classes is available and may be uniquely retrieved when necessary.

While getting the tables into 2NF is better than 1NF, there are still some problems with the form. For example, if the location of the class changes buildings, all records in the ENROLL table for that class will have to be updated. The Building_Name and Address are *transitively dependent* upon the Building_Number. Resolving the *transitive dependency* leads us to Third Normal Form (3NF).

A relation is in 3NF if, and only if, it is in 2NF and no nonkey fields are transitively dependent upon the key field(s). That is, no nonkey field can be functionally dependent upon another nonkey field. Our example is clearly not in 3NF since the building name (nonkey field) depends upon the building number (nonkey field). The relation can be resolved into 3NF by dividing it into component relations, each meeting 3NF form. We also have recognized that the class description, start time, and start date are transitively dependent upon the class code, which is not considered a key field here because it forms only part of the key field for the ENROLL object. We also recognize that Professor_Name is functionally dependent upon the Professor_Code, which is not a key field. The Building_Code and Professor_Code fields are not key fields because they are not used to uniquely identify each record in the ENROLL table. Thus we end up with the following tables:

```
ENROLL(Student_Number,Class_Code,Course_Grade)

BUILDING(Building_Number,Building_Name,Building_Address)

CLASS(Class_Code,Class_Description,Class_Start_Date,Class_Start_Time,
      Class_Room_Number,Building_Number,Professor_Code)

PROFESSOR(Professor_Code,Professor_Name,Department_Code,Department_Name)

MAJOR(Major_Code,Major_Description)

STUDENT(Student_Number,Student_Name,Address,City,State,Zip,Major_Code)
```

Note also that the PROFESSOR table is not in 3NF since the Department_Name is transitively dependent upon the Department_Code. We resolve this by splitting the table Professor into two tables

```
PROFESSOR(Professor_Code,Professor_Name,Department_Code)
```

```
DEPARTMENT(Department_Code,Department_Name)
```

The primary key in the tables given above are

```
BUILDING   Building_Number
CLASS      Class_Code
PROFESSOR  Professor_Code
MAJOR      Major_Code
STUDENT    Student_Number
DEPARTMENT Department_Code
```

For the `ENROLL` table the pair `Student_Number` and `Class_Code` is the primar key.

This illustrates that we must consider all relationships within the organization's database and resolve all relations into 3NF. An important point here is that no data may be lost during the normalization process. We must always be able to reconstruct the original data after the normalization. To lose data will cause problems and will be an invalid normalization process.

Foreign Keys and Permitted RDBMS Operations

The concept of *Foreign Keys* must be understood before we can logically utilize a database that is implemented within a RDBMS. Specific RDBMS operations include adding, updating, and deleting data from files. The foreign keys specify the relationships between files and define what can and can't be logically done in order to maintain what is called *referential integrity*. An table that has a part of a key field (one field in a compound field key) that refers to a complete key in another table is said to have a foreign key. For example, consider the tables

```
ENROLL(Student_Number,Course_Code,Course_Grade)

STUDENT(Student_Number,Student_Name, etc.)

COURSE(Course_Code,Course_Description, etc.)
```

In the `ENROLL` object, the `Course_Code` is a foreign key because it is a part of the key field and referes to the entire key field for the `COURSE` table. Similarly, the `Student_Number` is a foreign key since it refers to the entire key field for the `STUDENT` table. The consequences of these relationships is that the `ENROLL` table and all of its data cannot validly exist without corresponding data in both the `STUDENT` and the `COURSE` tables. For example, consider the enrollment of John Smith, student number 12234, into the Java class, code 503. We cannot validly enroll John unless his student number (and all dependent data) have first been established in the `STUDENT` table and the `Course_code` (and all dependent data) have been established in the `COURSE` table. In essence, the valid enrollment of John Smith depends upon the data in the `STUDENT` and `COURSE` tables.

If we were permitted to establish a record in the `ENROLL` table with John Smith's `Student_Number` and the `Course_Code`, we then want to see who was enrolled and in what course, it would be impossible. The `Student_Number` in the `ENROLL` table has no corresponding record in the `STUDENT` table and therefore cannot provide further information about John Smith. Similarly, the `Course_Code` in the `ENROLL` table has no corresponding record in the `COURSE` table and cannot provide further information. Permitting the entry of the data into the `ENROLL` table would create what is called a *parentless child*. The "child", the record in the `ENROLL` table, has no "parents" to provide the required additional data to complete the enrollment. Tables having foreign keys are said to be "members" of a database. Tables to which the foreign keys refer are said to be *owners* - the owners of the key fields to which the foreign keys refer. Why are foreign keys, relationships, members, and owners important? They establish specific requirements for permitted database operations in order to maintain referential integrity.

3.4 Fourth and Fifth Normal Forms

Fourth and fifth normal forms deal with multi-valued facts. The multi-valued fact may correspond to a many-to-many relationship, for example employees and skills, or to a many-to-one relationship, as with the children of an employee (assuming only one parent is an employee). By many-to-many we mean that an employee may have several skills, and a skill may belong to several employees. We look at the many-to-one relationship between children and fathers as a single-valued fact about a child but a multi-valued fact about a father. In a sense, fourth and fifth normal forms are also about composite keys. These normal forms attempt to minimize the number of fields involved in a composite key.

Under fourth normal form, a table should not contain two or more independent multi-valued facts about an entity. In addtion, the table must satisfy third normal form.

As an example, consider employees, skills and speaking languages. Thus an employee may have several skills and speak several languages. We have here two many-to-many relationships, one between employees and skills, and one between employees and languages. Under fourth normal form, these two relationships should not be represented in a single table such as

```
Employee   Skills   Language
========   ======   ========

============================
```

Instead they should be represented in two tables

```
Employee  Skills
========  ======

================
```

and

```
Employee  Language
========  ========

==================
```

Note that other fields, not involving multi-valued facts, are permitted to occur in the table. The main problem with violating fourth normal form is that it leads to uncertainties in the maintenance policies. Several policies are possible for maintaining two independent multi-valued facts in one table.

(1) A disjoint format, in which a record contains either a skill or a language, but not both:

```
Employee    Skill  Language
==========  =====  ========
Smith       cook
Smith       type
Smith              French
Smith              German
Smith              Greek
==========================
```

This is not much different from maintaining two separate record types. Such a format also leads to ambiguities regarding the meanings of blank fields. A blank `SKILL` could mean the person has no skill, or the field is not applicable to this employee, or the data is unknown, or, as in this case, the data may be found in another record.

(2) A random mix, with three variations:

(a) Minimal number of records, with repetitions:

```
Employee   Skill   Language
========   =====   ========
Smith      cook    French
Smith      type    German
Smith      type    Greek
==========================
```

(b) Minimal number of records, with null values:

```
Employee   Skill   Language
========   =====   ========
Smith      cook    French
Smith      type    German
Smith              Greek
============================
```

(c) Unrestricted:

```
Employee  Skill  Language
========  =====  ========
Smith     cook   French
Smith     type
Smith            German
Smith     type   Greek
=======   =====  ========
```

(3) A cross-product form, where for each employee, there must be a record for every possible pairing of one of his skills with one of his languages:

Employee	Skill	Language
Smith	cook	French
Smith	cook	German
Smith	cook	Greek
Smith	type	French
Smith	type	German
Smith	type	Greek

Other problems caused by violating fourth normal form are similar in spirit to those mentioned earlier for violations of second or third normal form. They take different variations depending on the chosen maintenance policy:

If there are repetitions, then updates have to be done in multiple records, and they could become inconsistent.

Insertion of a new skill may involve looking for a record with a blank skill, or inserting a new record with a possibly blank language, or inserting multiple records pairing the new skill with some or all of the languages.

Deletion of a skill may involve blanking out the skill field in one or more records (perhaps with a check that this doesn't leave two records with the same language and a blank skill), or deleting one or more records, coupled with a check that the last mention of some language hasn't also been deleted.

Fourth normal form minimizes such update problems.

We mentioned independent multi-valued facts earlier, and we now illustrate what we mean in terms of the example. The two many-to-many relationships,

`Employee:Skill`

and

`Employee:Language`

are independent in that there is no direct connection between skills and languages. There is only an indirect connection because they belong to some common employee. That is, it does not matter which skill is paired with which language in a record; the pairing does not convey any information. That's precisely why all the maintenance policies mentioned earlier can be allowed.

In contrast, suppose that an employee could only exercise certain skills in certain languages. Perhaps Smith can cook French cuisine only, but can type in French, German, and Greek. Then the pairings of skills and languages becomes meaningful, and there is no longer an ambiguity of maintenance policies. In the present case, only the following form is correct:

```
 Employee   Skill   Language
=========   =====   ========
 Smith      cook    French
 Smith      type    French
 Smith      type    German
 Smith      type    Greek
============================
```

Thus the `employee:skill` and `employee:language` relationships are no longer independent. These records do not violate fourth normal form. When there is an interdependence among the relationships, then it is acceptable to represent them in a single record.

Multivalued Dependencies

Fourth normal form is defined in terms of multivalued dependencies, which correspond to our independent multi-valued facts. Multivalued dependencies, in turn, are defined essentially as relationships which accept the cross-product maintenance policy mentioned above. That is, for our example, every one of an employee's skills must appear paired with every one of his languages. This is equivalent to our notion of independence: since every possible pairing must be present, there is no information in the pairings. Such pairings convey information only if some of them can be absent, that is, only if it is possible that some employee cannot perform some skill in some language. If all pairings are always present, then the relationships are really independent.

We should also point out that multivalued dependencies and fourth normal form apply as well to relationships involving more than two fields. For example, suppose we extend the earlier example to include projects, in the following sense:

An employee uses certain skills on certain projects.

An employee uses certain languages on certain projects.

If there is no direct connection between the skills and languages that an employee uses on a project, then we could treat this as two independent many-to-many relationships of the form EP:S and EP:L, where EP represents a combination of an employee with a project. A record including employee, project, skill, and language would violate fourth normal form. Two records, containing fields E,P,S and E,P,L, respectively, would satisfy fourth normal form.

Fifth Normal Form

Fifth normal form deals with cases where information can be reconstructed from smaller pieces of information that can be maintained with less redundancy. Second, third, and fourth normal forms also serve this purpose, but fifth normal form generalizes to cases not covered by the others.

We illustrate the central concept with a commonly used example, namely one involving agents, companies, and products. If agents represent companies, companies make products, and agents sell products, then we might want to keep a record of which agent sells which product for which company. This information could be kept in one record type with three fields:

Agent	Company	Product
Smith	Ford	car
Smith	GM	truck

This form is necessary in the general case. For example, although agent Smith sells cars made by Ford and trucks made by GM, he does not sell Ford trucks or GM cars. Thus we need the combination of three fields to know which combinations are valid and which are not. But suppose that a certain rule was in effect: if an agent sells a certain product, and he represents a company making that product, then he sells that product for that company.

Agent	Company	Product
Smith	Ford	car
Smith	Ford	truck
Smith	GM	car
Smith	GM	truck
Jones	Daimler	car

In this case, we can reconstruct all the true facts from a normalized form consisting of three separate record types, each containing two fields:

Agent	Company
Smith	Ford
Smith	GM
Jones	Daimler

Company	Product
Ford	car
Ford	truck
GM	car
GM	truck
Daimler	car

Agent	Product
Smith	car
Smith	truck
Jones	car

These three record types are in fifth normal form, whereas the corresponding three-field record shown previously is not. Roughly speaking, we may say that a record type is in fifth normal form when its information content cannot be reconstructed from several smaller record types, i.e., from record types each having fewer fields than the original record. The case where all the smaller records have the same key is excluded. If a record type can only be decomposed into smaller records which all have the same key, then the record type is considered to be in fifth normal form without decomposition. A record type in fifth normal form is also in fourth, third, second, and first normal forms.

Fifth normal form does not differ from fourth normal form unless there exists a symmetric constraint such as the rule about agents, companies, and products. In the absence of such a constraint, a record type in fourth normal form is always in fifth normal form.

One advantage of fifth normal form is that certain redundancies can be eliminated. In the normalized form, the fact that Smith sells cars is recorded only once; in the unnormalized form it may be repeated many times.

It should be observed that although the normalized form involves more record types, there may be fewer total record occurrences. This is not apparent when there are only a few facts to record, as in the example shown above. The advantage is realized as more facts are recorded, since the size of the normalized files increases in an additive fashion, while the size of the unnormalized file increases in a multiplicative fashion. For example, if we add a new agent who sells x products for y companies, where each of these companies makes each of these products, we have to add x+y new records to the normalized form, but xy new records to the unnormalized form.

All three record types are required in the normalized form in order to reconstruct the same information. From the first two record types shown above we learn that Jones represents Ford and that Ford makes trucks. But we can't determine whether Jones sells Ford trucks until we look at the third record type to determine whether Jones sells trucks at all.

The following example illustrates a case in which the rule about agents, companies, and products is satisfied, and which clearly requires all three record types in the normalized form. Any two of the record types taken alone will imply something untrue.

Agent	Company	Product
Smith	Ford	car
Smith	Ford	truck
Smith	GM	car
Smith	GM	truck
Jones	Ford	car
Jones	Ford	truck
Brown	Ford	car
Brown	GM	car
Brown	Daimler	car
Brown	Daimler	bus

Thus the fifth normal form is

Agent	Company	Company	Product	Agent	Product
Smith	Ford	Ford	car	Smith	car
Smith	GM	Ford	truck	Smith	truck
Jones	Ford	GM	car	Jones	car
Brown	Ford	GM	truck	Jones	truck
Brown	GM	Daimler	car	Brown	car
Brown	Daimler	Daimler	bus	Brown	bus

Thus Jones sells cars and GM makes cars, but Jones does not represent GM. Brown represents Ford and Ford makes trucks, but Brown does not sell trucks. Brown represents Ford and Daimler and Brown sells buses and cars, but Ford does not make buses.

Fourth and fifth normal forms both deal with combinations of multivalued facts. One difference is that the facts dealt with under fifth normal form are not independent, in the sense discussed earlier. Another difference is that, although fourth normal form can deal with more than two multivalued facts, it only recognizes them in pairwise groups. We can best explain this in terms of the normalization process implied by fourth normal form. If a record violates fourth normal form, the associated normalization process decomposes it into two records, each containing fewer fields than the original record. Any of these violating fourth normal form is again decomposed into two records, and so on until the resulting records are all in fourth normal form. At each stage, the set of records after decomposition contains exactly

the same information as the set of records before decomposition.

In the present example, no pairwise decomposition is possible. There is no combination of two smaller records which contains the same total information as the original record. All three of the smaller records are needed. Hence an information-preserving pairwise decomposition is not possible, and the original record is not in violation of fourth normal form. Fifth normal form is needed in order to deal with the redundancies in this case.

Normalization certainly doesn't remove all redundancies. Certain redundancies seem to be unavoidable, particularly when several multivalued facts are dependent rather than independent. In the example given above, it seems unavoidable that we record the fact that Smith can type; several times. Also, when the rule about agents, companies, and products is not in effect, it seems unavoidable that we record the fact that Smith sells cars several times.

The normal forms discussed here deal only with redundancies occurring within a single record type. Fifth normal form is considered to be the ultimate normal form with respect to such redundancies.

Other redundancies can occur across multiple record types. For the example concerning employees, departments, and locations, the following records are in third normal form in spite of the obvious redundancy:

```
Employee  Department    Department  Location     Employee  Location
========  ==========    ==========  ========     ========  ========

====================    ====================     ==================
```

In fact, two copies of the same record type would constitute the ultimate in this kind of undetected redundancy. Inter-record redundancy has been recognized for some time, and has recently been addressed in terms of normal forms and normalization.

The factors affecting normalization have to be assessed:

Single-valued vs. multi-valued facts.
Dependency on the entire key.
Independent vs. dependent facts.
The presence of mutual constraints.
The presence of non-unique or non-singular representations.

And, finally, the desirability of normalization has to be assessed, in terms of its performance impact on retrieval applications.

Chapter 4

Transaction

4.1 Introduction

A *transaction* is defined as a logical unit of work - a set of database changes to one or more tables that accomplishes a defined task. In other wordsa transaction is a group of database actions that are performed together. Either all the actions succeed together or all fail together.

A transaction begins after a `COMMIT` statement, a `ROLLBACK` statement, or an initial database connection. A transaction ends when any of the following events occurs.

1) A `COMMIT` statement is processed

2) A `ROLLBACK` statement is processed

3) The database connection is terminated

The `COMMIT` statement commits a transaction. It makes permanent all the database changes made since the execution of the previous `COMMIT` (or `ROLLBACK`). We can `COMMIT` only the database changes that we personally have made. The `COMMIT` statement that one user issues has no effect on another user's database changes.

The `ROLLBACK` statement will undo all database changes made by the user since the last committed transaction or since the beginning of the session.

For example, JDBC supports transaction processing with the `commit()` and `rollback()` methods. Also, JDBC has the `autocommit()` which, when on, all changes are committed automatically and, if off, the Java program has to use the `commit()` or `rollback()` methods to effect the changes to the data.

Example. Consider the `Cars` table and the following session (Oracle)

```
SELECT * FROM Cars;
```

The whole table consisting of two rows is displayed.

```
DELETE FROM Cars;
```

The output is: `2 rows deleted`.

```
SELECT * FROM Cars;
```

The output is: `No rows selected`

```
ROLLBACK;
```

The output is: `Rollback complete`

```
SELECT * FROM Cars;
```

The whole table is displayed again.

The rows that satisfy the conditions of a query are called the *result set*. The number of rows returned in a result set can be zero, one, or many or even the whole table. One accesses the data in a result set one row at the time, and a *cursor* provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows one to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

For transactions that involve the execution of multiple SQL statements, we might want to consider using *savepoints* as intermediate steps for the transaction. A savepoint is a label within a transaction that contains a subset of the changes made by the transaction. We can think of a savepoint as a label within a transaction that references a subset of a transaction's changes. The syntax is

```
SAVEPOINT name-of-savepoint
```

In MySQL we can choose between three basic formats for tables (ISAM, HEAP, and MyISAM). The default table type in MySQL is MyISAM. These are non-transaction-safe tables. MySQL also supports BDB (Berkely Database) and InnoDB. These are transaction-safe tables. For transaction-safe tables we can execute `ROLLBACK`. When we create a table in MySQL we can provide the type. For example,

```
CREATE TABLE person (name varchar(10)) type = bdb;
```

The concept of transactions is an integral part of any client/server database. A transaction is a group of SQL statements that update, add, and delete rows and fields in a database. Transactions have an all or nothing property-either they are committed if all statements are successful, or the whole transaction is rolled back if any of the statements cannot be executed successfully. Transaction processing assures the data integrity and data consistency in a database.

The characteristics of a transaction are described in terms of the

Atomicity, *Consistency*, *Isolation*, and *Durability* (ACID)

properties.

1) A transaction is atomic in the sense that it is an entity. All the components of a transaction happen or do not happen. There is no partial transaction. If only a partial transaction can happen, then the transaction is aborted. The atomicity is achieved by the `commit()` or `rollback()` methods. Thus a transaction must be an all-or-nothing proposition. Everything must be updated successfully or nothing should be updated.

2) A transaction is consistent because it does not perform any actions that violate the business logic or relationships between data elements. The consistent property of a transaction is very important when we develop a client/server system, because there will be many transactions to a data store from different systems and objects. If a transaction leaves the data store inconsistent, all other transactions also would potentially be wrong, resulting in a system-wide crash or data corruption. Thus individual operations within a transaction may leave data in such a state that it violates the system's integrity constraints. Before a transaction completes, the system's data as whole must be returned to a valid state.

3) A transaction is isolated because the results of a transaction are self-contained. They do not depend on any preceding or succeeding transaction. This is related to a property called *serializability*, which means the sequence of transactions are independent; in other words, a transaction does not assume any external sequence. Thus the system must hide the uncommited changes of a transaction from all the other transactions. The act of hiding or isolating changes is typically accomplished through locking.

4) Finally, a transaction is durable, meaning the effects of a transaction are permanent even in the face of a system failure. That means some form of permanent storage should be a part of a transaction. Thus when a transaction is committed, the data sources involved must keep all changes in stable storage and these changes must be recovered in the event of a system failure.

A related topic in transactions is the coordination of transactions across heterogeneous data sources, systems, and objects. When the transactions are carried out in one relational database, we can use the `commit()`, `rollback()`, `beginTransaction()`, and `endTransaction()` statements to coordinate the process. But what if we have diversified systems participating in a transaction? How do we handle such a system?

As an example, look at the Distributed Transaction Coordinator (DTC) available as a part of Microsoft SQL Server 6.5 database system.

In the Microsoft DTC, a transaction manager facilitates the coordination. Resource managers are clients that implement resources to be protected by transactions-for example, relational databases and ODBC data sources.

An application begins a transaction with the transaction manager, and then starts transactions with the resource managers, registering the steps (enlisting) with the transaction manager. The transaction manager keeps track of all enlisted transactions. The application, at the end of the multi-data source transaction steps, calls the transaction manager to either commit or abort the transaction.

When an application issues a `COMMIT` command to the transaction manager, the DTC performs a two-phase commit protocol:

It queries each resource manager if it is prepared to commit. If all resources are prepared to commit, DTC broadcasts a commit message to all of them.

The Microsoft DTC is an example of powerful next generation transaction coordinators from the database vendors. As more and more multi-platform, object-oriented Java systems are being developed, this type of transaction coordinators will gain importance. Many middleware vendors are developing Java-oriented transaction systems.

A *deadlock* is a condition that can occur when two or more users are all waiting for each other to give up locks. More advanced DBMSs do deadlock detection and abort one of the user transactions when this happens.

A *dirty read* happens when a transaction reads data from a database that has been modified by another transaction, and that data has not yet been committed. If dirty reads are not desired, one must specify a higher isolation level possibly resulting in lower performance.

A *phantom read* occurs when our program fetches a tuples that has been inserted by another user's transaction, and the other transaction subsequently aborts, erasing the tuple we fetched.

As an example let us consider three cases:

Case 1:

S1

T1	T2
read A	
A=A-10	
write A	
read B	
B=B+10	
write B	
	read B
	B=B-20
	write B
	read C
	C=C+20
	write C

S1 is serial.

Case 2:

S2

T1	T2
read A	
	read B
A=A-10	
	B=B-20
write A	
	write B
read B	
	read C
B=B+10	
	C=C+20
write B	
	write C

S2 is not serial but can be serialized.

Case 3:

S3

T1	T2
read A	
A=A-10	
	read B
write A	
	B=B-20
read B	
	write B
B=B+10	
	read C
write B	
	C=C+20
	write C

t

S3 cannot be serialized (transaction with conflict).

4.2 Data Replication

Many situations occur in the daily operation of an organization that involves the need to have the same information in more than one loaction.

Data replication is the distribution of corporate data to many locations across the organization, and it provides reliability, fault-tolerance, data-access performance due to reduced communication, and, in many cases, manageability as the data can be managed as subsets. Put simply, data replication is a process that automatically copies information from one database to one or more additional databases. The client/server systems span an organization, possibly its clients and suppliers, most probably in a wide geographic locations. Systems spanning the entire globe are not uncommon when we are talking about mission-critical applications, especially in today's global business market. If all the data is concentrated in a central location, it would be almost impossible for the systems to effectively access data and offer high performance. Also, if data is centrally located, in the case of mission-critical systems, a single failure will bring the whole business down. Using replicated data across an organization at various geographic locations is a sound strategy.

Different vendors handle replication differently.

For example, the Lotus Notes group-ware product uses a replication scheme where the databases are considered peers, and additions/updates/deletions are passed between the databases. Lotus Notes has replication formulas that can select subsets of data to be replicated based on various criteria.

The Microsoft SQL server, on the other hand, employs a publisher-subscriber scheme where a database or part of a database can be published to many subscribers. A database can be a publisher and a subscriber. For example, the western region can publish its slice of sales data while receiving (subscribing to) sales data from other regions.

There are many other replication schemes from various vendors to manage and decentralize data. Replication is a young technology that is slowly finding its way into many other products.

4.3 Locks

When one is accessing data in a database, someone else may be accessing the same data at the same time. If, for instance, one user is updating some columns in a table at the same time another user is selecting columns from that table, it could be possible that the data returned is partly the old data and partly updated data. For this reason, DBMSs use transaction to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency). A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a `COMMIT` or a `ROLLBACK`, depending on whether there are any problems with data consistency or data concurrency. The `COMMIT` statement makes permanent the changes resulting from the SQL statements in the transaction, and a `ROLLBACK` statement undoes all changes resulting from the SQL statements in the transaction.

A *lock* is a mechanism that prohibits two transactions from manipulation the same data at the same time. For example, a *table lock* prevents a table from being dropped if there is an uncommited transaction on that table. A *row lock* prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

For example, even though Oracle provides consistency within a single SQL statement, its default behaviour does not guarantee read consistency during more than one statement. If we query a table twice, we may obtain different results the second time if another Oracle user changes the table between our first and second queries. We may encounter a situation in which we need more than single-statement read consistency. In fact, we may need to have read consistency across a particular transaction. For this purpose, we need to issue the following statement

```
SET TRANSACTION READ ONLY;
```

The SQL Server of Microsoft lock manager uses two primary lock types: *write locks* and *read locks*. SQL Server uses write locks (also called exclusive locks) on data items to isolate the uncommitted changes of a transaction. SQL Server places read locks (also called shared locks) on data items while they are being read. A write lock conflicts with other write locks and with read locks. A transaction that has a write lock blocks all other transactions from reading or writing to the data item in question. The data item remains locked until the transaction is committed or rolled back. This makes sense because the system must isolate uncommitted changes to ensure data consistency. But this isolation has a price: the blocking reduces overall system concurrency and throughput. Read locks do not conflict with other read locks. Many transactions can obtain a read lock on the same data item concurrently. A transaction cannot obtain a write lock on a data item that has outstanding read locks. This ensures that a transaction does not overwrite a data item while another transaction is reading it.

Read locks are shared, i.e. there can be many read locks on the same data item. Write locks are exclusive, i.e. no other locks (read or write) can be held. The transaction manager maintains a lock table, and delays operations that are blocked by locks. Thus data integrity is ensured through record locking.

Deadlocking occurs when two user processes have locks on separate objects and each process is trying to acquire a lock on the object that the other process has. When this happens a database system ends the deadlock by automatically choosing one and aborting the process, allowing the other process to continue. The aborted transaction is rolled back and an error message is sent to the user of the aborted process.

For example, in mySQL we have the command `LOCK TABLES/UNLOCK TABLES`. The syntax is

```
LOCK TABLES table_name [AS alias]
             {READ | [READ_LOCAL] | [LOW_PRIORITY] WRITE}
             [, table_name {READ | [LOW_PRIORITY] WRITE} ...]
...
UNLOCK TABLES
```

`LOCK TABLES` locks tables for the current thread. `UNLOCK TABLES` releases any locks held by the current thread. All tables that are locked by the current thread are automatically unlocked when the thread issues another `LOCK TABLES`, or when the connection to the server is closed. If a thread obtains a `READ` lock on the table, that thread (all other threads) can only read from the table. If a thread obtains a `WRITE` lock on a table, then only the thread holding the lock can `READ` from or `WRITE` to the table. Other threads are blocked.

The difference between `READ_LOCAL` and `READ` is that `READ_LOCAL` allows non conflicting `INSERT` statements to execute while the lock is hold.

Each thread waits (without timing out) until it obtains all the locks it has requested.

`WRITE` locks normally have higher priority than `READ` locks, to ensure that updates are processed as soon as possible. This means that if one thread obtains a `READ` lock and then another thread requests a `WRITE` lock, subsequent `READ` lock requests will wait until the `WRITE` thread has gotten the lock and released it. We can use `LOW_PRIORITY WRITE` locks to allow other threads to obtain `READ` locks while the thread is waiting for the `WRITE` lock. We should only use `LOW_PRIORITY WRITE` locks if we are sure that there will eventually be a time when no threads will have a `READ` lock.

When we are using `LOCK TABLES`, we must lock all tables that we are going to use and we must use the same alias as we are going to use in our queries. If we are using a table multiple times in a query (with aliases), we must get a lock for each alias. This policy ensures that table locking is deadlock free.

Note that we should not lock any tables that we are using with `INSERT DELAYED`. This is because that in this case the `INSERT` is done by a separate thread.

Normally, we do not have to lock tables, as all single `UPDATE` statements are atomic; no other thread can interfere with any other currently executing SQL statement. There are a few cases when we would like to lock tables anyway:

1) If we are going to run many operations on a bunch of tables, it is must faster to lock the tables we are going to use. The downside is, of course, that no other thread can update a `READ`-locked table and no other thread can read a `WRITE`-locked table.

2) `mySQL` does not support a transaction enviroment, so we must use `LOCK TABLES` if we want to ensure that no other thread comes between a `SELECT` and an `UPDATE`. The example shown below require `LOCK TABLES` in order to execute safely:

```
> LOCK TABLES trans READ, customer WRITE;
> SELECT sum(value) FROM trans WHERE customer_id = some_id;
> UPDATE customer SET total_value = sum_from_previous_statement
      WHERE customer_id = some_id;
> UNLOCK TABLES
```

By using incremental updates

```
UPDATE customer SET value = value + new_value
```

or the `LAST_INSERT_ID()` function, we can avoid using `LOCK TABLES` in many cases. We can also solve some cases by using the user-level lock functions `GET_LOCK()` and `RELEASE_LOCK()`. These locks are saved in a hash table in the server and implemented with the

```
pthread_mutex_lock()
```

and

```
pthread_mutex_unlock()
```

for high speed.

4.4 Deadlocking

Deadlocking occurs when two user processes have locks on separate objects and each process is trying to acquire a lock on the object that the process has. An example of deadlock in a computer system is: Consider two processes p_1 and p_2. They update a file F and require a CD during the updating. Only one CD is available. CD and F are serially reuseable resources, and can be used only by exclusive access. p_2 needs CD immediately prior to updating. p_1 can block on CD holding F while p_2 can block on F holding CD.

When deadlocking occurs, Microsoft SQL Server ends the deadlock by automatically choosing one and aborting the other process, allowing the chosen process to continue. The aborted transaction is rolled back and an error message is sent to the user of the aborted process.

Most well-designed applications, after receiving this message, will resubmit the transaction, which most likely can now run successfully. This process, if it happens often on our server, can drag down performance.

Here are some tips on how to avoid deadlocking on a SQL Server.

1) Keep All Transact-SQL transactions as short as possible. This helps to reduce the number of locks (of all types), helping to speed up the overall performance of our SQL Server application. If practical, we may want to break down long transactions into groups of smaller transactions.

2) An often overlooked cause of locking is an I/O bottleneck. Whenever our server experiences an I/O bottleneck, the longer it takes user's transactions to complete. And the longer they take to complete, the longer locks must be held, which can lead to other transactions being prevented from processing because they have to wait for previous locks to be released.

3) If our server is experiencing excessive locking problems, be sure to check if we are also running into an I/O bottleneck. If we do find that we have an I/O bottleneck, then resolving it will help to resolve our locking problem, greatly speeding up the performance of our server.

4) To help reduce the amount of time tables are locked, which hurts concurrency and performance, avoid interleaving reads and database changes within the same transaction. Instead, try to do all our reading first, then perform all of the database changes (UPDATES, INSERTS, DELETES) near the end of the transaction. This helps to minimize the amount of time that exclusive locks are held.

5) Any conditional logic, variable assignment, and other related preliminary setup should be done outside of transactions, not inside them. Don't ever pause a transac-

tion to wait for user input. User input should always be done outside of a transaction.

6) Encapsulate all transactions within stored procedures, including both the `BEGIN TRANSACTION` and `COMMIT TRANSACTION` statements in the procedure. This provides two benefits that help to reduce blocking locks. First, it limits the client application and SQL Server to communications before and after when the transaction runs, thus forcing any messages between them to occur at a time other than when the transaction is running (reducing transaction time). Second, it prevents the user from leaving an open transaction (holding locks open) because the stored procedure forces any transactions that it starts to complete or abort.

7) If we have a client application that needs to "check-out" data for awhile, then perhaps update it later, or maybe not, we don't want the records locked during the entire time the record is being viewed. Assuming "viewing" the data is much more common that "updating" the data, then one way to handle this particular circumstance is to have the application select the record (not using `UPDATE`, which will put a share lock on the record) and send it to the client.

8) If the user just "views" the record and never updates it, then nothing has to be done. But if the user decides to update the record, then the application can perform an `UPDATE` by adding a `WHERE` clause that checks to see whether the values in the current data are the same as those that were retrieved.

9) Similarly, we can check a timestamp column in the record, if it exists. If the data is the same, the the `UPDATE` can be made. If the record has changed, then the application must include code to notify the user so he or she can decide how to proceed. While this requires extra coding, it reduces locking and can increase overall application performance.

10) Use the least restrictive transaction isolation level possible for our user connection, instead of always using the default `READ COMMITTED`. In order to do this without causing other problems, the nature of the transaction must be carefully analyzed as to what the effect of a different isolation will be.

11) Using cursors can reduce concurrency. To help avoid this, use the `READ_ONLY` cursor option if applicable, or if we need to perform updates, try to use the `OPTIMISTIC` cursor option to reduce locking. Try to avoid the `SCROLL_LOCKS` cursor option, which can increase locking problems.

12) If our users are complaining that they have to wait for their transactions to complete, we may want to find out if object locking on the server is contributing to this problem. To do this, use the SQL Server Locks Object: Average Wait Time (ms). We can use this counter to measure the average wait time of a variety of locks, including: database, extent, Key, Page, RID, and table.

13) If we can identify one or more types of locks causing transaction delays, then we will want to investigate further to see if we can identify what specific transactions are causing the locking. The Profiler is the best tool for this detailed analysis.

14) Use `sp_who` and `sp_who2` to identify which users may be blocking what other users.

15) Try one or more of the following suggestions to help avoid blocking locks: 1) Use clustered indexes on heavily used tables; 2) Try to avoid Transact-SQL statements that affect large numbers of rows at once, especially the `INSERT` and `UPDATE` statements; 3) Try to have the `UPDATE` and `DELETE` statements use an index; and 4) When using nested transactions, avoid commit and rollback conflicts.

16) On tables that change little, if at all, such as lookup tables, consider altering the default lock level for the table. By default, SQL Server uses row level locking for all tables, unless the SQL Query Optimizer determines that a more appropriate locking level, such as page or table locks, is more appropriate. For most lookup tables that aren't huge, SQL Server will automatically use row level locking. Because row locking has to be done at the row level, SQL Server needs to work harder maintain row locks that it does for either page or table locks. Since lookup tables aren't being changed by users, it would be more efficient to use a table lock instead of many individual row locks. How do we accomplish this? We can override how SQL Server performs locking on a table by using the `SP_INDEXOPTION` command. Below is an example of code we can run to tell SQL Server to use page locking, not row locks, for a specific table:

```
SP_INDEXOPTION 'table_name', 'AllowRowLocks', FALSE
GO
SP_INDEXOPTION 'table_name', 'AllowPageLocks', FALSE
GO
```

This code turns off both row and page locking for the table, thus only table locking is available.

17) If there is a lot of contention for a particular table in our database, consider turning off page locking for that table, requiring SQL Server to use row level locking instead. This will help to reduce the contention for rows located on the same page. It will also cause SQL Server to work a little harder in order to track all of the row locks. How well this option will work for us depends on the tradeoff in performance between the contention and the extra work SQL Server has to perform. Testing will be needed to determine what is best for our particular environment. Use the `SP_INDEXOPTION` stored procedure to turn off page locking for any particular table.

18) If table scans are used regularly to access data in a table, and our table doesn't have any useful indexes to prevent this, then consider turning off both row locking

and page locking for that table. This in effect tells SQL Server to only use table locking when accessing this table. This will boost access to this table because SQL Server will not have to escalate from row locking, to page locking, to table locking each time a table lock is needed on the table to perform the table scan. On the negative side, doing this can increase contention for the table. Assuming the data in the table is mostly read only, then this should not be too much of a problem. Testing will be needed to determine what is best for our particular environment.

19) Do not create temporary tables from within a stored procedure that is invoked by the `INSERT INTO EXECUTE` statement. If we do, locks on the syscolumns, sysobjects, and sysindexes tables in the TEMPDB database will be created, blocking others from using the TEMPDB database, which can significantly affect performance.

20) To help reduce the amount of time it takes to complete a transaction (and thus reducing how long records are locked) try to avoid using the `WHILE` statement or Data Definition Language within a transaction. In addition, do not open a transaction while browsing data and don't `SELECT` more data than we need for the transaction at hand. For best performance, we always want to keep transactions as short as possible.

21) While nesting transactions is perfectly legal, it is not recommended because of its many pitfalls. If we nest transactions and our code fails to commit or roll back a transaction properly, it can hold locks open indefinitely, significantly impacting performance.

22) By default in SQL Server, a transaction will wait indefinitely for a lock to be removed before continuing. If we want, we can assign a locking timeout value to SQL Server so that long running locks won't cause other transactions to wait long periods of time. To assign a locking timeout value to SQL Server, run this command,

```
SET LOCK_TIMEOUT length_of_time_in_milliseconds
```

from Query Analyzer.

23) Sometimes we need to perform a mass `INSERT` or `UPDATE` of thousands, if not millions of rows. Depending on what we are doing, this could take some time. Unfortunately, performing such an operation could cause locking problems for our other users. If we know users could be affected by our long-running operation, consider breaking up the job into smaller batches, perhaps even with a `WAITFOR` statement, in order to allow others to "sneak" in and get some of their work done.

24) One way to reduce locking, especially for queries or stored procedures that are used to create reports, is to force SQL Server to turn off shared locking when the query or stored procedure is run. By default, SQL Server will use shared locks on

any rows that are queried by a query or stored procedure. The purpose of these shared locks is to prevent any user from changing any of the affected rows while the transaction is still running. Assuming we don't care if any of the rows we are querying change during the query itself (which is common for many reports), we can turn off shared locking, which in turn increases concurrently in the database and can boost performance. Of course, if we require our data not to change while our transaction is still running, then we cannot take advantage of this technique.

25) One of the easiest ways to turn of shared locks during a query or stored procedure is to use the `NOLOCK` hint. This hint, when used with the `SELECT` statement, tells SQL Server not to issue any shared locks on the table(s) being read. This hint is only good for the query or stored procedure we give it to. It does not affect any other queries or stored procedures. Before using this technique, be sure we test it thoroughly to ensure that turning off sharing locks does not present we with any unexpected problems.

26) When a query affects many records in a table, SQL Server can automatically escalate locking from individual rows, to locking multiple pages or even an entire table. This happens because maintaining locks requires internal SQL Server resources, and it takes much less resources for SQL Server to lock pages rather than rows, or to lock an entire table instead of many different pages.

27) While it is good that SQL Server knows how to limit its use of lock resources, lock escalation can sometimes cause us problems. For example, if SQL Server escalates locking from rows to pages or to the entire table, this reduces concurrency, preventing users from accessing the data they need in a timely basis. Sometimes we may want to be able to control lock escalation in order to reduce concurrency problems.

28) There may be times when it is beneficial for our application, from a performance point of view, to actually escalate locks even before SQL Server decides it is a good idea. This may come in handy when we know that a specific type of lock will be needed, such as a page or table lock, and rather than having SQL Server waste's its time going through lock escalation, we can skip past this directly and go to the type of lock that is most appropriate for a specific query. This reduces concurrently, so we must consider this implication if we decide to escalate locks directly, rather than let SQL Server do it for us.

29) By default, SQL Server controls lock escalation, but we can control it ourself by using lock optimizer hints. Here are some lock escalation hints we may want to consider:

Rowlock: This hint tells SQL Server to use row-level locking instead of page locks for INSERTS. By default, SQL Server may perform a page-level lock instead of a less intrusive row-level lock when inserting data. By using this hint, we can tell SQL

Server to always start out using row-level locking. But, this hint does not prevent lock escalation if the number of locks exceeds SQL Server's lock threshold.

Pagelock: This hint forces SQL Server to use page-level locking no matter what. So SQL Server will not use either row-level or table-level locking if this hint is used.

Tablock: This hint forces SQL Server to use table-level locking only, and not to use either row-level or page-level locking.

30) If we decide to try one or more of these optimizer hints, keep in mind that using hints prevents SQL Server from figuring out what it thinks is best. If our data changes a lot, or if our queries or dynamic, using hints such as these may cause more problems than they cure. But if we know that a specific query has lock-related performance issues, and that the results of this query are predictable, then using one of these hints may be a good idea.

31) Sometimes it is beneficial for performance reasons to turn all locking off for a specific query. Normally, when a `SELECT` query runs, it places shared locks on each of the rows affected by the query as each row is read. Sometimes these shared row locks are escalated to page and tables locks, reducing concurrency in our database. If the query is long-running, it can prevent other users from accessing the rows they need to `UPDATE` or `DELETE` rows on a timely basis.

32) If we write queries that are used for reporting, we can often increase concurrently in our database by turning off locking for the specific query. While turning off all locking can result in "dirty reads", assuming that the report is used for seeing the "big picture" and does not have to reflect perfectly exact numbers, then turning off locking for the query is acceptable. Dirty reads occur when a query reads a row that is part of another transaction that isn't complete. So if the other transaction should be rolled back, then the data read by the original query for that row will be incorrect. In many cases, reports don't have to reflect perfect data, as "good enough" data is acceptable.

33) There are two ways to turn off all locking for a specific query or transaction. If we are writing a simple query that provides the data for a report, the easiest way to turn off locking is to use the `NOLOCK` optimizer hint as part of the query. If we are running a larger transactions that may include multiple queries, then we may consider setting the isolation level for the transaction to `READ UNCOMMITTED`. If we do this, we will have to turn it on, and then off, from within our transaction.

4.5 Threads

4.5.1 Introduction

Multitasking and *multithreading* are two types of concurrencies. Multitasking refers to the ability of executing more than one program at the time. It is usually supported on the operating system level. Multithreading, on the other hand, refers to a single pogram which has more than one execution thread running concurrently. Each of these independent subtasks is called a *thread*. An execution thread refers to the execution of a single sequence of instructions. In Java support for multithreaded programmming is built into the language.

Threads are sequences of code that can be executed independently alongside one another. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Thus a thread is an independent sequential flow of control within a process. Threads run with programs. A single application or applet can have many threads doing different things independently. The `Thread` class is defined in `java.lang` as a subclass of the `Object` class. To use Threads in a program, one needs to define a local subclass of the `Thread` class and therein override its

```
void run()
```

method. We put the code that we want the threads of that subclass to execute in that `void run()` method.

The `public abstract interface Runnable` should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run()`.

There are two ways in which to initiate a new thread of execution. The first is to create a subclass of the `Thread` class and redefine the `run()` method to perform the set of actions that the thread is intended to do. The second one is to define a new class that implements the `Runnable` interface. It must define the `run()` method and can be started by passing an instance of the class to a new `Thread` instance. In both cases we define a class which specifies the operations or actions that the new thread (of execution) performs.

A thread can be in any one of four states:

1) **New**: the thread object has been created but it has not been started yet so it cannot run.

2) **Runnable**: this means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running, but there is nothing to prevent it from being run if the scheduler can arrange it. It is not dead or blocked.

3) **Dead**: the normal way for a thread to die is by returning from its `run()` method.

4) **Blocked**: the thread could be run but there is something that prevents it. While a thread is in the blocked state the scheduler will simply skip over it and not give any CPU time. Until a thread re-enters the runnable state it will not perform any operations. A thread can be blocked for five reasons. First we have put the thread to sleep by calling the method

```
void sleep(milliseconds)
```

in which case it will not be run for the specified time. Second we have suspended the execution of the thread by calling the method `suspend()`. It will not become runnable again until the thread gets the `resume()` message. Note that both methods are deprecated. Third we have suspended the execution of the thread with the method `wait()`. The method

```
public final void wait() throws InteruptedException
```

waits to be notified by another thread of a change in this object. It will not become runnable again until the thread gets the `notify()` or `notifyAll()` message. The method

```
public final native void notify()
```

wakes up a single thread that is waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The method

```
public final native void notifyAll()
```

wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. Fourth the thread is waiting for some Input/Output to complete. Finally the thread is trying to call a `synchronized` method on another object and that object's lock is not available. The keyword `synchronized` indicates that while a method is accessing an object, other `synchronized` methods are blocked from accessing that object.

4.5.2 Thread Class

The `Thread` class provides all the facilities required to create a class which can execute within its own lightweight process (within the JVM). Next we give the methods that can change the state of a thread. The

```
void start()
```

method in class `Thread` causes this thread to begin execution. The Java Virtual Machine calls the `run` method of this thread. The `Runnable` interface has a single method called `run()`. The class which implements the `Runnable` interface must therefore supply its own `run()` method. Starting a thread causes the `run()` method to be executed. This method is executed for a brief time and then another thread of the application is executed. This thread runs briefly and is then suspended so another thread can run and so on. If this thread was constructed using a separate `Runnable` object, then that `Runnable` object's `run()` method is called; otherwise, this method does nothing and returns. The method

```
static void sleep(long millis)
```

in the `Thread` class causes the currently executing thread to sleep (temporely cease execution) for the specified number of milliseconds. The method

```
void yield()
```

causes the currentlty executing thread object to temporarily pause and allow other threads to execute. The method

```
void destroy()
```

destroys this thread without any cleanup.

There are a number of methods in the class `Thread` that provide information about the status of the process. The method

```
boolean isAlive()
```

tests to see if the thread has started execution. On the other hand the method

```
boolean isInterrupted()
```

tests if this thread has been interrupted.

Every thread has a unique name. If we do not provide one, then a name is automatically generated by the system. There are two methods that access and set a thread's name. The method

```
String getName() // in Thread class
```

returns this thread's name. The method

```
void setName(String name)
```

changes the name of this thread to be equal to the argument `name`.

The field

```
static int MAX_PRIORITY
```

is an integer which defines the maximum priority of a thread. The field

```
static int MIN_PRIORITY
```

is an integer which defines the minimum priority of a thread.

The methods

```
void resume(),   void stop(),   void suspend()
```

are deprecated. Instead of using the `stop()` method, it is recommended that threads monitor their execution and stop by returning from their `run()` method.

In many cases we call the method

```
void repaint()
```

inside the `run()` method. The method `repaint()` is in class `Component`. It repaints this component, i.e. it calls the method `paint()`. The method `paint()` is in class `Component`. It paints this component. The method

```
void repaint(long tm)
```

repaints the component. This will result in a call to `update()` within `tm` milliseconds, i.e. `tm` is the maximum time in milliseconds before update.

4.5.3 Example

In our example we create a subclass `TestThread` of class `Thread` and redefine the `run()` method to perform the set of actions that the thread is intended to do. The names of the threads are `0`, `1`, `2`, `3`.

```
// MyThread.java

class TestThread extends Thread
{
   int sleepTime;

   public TestThread(String s)
   {
   super(s);
   sleepTime = (int) (500*Math.random());
   System.out.println("Name: " + getName() + "\t Sleep: " + sleepTime);
   }  // end constructor TestThread

   public void run()
   {
   try {  sleep(sleepTime); }
   catch(Exception e) { }
   System.out.println("Thread " + getName());
   }  // end run
}  // end class TestThread

public class MyThread
{
   public static void main(String[] args)
   {
   TestThread thread0, thread1, thread2, thread3;
   thread0 = new TestThread("0"); thread1 = new TestThread("1");
   thread2 = new TestThread("2"); thread3 = new TestThread("3");
   thread0.start();
   thread1.start();
   thread2.start();
   thread3.start();

   }  // end main
}     // end class MyThread
```

A typical output is

```
first run:
Name: 0  Sleep: 223
Name: 1  Sleep:  55
Name: 2  Sleep: 401
Name: 3  Sleep: 482
Thread 0
Thread 2
Thread 3
Thread 1

second run
Name: 0  Sleep:  36
Name: 1  Sleep: 145
Name: 2  Sleep: 345
Name: 3  Sleep: 290
Thread 0
Thread 3
Thread 1
Thread 2
```

4.5.4 Priorities

The *priority* of a thread tells the scheduler how important this thread is. If there are a number of threads blocked and waiting to be run, the scheduler will run the one with the highest priority first. However, this does not mean that threads with lower priority do not get run. This means we cannot be deadlocked because of priorities. Lower priority threads just tend to run less often. In Java we can read the priority of a thread with the method `getPriority()` and change it with the method `setPriority()`. The following program shows an application of these methods.

```
// Bounce.java

import java.awt.*;
import java.awt.event.*;

public class Bounce extends Frame implements WindowListener,
             ActionListener
{
   Canvas canvas;

   public Bounce()
   {
   setTitle("BouncingBalls");
   addWindowListener(this);

   canvas = new Canvas();
   add("Center",canvas);

   Panel panel = new Panel();
   add("North",panel);

   Button b = new Button("Normal Ball");
   b.addActionListener(this);
   panel.add(b);

   b = new Button("High Priority Ball");
   b.addActionListener(this);
   panel.add(b);

   b = new Button("Close");
   b.addActionListener(this);
   panel.add(b);
   }  // end constructor Bounce()
```

```
   public void windowClosing(WindowEvent e) { System.exit(0); }
   public void windowClosed(WindowEvent e) { }
   public void windowOpened(WindowEvent e) { }
   public void windowDeiconified(WindowEvent e) { }
   public void windowIconified(WindowEvent e) { }
   public void windowActivated(WindowEvent e) { }
   public void windowDeactivated(WindowEvent e) { }

   public void actionPerformed(ActionEvent action)
   {
   if(action.getActionCommand() == "Normal Ball")
   {
   Ball ball = new Ball(canvas,Color.blue);
   ball.setPriority(Thread.NORM_PRIORITY);
   ball.start();
   }
   else if(action.getActionCommand() == "High Priority Ball")
   {
   Ball ball = new Ball(canvas,Color.red);
   ball.setPriority(Thread.NORM_PRIORITY+1);
   ball.start();
   }
   else if(action.getActionCommand() == "Close")
   System.exit(0);
   }

   public static void main(String[] args)
   {
   Frame frame = new Bounce();
   frame.setSize(400,300);
   frame.setVisible(true);
   }

}  // end class Bounce

class Ball extends Thread
{
   Canvas box;
   private static final int diameter = 10;
   private int x = 0;
   private int y = 0;
   private int dx = 2;
   private int dy = 2;
   Color color;
```

```
   Ball(Canvas canvas,Color col)
   {
   box = canvas;
   color = col;
   }

   public void draw()
   {
   Graphics g = box.getGraphics();
   g.setColor(color);
   g.fillOval(x,y,diameter,diameter);
   g.dispose();
   } // end draw

   public void move()
   {
   Graphics g = box.getGraphics();
   g.setColor(color);
   g.setXORMode(box.getBackground());
   g.fillOval(x,y,diameter,diameter);
   x += dx;
   y += dy;
   Dimension d = box.getSize();
   if(x < 0) { x = 0; dx = -dx; }
   if(x+diameter >= d.width) { x = d.width-diameter; dx = -dx; }
   if(y < 0) { y = 0; dy = -dy; }
   if(y+diameter >= d.height) { y = d.height - diameter; dy = -dy; }
   g.fillOval(x,y,diameter,diameter);
   g.dispose();
   } // end move

   public void run()
   {
   draw();
   for(;;)
   {
   move();
   try { sleep(10); }
   catch(InterruptedException e) { }
   } // end for loop
   } // end run

}  // end class Ball
```

4.5.5 Synchronization and Locks

Threads behave fairly independently with the Java virtual machine switching between the threads. This can cause problems since it is not known when a thread will be paused to allow another thread to run. Java has built-in support to prevent collisions over one kind of resource: the memory in an object. We typically make the data elements of a class `private` and access that memory only through methods, we can prevent collisions by making a particular method `synchronized`. Only one thread at the time can call a `synchronized` method for a particular object. Thus to prevent one thread from interfering with the other the `synchronized` modifier can be used when defining a method.

Any method can be preceded by the word

```
synchronized
```

The rule is: no two threads may be executing `synchronized` methods of the same object at the same time. The Java system enforces this rule by associating a monitor lock with each object. When a thread calls a `synchronized` method of an object, it tries to grab the object's monitor lock. If another thread is holding the lock, it waits until that thread releases it. A thread releases the monitor lock when it leaves the `synchronized` method. Of one `synchronized` method of a call contains a call to another, a thread may have the same lock multiple times. Java keeps track of that correctly.

Thus there is a lock with every object. The `synchronized` statement computes a reference to an object. It then attempts to perform a lock operation on that object and does not proceed further until the lock operation has successfully completed. A lock operation may be delayed because the rules about locks can prevent the main memory from participating until some other thread is ready to perform one or more unlock operations. After the lock operation has been performed, the body of the `synchronized` statement is executed. Normally, a compiler ensures that the lock operation implemented by a monitorenter instruction executed prior to the execution of the body of the `synchronized` statement is matched by an unlock operation implemented by a monitorexit instruction whenever the `synchronized` statement completes, whether completion is normal or abrupt. The Java Virtual Machine provides separate monitorenter and monitorexit instructions that implements the lock and unlock operations.

A `synchronized` method automatically performs a lock operation when it is invoked. Its body is not executed until the lock operation has successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked. This means, the object that will be known as `this` during execution of the method's body. If the method is `static`, it locks the lock associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an

unlock operation is automatically performed on that same lock.

This mutual exclusion can be accomplished in Java using the `synchronized` keyword. This can be applied to methods, such as

```
public synchronized void withdraw(double ammount) {
...
}
```

A synchronized method/block can be viewed as a "critical section" that excludes access the object by any other synchronized methods during its execution.

In the following program we start two threads `f1` and `f2`. The method `display()` is synchronized. The output the program `Synchronized.java` is

```
100 101 102
100 101 102
100 101 102
100 101 102
100 101 102
100 101 102
```

If we change the line

```
synchronized void display()
```

in the program to

```
void display()
```

the output is

```
100 100 101 101 102 102
100 100 101 101 102 102
100 100 101 101 102 102
```

```
// Synchronized.java

public class Synchronized
{
    Count f1, f2;

    Synchronized()
    {
    f1 = new Count(this);
    f2 = new Count(this);
    f1.start();
    f2.start();
    } // end constructor Synchronized

    synchronized void display()
    {
    System.out.print("100 ");
    System.out.print("101 ");
    System.out.print("102 ");
    System.out.println();
    }

    public static void main(String[] args)
    {
    new Synchronized();
    }
}  // end class Synchronized

class Count extends Thread
{
   Synchronized current;

   Count(Synchronized thread)
   {
   current = thread;
   }

   public void run()
   {
   int i;
   for(i=0; i < 3; i++)
   current.display();
   }
}
```

In the following programs `BankAccount.java` and `Transfer.java` we consider the problem of transferring money between bank accounts. It is left as an exercise to the reader to investigate whether or not the method `getBalance()` should be synchronized or not in the following two programs.

```
// BankAccount.java

public class BankAccount
{
   private double balance;

   public BankAccount(double balance)
   {
   this.balance = balance;
   }

   public double getBalance()
   {
   return balance;
   }

   public synchronized void deposit(double amount)
   {
   balance += amount;
   }

   public synchronized void withdraw(double amount)
   throws RuntimeException
   {
   if(amount > balance)
   {
   throw new RuntimeException("Overdraft");
   }
   balance -= amount;
   }

   public synchronized void transfer(double amount,BankAccount destination)
   {
   this.withdraw(amount);
   Thread.yield(); // allows the scheduler to select another runnable thread
   destination.deposit(amount);
   }
}  // end class BankAccount
```

```
// Transfer.java

import BankAccount;

public class Transfer implements Runnable
{
   public static void main(String[] args)
   {
   Transfer x =
   new Transfer(new BankAccount(100.0),new BankAccount(100.0),50.0);
   Thread t = new Thread(x);

   t.start();
   Thread.yield(); // the thread on which yield() is invoked would
                   // move from running state to ready state

   System.out.println("Account A has Dollar: " + x.A.getBalance());
   System.out.println("Account B has Dollar: " + x.B.getBalance());
   }

   public BankAccount A, B;
   public double amount;

   public Transfer(BankAccount A,BankAccount B,double amount)
   {
   this.A = A;
   this.B = B;
   this.amount = amount;
   }

   public void run()
   {
   System.out.println("Before transfer A has Dollar: "
                      + A.getBalance());
   System.out.println("Before transfer B has Dollar: "
                      + B.getBalance());
   A.transfer(amount,B);
   System.out.println("After transfer A has Dollar: "
                      + A.getBalance());
   System.out.println("After transfer B has Dollar: "
                      + B.getBalance());
   }
}
```

4.5.6 Producer Consumer Problem

The producer consumer problem is a classic *synchronization problem*. The producer and consumer processes share a common buffer. The producer executes a loop in which it puts new items into the buffer and the consumer executes a loop in which it removes items from the buffer. The following important conditions have to be satisfied by the producer and consumer. At most one process (producer or consumer) may be accessing the shared buffer at any time. This condition is called mutual exclusion. When the buffer is full, the producer should be put to sleep. It should only wake up when an empty slot becomes available. This is called synchronization. When the buffer is empty, the consumer should be put to sleep. It should only wake up when at least one slot becomes full. This is also called synchronization.

The following four programs

`Producer.java, Consumer.java, Buffer.java, Main.java`

provide a solution to this problem. The methods `wait()` and `notify()` are in the class `Object`. Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class. The method `wait()` causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. The method `notify()` wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakend. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

```
// Producer.java

public class Producer extends Thread
{
   private Buffer buffer;

   public Producer(Buffer b)
   {
   buffer = b;
   }

   public void run()
   {
   for(int i=0; i < 10; i++)
   {
   buffer.put(i);
   System.out.println("Producer put: " + i);
```

```
   try
   {
   sleep((int) (Math.random()*100));
   }
   catch(InterruptedException e) { }
   }
   }
}


// Consumer.java

public class Consumer extends Thread
{
   private Buffer buffer;

   public Consumer(Buffer b)
   {
   buffer = b;
   }

   public void run()
   {
   int value = 0;
   for(int i=0; i < 10; i++)
   {
   value = buffer.get();
   System.out.println("Consumer got: " + value);
   }
   }
}


// Buffer.java

public class Buffer
{
   private int contents;
   private boolean available = false;

   public synchronized int get()
   {
   while(available == false)
   {
   try
```

```
   {
   wait();
   }
   catch(InterruptedException e) {  }
   }
   available = false;
   notify();
   return contents;
   }

   public synchronized void put(int value)
   {
   while(available == true)
   {
   try
   {
   wait(); // block until woken up by a call to notify
   }
   catch(InterruptedException e) {  }
   }
   contents = value;
   available = true;
   notify();
   }
}


// Main.java

public class Main
{
   public static void main(String[] args)
   {
   Buffer b = new Buffer();
   Producer p = new Producer(b);
   Consumer c = new Consumer(b);
   p.start();
   c.start();
   }
}
```

4.6 Locking Files for Shared Access

Using the `FileLock` class and `FileChannel` class we can use a file lock to restrict access to a file from multiple processes. We have the option of restricting access to an entire file or just a region of it. A file-lock is either shared or exclusive. A file lock object is created each time a lock is acquired on a file via one of the `lock()` or `tryLock()` methods of the `FileChannel` class. A file-lock is initially valid. It remains valid until the lock is released by invoking the `release()` method. The `release()` method is in the `FileLock` class. The following two programs show an example.

We compile the program `LockFile.java` which accesses the file `data.dat` for read and write. Then we run the program. The program gets an exclusive lock on the file `data.dat`, reports when it has the lock, and then waits until we press the Enter key. Before we press the Enter key we start a new process by compiling and running the program `NoOfLines.java`. This program counts the numbers of lines in the file `data.dat`. Since the file `data.dat` is locked it cannot access the file only after we press the Enter key in the first process. Then the lock is released.

```
// LockFile.java

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class LockFile
{
   public static void main(String[] args) throws IOException
   {
   File f = new File("c:\\javacourse/data.dat");
   RandomAccessFile raf = new RandomAccessFile(f,"rw");
   FileChannel channel = raf.getChannel();
   FileLock lock = channel.lock();
   try
   {
   System.out.println("locked");
   System.out.println("Press ENTER to continue");
   System.in.read(new byte[10]);
   }
   finally
   {
   lock.release();
   }
   }
}
```

```
// NoOfLines.java

import java.io.*;
import java.util.*;

public class NoOfLines
{
   public static void main(String[] args) throws IOException
   {
   FileInputStream fin = new FileInputStream("c:\\javacourse/data.dat");
   BufferedReader in;
   in = new BufferedReader(new InputStreamReader(fin));

   int count = 0;

   while(in.readLine() != null)
   {
   count++;
   }
   System.out.println("count = " + count);

   } // end main
}
```

Chapter 5

JDBC

5.1 Introduction

Standard relational data access is very important for Java programs because the Java applets by nature are not monolithic, all-consuming applications. As applets by nature are modular, they need to read persistent data from data stores, process the data, and write the data back to data stores for other applets to process. Monolithic programs could afford to have their own proprietary schemes of data handling. But as Java applets cross operating system and network boundaries, we need published open data access schemes.

The Java Database Connectivity (JDBC) of the Java Enterprise API's JavaSoft is the first of such cross-platform, cross-database approaches to database access from Java programs. From a developer's point of view, JDBC is the first standardized effort to integrate relational databases with Java programs. JDBC has opened all the relational power that can be mustered to Java applets and applications. We see how JDBC can be used to develop database programs using Java.

JDBC supports transaction processing with the `commit()` and `rollback()` methods. JDBC also has the `autocommit()` method which, when on, all changes are committed automatically and, if off, the Java program has to use the `commit()` or `rollback()` methods to effect the changes to the data.

JDBC is Java Database Connectivity - a set of relational database objects and methods for interacting with data sources. Even though the objects and methods are based on the relational database model, JDBC makes no assumption about the underlying data source or the data storage scheme. We can access and retrieve audio or video data from many sources and load into Java objects using the JDBC APIs! The only requirement is that there should be a JDBC implementation for that source.

The JDBC designers based the API on X/Open SQL Call Level Interface (CLI). It is not coincidental that Open Database Connection (ODBC) an API defined by Microsoft is also based on the X/Open CLI. The JavaSoft engineers wanted to gain leverage from the existing ODBC implementation and development expertise, thus making it easier for Independent Software Vendors (ISVs) and system developers to adopt JDBC. But ODBC is a C interface to DBMSs and thus is not readily convertible to Java. So JDBC design followed ODBC in spirit as well in its major abstractions and implemented the SQL CLI with a Java interface that is consistent with the rest of the Java system. For example, instead of the ODBC SQLBindColumn and SQLFetch to get column values from the result, JDBC used a simpler approach.

JDBC is designed upon the CLI model. JDBC defines a set of API objects and methods to interact with the underlying database. A Java program first opens a connection to a database, makes a statement object, passes SQL statements to the underlying DBMS through the statement object, and retrieves the results as well as information about the result sets. Typically, the JDBC class files and the Java applet/application reside in the client. They could be downloaded from the network also. To minimize the latency during execution, it is better to have the JDBC classes in the client. The Database Management System and the data source are typically located in a remote server. The applet/application and the JDBC layers communicate in the client system, and the driver takes care of interacting with the database over the network.

The JDBC driver can be a native library, like the JDBC-ODBC Bridge, or a Java class talking across the network to an RPC or Jeeves Servlet or HTTP listener process in the database server.

The JDBC classes are in the `java.sql` package, and all Java programs use the objects and methods in the `java.sql` package to read from and write to data sources. A program using the JDBC will need a driver for the data source with which it wants to interface. This driver can be a native module (like the `JDBCODBC.DLL` for the Windows JDBC-ODBC Bridge developed by Sun/Intersolv), or it can be a Java program that talks to a server in the network using some RPC or Jeeves Servlet or an HTTP talker-listener protocol.

JDBC can be implemented as a native driver or as a gateway to an RPC. Which implementation is better is a question that will be answered as the JDBC architecture matures. One reason to implement a native library is the advantage of speed. Also, local databases could be handled using native libraries more easily than gateways. On the other hand, for a handheld device or a network computer, network is the system. For these devices, a full Java implementation of JDBC that talks to an RPC type of system or a Jeeves servlet on the database server is a good solution.

It is conceivable that an application will deal with more than one data source-possibly heterogeneous data sources. A database gateway program is a good example of an application that accesses multiple heterogeneous data sources. For this reason, JDBC has a `DriverManager` whose function is to manage the drivers and provide a list of currently loaded drivers to the application programs.

Even though the word Database is in the name JDBC, the form, content, and location of the data is immaterial to the Java program using JDBC, so long as there is a driver for that data. Hence, the notation data source to describe the data is more accurate than Database, DBMS, DB, or just file. ODBC also refers to data sources, rather than databases when being described in general terms.

Security is an important issue, especially when databases are involved. JDBC follows the standard security model in which applets can connect only to the server from where they are loaded; remote applets cannot connect to local databases. Applications have no connection restrictions. For pure Java drivers, the security check is automatic, but for drivers developed in native methods, the drivers must have some security checks.

As a part of JDBC, JavaSoft also will deliver a driver to access ODBC data sources from JDBC. This driver is jointly developed with Intersolv and is called the JDBC-ODBC bridge. The JDBC-ODBC bridge is implemented as the `JdbcOdbc.class` and a native library to access the ODBC driver. For the Windows platform, the native library is a DLL (`JDBCODBC.DLL`). As JDBC is close to ODBC in design, the ODBC bridge is a thin layer over JDBC. Internally, this driver maps JDBC methods to ODBC calls, and thus interacts with any available ODBC driver. The advantage of this bridge is that now JDBC has the capability to access almost all databases, as ODBC drivers are widely available.

JDBC makes it possible to do three things:

1. establish a connection with a database

2. send SQL statements

3. process the results

5.2 Classes for JDBC

5.2.1 Introduction

The first thing we need to do is establish a connection with the DBMS we want to use. This involves two steps:

(1) loading the driver

(2) making the connection.

Loading the driver or drivers we want to use is simple and involves one line of code. If, for example, we want to use the JDBC-ODBC Bridge driver, the following code will load it

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

The driver documentation will give us the class name to use. For instance, if the class name is `jdbc.DriverXYZ`, we would load the driver with the following line of code

```
Class.forName("jdbc.DriverXYZ");
```

We do not need to create an instance of a driver and register it with the class `DriverManager` because calling `Class.forName` will do that for us automatically. If we were to create our own instance, we would be creating an unecessary duplicate.

When we look at the class hierarchy and methods associated with it, the topmost class in the hierarchy is the `DriverManager`. This class provides the basic service for managing a set of JDBC drivers. Thus the `DriverManager` keeps the driver information, state information, and so on. When each driver is loaded, it registers with the `DriverManager`. The `DriverManager`, when required to open a connection, selects the driver depending on the JDBC URL. JDBC identifies a database with an URL. Databases are specified with URL syntax. The URL is of the form

```
jdbc:<subprotocol>:<subname related to the DBMS/Protocol>
```

For databases on the Internet/intranet, the subname can contain the Net URL

```
//hostname:port/...
```

The `<subprotocol>` can be any name that a database understands. The `odbc` subprotocol name is reserved for ODBC-style data sources. A normal ODBC database JDBC URL looks like:

```
jdbc:odbc:<>;User=<>;PW=<>
```

5.2.2 Classes DriverManager and Connection

The `java.sql.Driver` class is usually referred to for information such as PropertyInfo, version number, and so on. So the class could be loaded many times during the execution of a Java program using the JDBC API. Looking at the `java.sql.Driver` and `java.sql.DriverManager` classes and methods we see that the DriverManager returns a `Connection` object when we use one of the three `getConnection()` method. The methods

```
static Connection getConnection(String url)

static Connection getConnection(String url,Properties info)

static Connection getConnection(String url,String user,String password)
```

attempt to establish a connection to a given database.

The following line of code illustrates the use

```
Connection con = DriverManager.getConnection(url,"myLogin","myPassword");
```

Other useful methods include the

```
static void registerDriver(Driver driver)
```

which registers the given driver with the `DriverManager`

```
static void deregisterDriver(Driver driver)
```

drops a driver from the DriverManager's list

```
static Driver getDrivers(String url)
```

atempts to locate a driver that understands the given URL.

If we are using the JDBC-ODBC bridge driver, the JDBC URL will start with `jdbc:odbc:`. The rest of the URL is generally our data source name or database system. The connection returned by the method

```
DriverManager.getConnection()
```

is an open connection we can use to create JDBC statements that pass our SQL statements to the DBMS.

First we need to configure an ODBC data source. The `getConnection()` method requires

1) data source name (DSN),

2) user ID,

3) password

for the ODBC data source. The database driver type or subprotocol name is `odbc`. So the driver manager finds out from the ODBC driver the rest of the details. Where do we put the rest of the details? This is where the ODBC setup comes into the picture. The ODBC Setup program runs outside the Java application from the Microsoft ODBC program group. With the ODBC Setup program we set up the data source so that this information is available to the ODBC Driver Manager, which, in turn, loads the Microsoft Access ODBC driver. If the database is in another DBMS form-say, Oracle we configure this source as Oracle ODBC driver. With Windows 98 and Windows NT 4.0, this information is in the Registry.

The `Connection` class is one of the major classes in JDBC. It has a lot of functionality, ranging from transaction processing to creating statements, in one class. The connection is for a specific database that can be interacted with in a specific subprotocol. The `Connection` object internally manages all aspects about a connection, and the details are transparent to the program. The `Connection` object is a pipeline into the underlying DBMS driver. The information to be managed includes the data source identifier, the subprotocol, the state information, the DBMS SQL execution plan ID or handle, and any other contextual information needed to interact successfully with the underlying DBMS. The data source identifier could be a port in the Internet database server that is identified by the server

```
name:port/...
```

URL, just a data source name used by the ODBC driver, or a full path name to a database file in the local computer.

Another important function performed by the `Connection` object is transaction management. The handling of transactions depends on the state of an internal autocommit flag that is set using the

```
void setAutoCommit(boolean)
```

method, and the state of this flag can be read using the

```
boolean getAutoCommit()
```

method. When the flag is true, the transactions are automatically committed as soon as they are completed. There is no need for any intervention or commands from the Java application program. When the flag is false, the system is in the manual mode. The Java program has the option to commit the set of transactions that happened after the last commit or to rollback the transactions using the `commit()` and `rollback()` methods. Both methods are in the class `Connection`.

JDBC also provides methods for setting the transaction isolation modularity. When we are developing multi-tiered applications, there will be multiple users performing concurrently interleaved transactions that are on the same database tables. A database driver has to employ sophisticated locking and data-buffering algorithms and mechanisms to implement the transaction isolation required for a large-scale JDBC application. This is more complex when there are multiple Java objects working on many databases that could be scattered across the globe!

Once we have a successful `Connection` object to a data source, we can interact with the data source in many ways. The most common approach from an application developer standpoint is the objects that handle the SQL statements.

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed. To be more precise, the default is for SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all case, however, a statement is completed, and therefore committed, right after it is executed.

Calling the method `rollback()` aborts a transaction and returns any values that were modified to their previous values. If we are trying to execute one or more statements in a transaction and get an `SQLException`, we should call the method `rollback()` to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed. Catching an `SQLException` provides the information that something is wrong, but we do not what was or was not committed. Since we cannot count on the fact that nothing was committed, calling the method `rollback()` is the only way to be sure.

5.2.3 Class Statement

The `Statement` object does all of the work to interact with the Database Management System in terms of SQL statements. We can create many `Statement` objects from one `Connection` object. Internally, the `Statement` object would be storing the various data needed to interact with a database, including state information, buffer handles, and so on. But these are transparent to the JDBC application program.

To handle data from a database, a Java program implements the following general steps. First, the program calls the `getConnection()` method to get the `Connection` object. Then, it creates the `Statement` object and prepares a SQL statement. A SQL statement can be executed immediately (`Statement` object), or can be a compiled statement (`PreparedStatement` object) or a call to a stored procedure (`CallableStatement` object). When the method `executeQuery()` is executed, a `ResultSet` object is returned. SQL statements such as update or delete will not return a `ResultSet`. For such statements, the `executeUpdate()` method is used. The `executeUpdate()` method returns an integer which denotes the number of rows affected by the SQL statement. The `ResultSet` contains rows of data that is parsed using the `next()` method. In case of a transaction processing application, methods such as `rollback()` and `commit()` can be used either to undo the changes made by the SQL statements or permanently affect the changes made by the SQL statements.

When a program attempts an operation that is not in sync with the internal state of the system (for example, a `next()` method to get a row when no SQL statements have been executed), this discrepancy is caught and an exception is raised. This exception, normally, is probed by the application program using the methods in the SQLException object.

JDBC supports three types of statements:

```
Statement
PreparedStatement
CallableStatement
```

The `Connection` object has the

```
public Statement createStatement() throws SQLException
```

```
public PreparedStatement prepareStatement(String sql) throws SQLException
```

and

```
public CallableStatement prepareCall(String sql) throws SQLException
```

methods to create these `Statement` objects.

A Java application program first builds the SQL statement in a string buffer and passes this buffer to the underlying DBMS through some API call. A SQL statement needs to be verified syntactically, optimized, and converted to an executable form before execution. In the Call Level Interface (CLI) Application Program Interface (API) model, the application program passes the SQL statement to the driver which, in turn, passes it to the underlying DBMS. The DBMS prepares and executes the SQL statement. After the DBMS receives the SQL string buffer, it parses the statement and does a syntax check run. If the statement is not syntactically correct, the system returns an error condition to the driver, which generates a `SQLException`. If the statement is syntactically correct, depending on the DBMS, then many query plans usually are generated that are run through an optimizer (often a cost-based optimizer). Then, the optimum plan is translated into a binary execution plan. After the execution plan is prepared, the DBMS usually returns a handle or identifier to this optimized binary version of the SQL statement back to the application program.

The three JDBC statements

`Statement, PreparedStatement, CallableStatement`

differ in the timing of the SQL statement preparation and the statement execution. In the case of the simple `Statement` object, the SQL is prepared and executed in one step (at least from the application program point of view. Internally, the driver might get the identifier, command the DBMS to execute the query, and then discard the handle). In the case of a `PreparedStatement` object, the driver stores the execution plan handle for later use. In the case of the `CallableStatement` object, the SQL statement is actually making a call to a stored procedure that is usually already optimized.

Stored procedures are encapsulated business rules or procedures that reside in the database server. They also enforce uniformity across applications, as well as provide security to the database access. Stored procedures last beyond the execution of the program. So the application program does not spend any time waiting for the DBMS to create the execution plan.

The JDBC processing is synchronous; that is, the application program must wait for the SQL statements to complete. However Java is a multithreaded platform. Thus the JDBC designers suggest using threads to simulate asynchronous processing.

The `Statement` object is best suited for ad hoc SQL statements or SQL statements that are executed once. The DBMS goes through the syntax run, query plan optimization, and the execution plan generation stages as soon as this SQL statement is received. The DBMS executes the query and then discards the optimized execution plan. So, if the `executeQuery()` method is called again, the DBMS goes through all of the steps again.

We give now all methods in the class `Statement`. A `Statement` object is created using the `createStatement()` method in the Connection object.

Table: `Statement` Methods

```
ResultSet executeQuery(String sql)
int executeUpdate(String sql)
Boolean execute(String sql)
Boolean getMoreResults()
void close()
int getMaxFieldSize()
void setMaxFieldSize(int max)
int getMaxRows()
void setMaxRows(int max)
void setEscapeProcessing(boolean enable)
int getQueryTimeout()
void setQueryTimeout(int seconds)
void cancel()
java.sql.SQLWarning getWarnings()
void clearWarnings()
void setCursorName(String name)
ResultSet getResultSet()
int getUpdateCount
```

The most important methods are

```
ResultSet executeQuery(String)
int executeUpdate(String)
boolean execute(String)
```

As we create a `Statement` object with a SQL statement, the `executeQuery()` method takes a SQL string. It passes the SQL string to the underlying data source through the driver manager and gets the `ResultSet` back to the application program.

Important! The `executeQuery()` method returns only one `ResultSet`.

For those cases that return more than one `ResultSet`, the `execute()` method should be used. Only one `ResultSet` can be opened per `Statement` object at one time. For SQL statements that do not return a `ResultSet` such as the `UPDATE`, `DELETE`, and DDL statements, the `Statement` object has the `executeUpdate()` method that takes a SQL string and returns an integer. This integer indicates the number of rows that are affected by the SQL statement.

5.2.4 Class PreparedStatement

In the case of a `PreparedStatement` object, as the name implies, the application program prepares a SQL statement using the

```
java.sql.Connection.prepareStatement()
```

method. The `prepareStatement()` method takes a SQL string, which is passed to the underlying DBMS. The DBMS goes through the syntax run, query plan optimization, and the execution plan generation stages but does not execute the SQL statement. Possibly, the DBMS returns a handle to the optimized execution plan that the JDBC driver stores internally in the `PreparedStatement` object. The methods of the PreparedStatement object are shown in the following Table. Notice that the `executeQuery()`, `executeUpdate()`, and `execute()` methods do not take any parameters. They are just calls to the underlying DBMS to perform the already optimized SQL statement.

Table: `PreparedStatement` Methods

`ResultSet`	`executeQuery()`
`int`	`executeUpdate()`
`boolean`	`execute()`

One of the major features of a `PreparedStatement` is that it can handle IN types of parameters. The parameters are indicated in a SQL statement by placing the `?` as the parameter marker instead of the actual values. In the Java program, the association is made to the parameters with the

```
void setXXXX()
```

methods, as shown in the following Table. All of the `setXXXX()` methods take the parameter index, which is

1 for the first `?`, 2 for the second `?`

and so on.

Table: `java.sql.PreparedStatement` Methods

```
void clearParameters()
void setAsciiStream(int parameterIndex,InputStream x,int length)
void setBinaryStream(int parameterIndex,InputStream x,int length)
void setBoolean(int parameterIndex, boolean x)
void setByte(int parameterIndex,byte x)
void setBytes(int parameterIndex,byte x[])
void setDate(int parameterIndex,java.sql.Date x)
void setDouble(int parameterIndex,double x)
void setFloat(int parameterIndex,float x)
void setInt(int parameterIndex,int x)
void setLong(int parameterIndex,long x)
void setNull(int parameterIndex,int sqlType)
void setNumeric(int parameterIndex,Numeric x)
void setShort(int parameterIndex,short x)
void setString(int parameterIndex,String x)
void setTime(int parameterIndex,java.sql.Time x)
void setTimestamp(int parameterIndex,java.sql.Timestamp x)
void setUnicodeStream(int parameterIndex,InputStream x,int length)
```

Advanced Features-Object Manipulation

```
void setObject(int parameterIndex,Object x,int targetSqlType,int scale)
void setObject(int parameterIndex,Object x,int targetSqlType)
void setObject(int parameterIndex,Object x)
```

In the case of the `PreparedStatement`, the driver actually sends only the execution plan ID and the parameters to the DBMS. This results in less network traffic and is well-suited for Java applications on the Internet. The `PreparedStatement` should be used when we need to execute the SQL statement many times in a Java application. But remember, even though the optimized execution plan is available during the execution of a Java program, the DBMS discards the execution plan at the end of the program. So, the DBMS must go through all of the steps of creating an execution plan every time the program runs. The `PreparedStatement` object achieves faster SQL execution performance than the simple `Statement` object, as the DBMS does not have to run through the steps of creating the execution plan.

5.2.5 Class CallableStatement

For a secure, consistent, and manageable multi-tier client/server system, the data access should allow the use of stored procedures. Stored procedures centralize the business logic in terms of manageability and also in terms of running the query. Java applets running on clients with limited resources cannot be expected to run huge queries. But the results are important to those clients. JDBC allows the use of stored procedures by the

`CallableStatement`

class and with the escape clause string. A `CallableStatement` object is created by the `prepareCall()` method in the `Connection` object. The `prepareCall()` method takes a string as the parameter. This string, called an escape clause, is of the form

```
{[? =] call stored procedure name [parameter,parameter ...]}
```

The `CallableStatement` class supports parameters. These parameters are of the OUT kind from a stored procedure or the IN kind to pass values into a stored procedure. The parameter marker (question mark) must be used for the return value (if any) and any output arguments, because the parameter marker is bound to a program variable in the stored procedure. Input arguments can be either literals or parameters. For a dynamic parameterized statement, the escape clause string takes the form:

```
{[? =] call stored procedure name [?,?; ...]}
```

The OUT parameters should be registered using the `registerOutparameter()` method-as shown in the Table before the call to the `executeQuery()`, `executeUpdate()`, or `execute()` methods.

Table: `CallableStatement` Methods

```
void registerOutParameter(int parameterIndex,int sqlType)
void registerOutParameter(int parameterIndex,int sqlType,int scale)
```

After the stored procedure is executed, the DBMS returns the result value to the JDBC driver. This return value is accessed by the Java program using the methods in following Table.

Table: `CallableStatement` Methods

```
Boolean getBoolean(int parameterIndex)
byte getByte(int parameterIndex)
byte[] getBytes(int parameterIndex)
java.sql.Date getDate(int parameterIndex)
double getDouble(int parameterIndex)
float getFloat(int parameterIndex)
int getInt(int parameterIndex)
long getLong(int parameterIndex)
Numeric getNumeric(int parameterIndex,int scale)
Object getObject(int parameterIndex)
short getShort(int parameterIndex)
String getString(int parameterIndex)
java.sql.Time getTime(int parameterIndex)
java.sql.Timestamp getTimestamp(int parameterIndex)
boolean wasNull()
```

The method `boolean wasNull()` indicates whether or not the last `OUT` parameter read had the value of SQL `NULL`.

JDBC has minimized the complexities of getting results from a stored procedure. It still is a little involved, but is simpler. Maybe in the future, these steps will become simpler.

5.2.6 Class ResultSet

The `ResultSet` object is actually a tabular data set; that is, it consists of rows of data organized in uniform columns. In JDBC, the Java program can see only one row of data at one time. The program uses the `next()` method to go to the next row. JDBC does not provide any methods to move backwards along the `ResultSet` or to remember the row positions (called bookmarks in ODBC). Once the program has a row, it can use the positional index (1 for the first column, 2 for the second column, and so on) or the column name to get the field value by using the `getXXXX()` methods. The Table shows the methods associated with the `ResultSet` object.

Table: `java.sql.ResultSet` Methods

```
boolean next()
void close()
boolean wasNull()
```

The `ResultSet` methods are very simple. The major ones are the `getXXX()` methods. The `getMetaData()` method returns the meta data information about a `ResultSet`. The interface `DatabaseMetaData` also returns the results in the `ResultSet` form. The `ResultSet` also has methods for the silent SQLWarnings. It is a good practice to check any warnings using the `getWarning()` method that returns a `null` if there are no warnings.

Although the method `getString` is recommended for retrieving the SQL types `CHAR` and `VARCHAR`, it is also possible to retrieve any of the basic SQL types with it. For instance, if it is used to retrieve a numeric type, the method `getString` will convert the numeric value to a Java `String` object, and the value will have to be converted back to a numeric type before it can be operated on as a number. In case where the value will be treated as a string anyway, there is no drawback.

It is now possible to produce `ResultSet` objects that are scrollable and/or updatable. For example

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM MyTable");
```

The methods in class `ResultSet` to get data by column position are

```
java.io.InputStream
getAsciiStream(int columnIndex)
java.io.InputStream
getBinaryStream(int columnIndex)
boolean getBoolean(int columnIndex)
byte getByte(int columnIndex)
byte[] getBytes(int columnIndex)
java.sql.Date getDate(int columnIndex)
double getDouble(int columnIndex)
float getFloat(int columnIndex)
int getInt(int columnIndex)
long getLong(int columnIndex)
java.sql.Numeric
getNumeric(int columnIndex, int scale)
Object getObject(int columnIndex)
short getShort(int columnIndex)
String getString(int columnIndex)
java.sql.Time getTime(int columnIndex)
java.sql.Timestamp getTimestamp(int columnIndex)
java.io.InputStream getUnicodeStream(int columnIndex)
```

The methods in class `ResultSet` to get data by column name are

```
java.io.InputStream getAsciiStream(String columnName)
java.io.InputStream getBinaryStream(String columnName)
boolean getBoolean(String columnName)
byte getByte(String columnName)
byte[] getBytes(String columnName)
java.sql.Date getDate(String columnName)
double getDouble(String columnName)
float getFloat(String columnName)
int getInt(String columnName)
long getLong(String columnName)
java.sql.Numeric getNumeric(String columnName,int scale)
Object getObject(String columnName)
short getShort(String columnName)
String getString(String columnName)
java.sql.Time getTime(String columnName)
java.sql.Timestamp getTimestamp(String columnName)
java.io.InputStream getUnicodeStream(String columnName)
int findColumn(String columnName)
SQLWarning getWarnings()
void clearWarnings()
String getCursorName()
ResultSetMetaData getMetaData()
```

5.2.7 Class SQLException

The `SQLException` class in JDBC provides a variety of information regarding errors that occurred during a database access. The `SQLException` objects are chained so that a program can read them in order. This is a good mechanism, as an error condition can generate multiple errors and the final error might not have anything to do with the actual error condition. By chaining the errors, we can actually pinpoint the first error. Each `SQLException` has an error message and vendor-specific error code. Also associated with a `SQLException` is a SQLState string that follows the XOPEN SQLstate values defined in the SQL specification. The following Table lists the methods for the `SQLException` class.

Table: `SQLException` Methods

```
SQLException SQLException(String reason,String SQLState,int vendorCode)
SQLException SQLException(String reason,String SQLState)
SQLException SQLException(String reason)
SQLException SQLException()
String getSQLState()
int getErrorCode()
SQLException getNextException()
void setNextException(SQLException ex)
```

Handling Exceptions in JDBC-`SQLWarning` Class

Unlike the SQLExceptions that the program knows have happened because of raised exceptions, the SQLWarnings do not cause any commotion in a Java program. The SQLWarnings are tagged to the object whose method caused the warning. So we should check for warnings using the `getWarnings()` method that is available for all objects. The Table lists the methods associated with the SQLWarnings class.

Table: `SQLWarning` Methods

```
SQLWarning SQLWarning(String reason,String SQLstate,int vendorCode)
SQLWarning SQLWarning(String reason,String SQLstate)
SQLWarning SQLWarning(String reason)
SQLWarning SQLWarning()
SQLWarning getNextWarning()
void setNextWarning(SQLWarning w)
```

5.2.8 Classes Date, Time and TimeStamp

We now look at some of the supporting classes that are available in JDBC. These classes are `Date`, `Time` and `TimeStamp`. Most of these classes extend the basic Java classes to add the capability to handle and translate data types that are specific to SQL. The package

`java.sql.Date`

gives a Java program the capability to handle SQL `Date` information with only year, month, and day values. This package contrasts with the `java.util.Date`, where the time in hours, minutes, and seconds is also kept.

Table: `java.sql.Date` Methods

```
Date Date(int year,int month,int day)
Date valueOf(String s)
String toString()
```

The package

`java.sql.Time`

adds the `Time` object to the `java.util.Date` package to handle only hours, minutes, and seconds. `java.sql.Time` is also used to represent SQL time information.

Table: `java.sql.Time` Methods

```
Time Time(int hour,int minute,int second)
Time Time valueOf(String s)
String toString()
```

The

`java.sql.Timestamp`

package adds the `Timestamp` class to the `java.util.Date` package. It adds the capability of handling nanoseconds. But the granularity of the subsecond timestamp depends on the database field as well as the operating system.

Table: `java.sql.Timestamp` Methods

```
Timestamp Timestamp(int year,int month,int day,int hour,
                    int minute,int second,int nano)
Timestamp valueOf(String s)
String toString()
int getNanos()
void setNanos(int n)
boolean equals(Timestamp ts)
```

5.3 Data Types in SQL

In JDBC, the SQL types are defined in the

`java.sql.Types`

class and the different numeric types are handled in the

`java.sql.Numeric`

class. The class `java.sql.Types` defines a set of XOPEN equivalent integer constants that identify SQL types. The constants are `final` types. Therefore, they cannot be redefined in applications or applets. The Table lists the constant names and their values.

Table: `java.sql.Types` Constants

Constant Name	Value
BIGINT	-5
BINARY	-2
BIT	-7
CHAR	1
DATE	91
DECIMAL	3
DOUBLE	8
FLOAT	6
INTEGER	4
LONGVARBINARY	-4
LONGVARCHAR	-1
NULL	0
NUMERIC	2
OTHER	1111
REAL	7
SMALLINT	5
TIME	92
TIMESTAMP	93
TINYINT	-6
VARBINARY	-3
VARCHAR	12

JDBC Types Mapped to Java Types

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Standard Mapping from Java Types to JDBC Types

Java Type	JDBC Type
String	CHAR, VARCHAR or LONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	BINARY, VARBINARY, LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

5.4 Example

The database is the Microsoft Access database. We access the database using JDBC and the JDBC-ODBC bridge. Here we describe how to link JDBC with Microsoft ACCESS.

1) First we create in Microsoft ACCESS an empty (blank) database with filename:

`Phone.mdb`

in a given directory, for example

`C:\Swiss`

2) Next we have to set up the database driver and the DSN (data source name). We select `PhoneBook` for the DSN. The DSN is used in our Java code to specify (or points to) the database that are going to be used.

a) Click on MyComputer
b) Click on Control Panel
c) Click on Data Source (32 bit)(ODBC)
d) Click on Add
e) Select the Microsoft Access Driver
f) Click the Finish button
g) Fill in the Data Source Name: `PhoneBook`
h) Click on Select
i) Fill in name of `.mdb` file, in our case
`Phone.mdb`
in the directory `Swiss`.
j) Click O.K.

The string `PhoneBook` is stored in the file `Odbc.ini`

```
......
PhoneBook=Microsoft Access Driver (*.mdb) (32 bit)
......
[PhoneBook]
Driver32=C:\WINNT\System32\odbcjt32.dll
```

5.5 Programs

We creates a table called `MyPhoneBook` with four columns for `firstname`, `surname`, `phonehome`, `phonebus`.

```
// CreateTable.java

import java.sql.*;

public class CreateTable
{
   public static void main(String[] args)
   {
   String url = "jdbc:odbc:PhoneBook"; // PhoneBook is the DSN
                                       // it points to the file
                                       // Phone.mdb
                                       // in directory Swiss
   String username = "";
   String password = "";

   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.execute("CREATE TABLE MyPhoneBook(firstname char(20),"
               + "surname char(20),phonehome char(15),phonebus char(15))");
   // MyPhoneBook is the name of the table in the database file Phone.mdb
   stmt.close();
   con.close();
   System.out.println("MyPhoneBook table has been successfully created.");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

We insert data in the table called `MyPhoneBook` with the four columns for `firstname`, `surname`, `phonehome`, `phonebus` using the `INSERT` command.

```
// InsertData.java

import java.sql.*;

public class InsertData
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.executeUpdate("INSERT INTO MyPhoneBook " +
                      "VALUES('John','Smith','233-4567',null)");
   stmt.executeUpdate("INSERT INTO MyPhoneBook " +
                      "VALUES('Lea','Cooper','345-4567','908-4567')");
   // MyPhoneBook is the name of the table in the database Phone.mdb
   stmt.close();
   con.close();
   System.out.println("Data inserted successfully");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

We use the SELECT statement to retrieve data from the table called `MyPhoneBook`.

```
// SelectData.java

import java.sql.*;

public class SelectData
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT phonehome, phonebus FROM " +
                  " MyPhoneBook WHERE firstname = 'Lea'");

   while(rs.next())
   {
   System.out.println(rs.getString(1) + rs.getString(2));
   }

   stmt.close();
   con.close();
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

By updating the table we can modify selected rows and columns in the table called MyPhoneBook.

```
// UpdateData.java

import java.sql.*;

public class UpdateData
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.executeUpdate("UPDATE MyPhoneBook " +
                      "SET phonehome = '333-6666' WHERE " +
                      "surname = 'Smith' AND " +
                      "firstname = 'John'");

   System.out.println("Update is done successfully");

   stmt.close();
   con.close();
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

Doing an update with a `PreparedStatement` for the table `MyPhoneBook`.

```
// Prepared.java

import java.sql.*;

public class Prepared
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   PreparedStatement pstmt = con.prepareStatement("UPDATE MyPhoneBook " +
                      "SET phonehome=?, phonebus=? WHERE " +
                      "firstname=? AND surname=?");

   pstmt.setString(1,"888-9999");
   pstmt.setString(2,"555-7777");
   pstmt.setString(3,"Lea");
   pstmt.setString(4,"Cooper");
   pstmt.executeUpdate();

   System.out.println("Update is done successfully");

   pstmt.close();
   con.close();
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

Deleting rows from the table `MyPhoneBook`..

```
// DeleteData.java

import java.sql.*;

public class DeleteData
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.execute("DELETE FROM MyPhoneBook WHERE " +
                 "firstname = 'Lea'");
   stmt.close();
   con.close();
   System.out.println("Row has been successfully deleted from the table");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

In the following program we show how to use the

`ALTER TABLE ... ADD COLUMN`

statement. We add a new column called `email` to our table.

```
// AlterTable.java

import java.sql.*;

public class AlterTable
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   {
   System.out.println(cnfe);
   }

   String url = "jdbc:odbc:PhoneBook";

   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.executeUpdate("ALTER TABLE MyPhoneBook ADD COLUMN email char(10)");
   stmt.close();
   con.close();
   System.out.println("Data inserted successfully");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

We can delete (drop) a tables from the database. We drop the table `MyPhoneBook`. The file `Phone.mdb` of course still exists.

```
// DropTable.java

import java.sql.*;

public class DropTable
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.execute("DROP TABLE MyPhoneBook");

   stmt.close();
   con.close();
   System.out.println("Table has been successfully dropped");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

Implementation of a graphic user interface for JDBC.

```
// JDBCGUI.java

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

public class JDBCGUI extends JFrame implements ActionListener
{
 private JTextField firstname, surname, phonehome, phonebus;
 private JButton newrec, ret, upd, del;
 private String dbfirstname, dbsurname, dbphonehome, dbphonebus;
 private String url = "jdbc:odbc:PhoneBook";
 private String username = null;
 private String password = null;

 public JDBCGUI()
 {
 addWindowListener(new WindowAdapter()
 {
 public void windowClosing(WindowEvent e) { System.exit(0); }});

 getContentPane().setLayout(new GridLayout(4,1));

 firstname = new JTextField(8); surname = new JTextField(8);
 phonehome = new JTextField(8); phonebus = new JTextField(8);

 newrec = new JButton("Insert");  newrec.addActionListener(this);
 ret = new JButton("Select");  ret.addActionListener(this);
 upd = new JButton("Update");  upd.addActionListener(this);
 del = new JButton("Delete");  del.addActionListener(this);

 JPanel panel = new JPanel();
 panel.add(new JLabel("Firstname"));
 panel.add(firstname);
 panel.add(new JLabel("Surname"));
 panel.add(surname);
 panel.add(new JLabel("Home Phone#"));
 panel.add(phonehome);
 panel.add(new JLabel("Business Phone#"));
 panel.add(phonebus);
 getContentPane().add(panel);
```

```
panel = new JPanel();
panel.add(newrec); panel.add(ret);
panel.add(upd); panel.add(del);
getContentPane().add(panel);

setTitle("JDBCGUI");
setSize(400,300);
setLocation(100,100);
setVisible(true);

try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(ClassNotFoundException cnfe)
{
JOptionPane.showMessageDialog(this,cnfe,"Driver Error",
                              JOptionPane.ERROR_MESSAGE);
}
} // end default constructor JDBCGUI()

public void actionPerformed(ActionEvent e)
{
if(e.getSource() == newrec) { insert(); }
if(e.getSource() == ret) { select(); }
if(e.getSource() == upd) { update(); }
if(e.getSource() == del) { delete(); }
} // end actionPerformed

public void insert()
{
dbfirstname = firstname.getText();
dbsurname = surname.getText();
dbphonehome = phonehome.getText();
dbphonebus = phonebus.getText();

try
{
Connection con = DriverManager.getConnection(url,username,password);
Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO MyPhoneBook " +
"VALUES('"+dbfirstname+"','"+dbsurname+"','"+dbphonehome+"','"+dbphonebus+"')");

firstname.setText("");
surname.setText("");
```

```
phonehome.setText("");
phonebus.setText("");

stmt.close();
stmt.close();
} // try block end
catch(SQLException sqle)
{
JOptionPane.showMessageDialog(this,sqle,"SQLERROR",JOptionPane.ERROR_MESSAGE);
}
} // end insert()

public void select()
{
dbfirstname = firstname.getText();
dbsurname = surname.getText();
try
{
Connection con = DriverManager.getConnection(url,username,password);
Statement stmt = con.createStatement();
ResultSet rs =
   stmt.executeQuery("SELECT phonehome, phonebus FROM MyPhoneBook " +
  "WHERE firstname = '"+dbfirstname+"'AND surname = '"+dbsurname+"'");

while(rs.next())
{
phonehome.setText(rs.getString(1));
phonebus.setText(rs.getString(2));
}
stmt.close(); con.close();
} // end try
catch(SQLException sqle)
{
JOptionPane.showMessageDialog(this,sqle,"SQLERROR",JOptionPane.ERROR_MESSAGE);
}
} // end select()

public void update()
{
dbfirstname = firstname.getText();
dbsurname = surname.getText();
dbphonehome = phonehome.getText();
dbphonebus = phonebus.getText();

try
```

```
 {
 Connection con = DriverManager.getConnection(url,username,password);
 Statement stmt = con.createStatement();
 stmt.executeUpdate("UPDATE MyPhoneBook SET " +
  "phonehome = '"+dbphonehome+"' AND surname = '"+dbsurname+"'");

 stmt.close(); stmt.close();
 } // try block end
 catch(SQLException sqle)
 {
 JOptionPane.showMessageDialog(this,sqle,"SQLERROR",JOptionPane.ERROR_MESSAGE);
 }
 } // end update()

 public void delete()
 {
 dbfirstname = firstname.getText();
 dbsurname = surname.getText();

 try
 {
 Connection con = DriverManager.getConnection(url,username,password);
 Statement stmt = con.createStatement();
 stmt.execute("DELETE FROM MyPhoneBook " +
  "WHERE firstname = '"+dbfirstname+"' AND surname = '"+dbsurname+"'");

 firstname.setText(""); surname.setText("");
 phonehome.setText(""); phonebus.setText("");

 stmt.close(); stmt.close();
 } // try block end
 catch(SQLException sqle)
 {
 JOptionPane.showMessageDialog(this,sqle,"SQLERROR",JOptionPane.ERROR_MESSAGE);
 }
 }

 public static void main(String[] args)
 { new JDBCGUI(); }
}  // end class JDBCGUI
```

Assume we added a new column `DateInserted` of data type `Date` to the table `MyPhoneBook` we can insert a new row into the table `MyPhoneBook` with the present date as follows.

```
// InsertData1.java

import java.sql.*;
import java.util.Date;

public class InsertData1
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   Date now = new Date();

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   stmt.executeUpdate("INSERT INTO MyPhoneBook " +
                      "VALUES('Jolly','Pulli','233-4567','908-4567',now)");
   // MyPhoneBook is the name of the table in the database Phone.mdb
   stmt.close();
   con.close();
   System.out.println("Data inserted successfully");
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

To display the dates we use the following program.

```
// SelectData1.java

import java.sql.*;
import java.sql.Date;
import java.util.*;

public class SelectData1
{
   public static void main(String[] args)
   {
   try
   {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // loads the driver
   }
   catch(ClassNotFoundException cnfe)
   { System.out.println(cnfe); }

   String url = "jdbc:odbc:PhoneBook";
   String username = "";
   String password = "";

   try
   {
   Connection con = DriverManager.getConnection(url,username,password);
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT DateInserted FROM MyPhoneBook");

   while(rs.next())
   {
   System.out.println(rs.getDate(1));
   }

   stmt.close();
   con.close();
   }
   catch(SQLException sqle)
   {
   System.out.println(sqle);
   }
   } // end main
}
```

5.6 Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. For example in JDBC we can call the methods of the classes `DatabaseMetaData` and `ResultSetMetaData` to get this information.

Thus metadata is the data about data. Metadata describe how and when and by whom a particular set of data was collected, and how the data is formatted. Metadata is essential for understanding information stored in data warehouses.

5.7 JDBC 3.0

JDBC 1.0 – introduced with JDK 1.1 – had minimal database functionality. JDK 1.2 delivered JDBC 2.0, which contained enhanced features like scrollable result sets, batch updates, `Blob` and `Clob` support, `Array` type, user defined types (UDTs), structured types (`Ref` type), and distinct types.

`public interface Array`

provides the mapping in the Java programming language for the SQL type `ARRAY`.

`public interface Ref`

provides the mapping in the Java programming language of a SQL `REF` value, which is a reference to an SQL structure type value in the database.

Then came the development of the JDBC 2.0 optional package: a standard extension that provides the `DataSource` class (which uses JNDI to connect to any kind of data, such as flat files or spreadsheets), connection pooling, distributed transactions, and `RowSet`s (a higher-level interface on top of `ResultSet`).

For its part, JDK 1.4 has major JDBC changes in store. It is being bundled with JDBC 3.0 including the packages `java.sql` and `javax.sql`.

JDBC 3.0 contains the JDBC 2.0 classes and the JDBC optional package classes. That means the optional package is being merged into standard JDBC, making the Java platform more flexible and complete. The SQL99 standard has also been finalized, so JDBC 3.0 will also attempt to be consistent with that standard. JDBC 3.0 will not support the whole SQL99, but the features that are implemented will be consistent with the standard.

A mechanism for connecting JDBC with a connector architecture using the Service Provider Interface (SPI) is also in the works. The connector architecture is a general way to connect to enterprise information systems (EIS), such as ERP systems, mainframe transaction processing systems, and hierarchical databases. The connector architecture specification defines a set of contracts that allow a resource adapter to extend a container in a pluggable way.

RowSets are a Tabular Data Datatype. A `javax.sql.RowSet` object encapsulates a set of rows that have been retrieved from a tabular data source. A `RowSet` object is simply a `ResultSet` object that can be function as a JavaBeans component.

```
public interface RowSet extends ResultSet
```

Since the `RowSet` interface includes an event notification mechanism and supports getting and setting properties, every `RowSet` object is a JavaBeans component. This means, for example, that a `RowSet` can be used as a JavaBeans component in a visual JavaBeans development enviroment. As a result, a `RowSet` instance can be created and configured at design time, and its methods can be executed at run time. Let `rset` be an object of `RowSet`. An example is

```
rset.setDataSourceName("jdbc/SomeDataSourceName");
rset.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
rset.setCommand("SELECT Model, Make FROM Cars");
```

The `javax.sql` package also introduces the class `javax.sql.RowSetEvent`. The class `RowSetEvent` is used when an event occurs to a `RowSet` object. A `RowSetEvent` is generated when a single row in a `RowSet` is changed, the whole `RowSet` is changed, or the `RowSet` cursor moves.

Three new row sets could be implemented:

`JDBCRowSet` makes the JDBC driver look like a JavaBean component. We can use it to make JDBC applications with GUI tools.

`CachedRowSet` loads the data into a cache and disconnects the SQL connection from the database. Hence, we can pass that cached row set between tiers of the application. It also helps to optimize the database connections. Any changes made to the cached data can later be reflected back into the database.

`WebRowSet` lets us convert JDBC data (with its properties and metadata) to XML, which we can use anywhere. We can also convert the XML back to data and save any data changes back to the database. This will be very useful with upcoming XML communication protocols and tools. We can also use custom readers and writers with these row sets to provide our own mechanisms to read and write data. For instance, we can provide custom mechanisms to read data from a nonrelational

database or we can provide our own conflict-resolution mechanism while saving data.

Other features of JDBC 3.0 include:

1) Savepoint support (the ability to roll back transactions to designated savepoints)

2) Connection pool configurations (added properties to describe how `PooledConnection` objects should be pooled)

3) Reuse of prepared statements with connection pools (in JDK 1.3, when we close a SQL connection, the prepared statement dies with it, but now the statement will be independent)

4) Retrieval of parameter metadata (the new interface `ParameterMetaData` describes the number, type, and properties of parameters to prepared statements)

5) Retrieval of auto-generated key columns

6) Multiple open result sets on a statement

7) `BOOLEAN` data type

8) `DATALINK` data type (allows JDBC drivers to store and retrieve references to external data)

9) Updateable `Blob`, `Clob`, `Array`, and `Ref` objects. The methods `updateBlob()`, `updateClob()`, `updateArray()` and `updateRef()` have been added to the `ResultSet` interface

10) Transform groups and type mapping (defines how UDT can be transformed to a predefined SQL type and how that is reflected in metadata)

11) DatabaseMetaData APIs that retrieve SQL type hierarchies

12) Holdable cursor support

An application of interface `Savepoint` would be

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO Tab1 (Col1) VALUES " +
           "('FIRST')");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO Tab1 (Col1) " +
                          "VALUES ('SECOND')");
...
conn.rollback(svpt1);
...
conn.commit();
```

Security

Also bundled in the new release is the optional Java Secure Socket Extension (JSSE) package. JSSE implements Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols.

Java Authentication and Authorization Service (JAAS) is a new API that lets us establish access control on a per-user basis. JAAS can be used in situations when an administrator needs more menus and options than a normal user.

Java Cryptography Extension (JCE) is another standard extension that will be bundled with Merlin. JCE provides functionality for encryption, key generation and key agreement, and message authentication code (MAC). Fortunately, JCE can now be exported from the US and Canada because of more lenient security regulations. Now, applications that use cryptography can be exported, so people in other countries can benefit from the Cryptography API. That means only the policy files that define the cryptographic strength are altered; then the application can be exported to international customers. Public Key Cryptography Standards (PKCS) support is also being finalized.

A reference implementation for the Certification Path Building and Verification API is already under way. This will allow access to certificates (and hence, public keys) via LDAP/JNDI. This benefits applications that need to deal with SSL, signed code, S/MIME, and so forth.

Chapter 6

Object-Oriented Databases

6.1 Introduction

Object oriented databases emerged in the mid-80's in response to the feeling that relational databases were inadequate for certain classes of applications. The problem of relational databases is decribed metaphorically in the literature as *impedance mismatch*. The two components whose impedance do not match are the application and the database. The application is procedural, deals with one item at a time and creates complex data structures. The database, conversely, is declarative, deals with tuples in sets instead of individually and holds data in flat tables. Both approaches have their good reasons and strengths in their respective fields, but bringing the two paradigms together in an application that uses the database exposes this mismatch.

Roughly speaking, a database system is based on the idea that data and associated programs are separated, and thus is very different from a typical file system. However, seeming at first glance to be a contradiction, object-oriented databases aim to achieve integration of data and programs, but this is now achieved in an adequate methodical framework, which borrows much from abstract data types.

The solution proposed by the supporters of object oriented databases is to bring the type systems of application and database closer to each other, so that a data item defined in an application can be stored in the database without being first unwound, and to integrate the programming language of the application with the database's DML.

Relational databases do not normally provide us with the ability to define our own data types. Some authors argue that this is a deficiency of current implementations and not a prescription of the relational model. With object database we can define arbitrarily complex data types like their programming language counterparts, possibly nested in *is-a* and/or *has-a* hierarchies. Sets, list, and bags (multisets) and other containers are used, among other things, to represent the result of a query which returns several objects.

The number of relational database system (for example Oracle) provide an interactive SQL-based interface to the database as well as an embedded SQL-based interface for an application programming interface. Embedded SQL interfaces provide a means to bridge the differences in syntax and semantics between the host programming language and the database language, namely SQL. There are five serious problems with embedding query statements in a general-purpose programming language:

1. The data model of the database and the type system of the programming language do not match. For example, programming languages do not provide sets as a basic type, and sets of objects returned by a query cannot be manipulated directly by the programming language constructs.

2. Since there is no type system spanning the application and the query code, limited type checking can be done accross the junction.

3. Queries can be formulated only on persistent data objects, and cannot be formulated against transient objects and persistent objects after they have entered the application address space.

4. Query and programming language statements cannot be freely combined.

5. The syntax and semantics of the two languages are completely different, and the programmer needs to learn to be aware of the differences.

One of the main motivations for the development of OODBMSs and better programming environments is the *impedance mismatch problem*. The main bottleneck to the productivity of the application programmer is the impedance mismatch between the programming language and the query language. This impedance mismatch cannot be solved by redefining the database box, but by mixing database technology and programming language technology to build a complete system which will have the functionalities of a DBMS and of a programming language.

A method is a piece of code associated with an object that has privileged access to the object's state. Checking whether an object has or does not have a property which is derived from its state may sometimes be done more efficiently by invoking a specially written method than by running a query on the attributes. Stored procedures (stored in the database, that is) which can be executed on the server side were for certain aspects a precursor of this idea. Rules is a generic name for code that is automatically activated when certain events occur. A form of constraint programming. Finally, one of the most important features of object-oriented databases is that the commonality between the application's and the database's type systems preserves the strong typing in the communication between the two. The advantages of the object database approach are that applications which define a rich data type system for their data structures can carry this over to the database when these

data structures are made persistent. The impedance mismatch is eliminated as the database becomes a persistent extension to the language rather than an external service that one has to talk to through a narrow and limited interface.

Object systems introduce pointers linking objects to each other. This in turn promotes a procedural style of navigation among the data items that can be seen as a step backwards from the higher-level declarative approach introduced by relational systems.

Encapsulation is the technique that an object's internal state is only accessible through its designated methods. While encapsulation is a valuable modular technique, it presents problems with queries. For example, even if a `Employee` class has an extensive set of methods for `hire_employee`, `change_salary`, `fire_employee` etc.. we might still not be able to answer a simple question like "Who is the manager of this employee?" unless an appropiate method has been written. There is a tradeoff between giving up encapsulation by exposing all the object's state through methods or by allowing queries to violate encapsulation under certain circumstances and severely restricting the class of possible queries.

Closure of the relational algebra (the property that operators take relations as their arguments and give out relations as their results, thus allowing nested queries) is an important property that is normally lost in object systems. Finally, but perhaps most importantly, object databases do not rest on a formal mathematical base like their relational predecessors. A powerful instrument of analysis, deduction and insight is lost.

ODMG is a consortium of object database vendors led by a small focused group of experts. Their aim is to produce an open standard for object database management systems. The ODMG has a strong CORBA flavour, altough not derived from OMG, ODMG is affiliated with OMG and OMG has approved ODMG-93 as a standard interface for a Persistence Service. The ODL (Object Definition Language) is an extension of CORBA's IDL and as such has a C++ flavour. The OQL (Object Definition Language) has a syntax based on SQL. The OML (Object Manipulation Language) is a C++ language binding which provides access to database functions and persistence of objects defined with ODL. Among other things it defines standard container classes. An object-oriented database management system (OODBMS), sometimes shortened to ODBMS for object database management system), is a database management system (DBMS) that supports the modelling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. There is currently no widely agreed-upon standard for what constitutes an OODBMS, and OODBMS products are considered to be still in their infancy. In the meantime, the object-relational database management system (ORDBMS), the idea that object-oriented database concepts can be superimposed on relational databases, is more commonly encountered in available products. An object-oriented database interface

standard is being developed by an industry group, the Object Data Management Group (ODMG). The Object Management Group (OMG) has already standardized an object-oriented data broking interface between systems in a network. A possible definition of an OODBMS could be:

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features:

```
persistence
secondary storage management
concurrency
recovery
ad hoc query facility
```

The second criteria translates into eight features

```
complex objects
objects identity
encapsulation
types or classes
inheritance
overriding combined with late binding
extensibility and computational completeness
```

Conventional database models provide adequate support for applications where the structure of the database tends to be fairly static once the large files of records are set up. Such databases typically need to provide the following:

1) Persistence of data
2) Data sharing
3) Protection of data
4) Acceptable levels of performance

OODBs have to provide all the features listed above, and a lot more features demanded by complex applications that are not currently met by conventional database systems. The strengths of conventional databases, especially relational DBMSs, can be summarized as follows:

1) Suitability for online transaction processing
2) Performance with large amounts of structurally regular data
3) A more or less mathematically tractable data model
4) Industry standard data definition and data manipulation languages that are more or less adhered to
5) A large body of experience and knowledge in the data management community

6.2 Object-Oriented Properties

The need for the object paradigm springs from both advances in software sophistication and scope. Object orientation promises to:

(1) facilitate the development and maintenance of applications that incorporate large amounts of small heterogenous data types (e.g. text, graphics, images, voice and facsimiles),

(2) to enhance productivity in the long run by permitting programming pieces or modules that can be very flexibly coupled and re-used,

(3) to empower the development of software that requires a high level of integration and connectivity.

Remark. Graphics, images and voice are stored as Blob (binary large objects). For example Microsoft Access sees the Blob field as an OLE object (Object-linking and embedding). It is a locator of a large binary file outside the database. Instead of holing the data in the database, a file reference contains a link to the data.

From a developers perspective, this implies software that is easier to create, simpler to use, far more reliable, and less costly to maintain or evolve in increments. From a users perspective, this implies software that has enhanced functionality.

Traditional databases incorporate two kinds of information: data and schema. The schema is the description of the structure of the data contained in a database. It details the set of attributes or fields that comprise the database. The bulk of the database is simple data such as integer, string and boolean types that are arranged into records consistent with the schema.

The architecture of a database refers to the way data models are implemented and combined with application programming languages. Database models can be distinguished and classified by the constraints placed on the kinds of data and schema they are permitted to contain. We distinguish four data models classes. These are:

(1) relational

(2) extended relational

(3) functional

(4) object-oriented.

The relational model is a useful benchmark class because of its extensive documentation, popularity and importance. There is only one data structure:

a table with rows and columns containing data of specified –usually simple– types and the query language is based on a few simple operations on tables.

Extended relational models retain the basic relational table and query language but also incorporates some concept of "object" and has the ability to store procedures as well as data in the database. Functional data models use a data-access language based on mathematical function notation with a declarative functional query language to define the function values.

The object-oriented model originated with the object-oriented programming paradigm,

C++, Java and Smalltalk.

This model is based on the modularization of abstract data-type declarations into classes. The distinguishing characteristics of this model category will be more fully elaborated in the following.

The data model classification scheme appears somewhat tenuous because many of the data models are being combined into hybrids that then exhibit a combination of data and schema features. Also, the term "object-oriented" is often loosely used to refer to systems based on all these models if they incorporate any concepts from the object-oriented paradigm.

Databases have long been an essential part of most software packages. They are a critical part of accounting systems and production record systems. In the earlier developed systems, databases were an integral part of the overall package and the schema closely tied to the programming language used to develop the package. In recent years, the trend has been toward the use of database management systems to store, retrieve and manipulate data. Relational systems are the most common approach employed by current database management systems. Relational systems involve the structuring of data in tabular form and, thus, these databases are often referred to as "tables".

6.3 Terms Glossary

Abstract class. A class that acts as a template for other classes. It is usually used as the root of a class hierarchy. The abstract class represents a concept; classes derived from it represent implementations of the concept.

Activation. Copying the persistent data and associated persistent form of methods into the executable address space to invoke the operation of methods on the data.

Actor. A model of concurrent computation in distributed systems. Computations are carried out in response to communications sent to the actor system.

Application objects. Applications and their components that are managed within an object-oriented system. Operations on such objects include open, install, move, and remove.

Associations. Associations are constructs that logically link objects. Composite objects are logical objects that are made out of simpler objects by connecting them with associations. The cardinality of associations is the number of objects involved on both sides of an association. Assocations can be categorized into four groups according to their cardinality: $1:1$ (one-to-one), $1:m$ (one-to-many), $m:1$ (many-to-ome), $n:m$ (many-to-many).

Atomicity. The property that ensures that an operation either changes the state associated with all participating objects consistent with the request, or changes none at all. If a set of operations is atomic, then multiple requests for those operations are serializable.

Attribute. A conceptual notion employed to express an identifiable association between the objects and some other entity or entities.

Behavior. The observable effects of performing the requested service.

Binding. The process of selecting a method to perform a requested service and selecting the data to be accessed by the method.

Class. A specification of the behavioral rules that all possible instances must be compliant with. It defines the data and the methods permissible within an instance (object) grouping. Template from which objects can be created. It is used to specify the behavior and attributes comon to all objects of the class.

Class inheritance. The process of defining a class in terms of some existing class, by means of incremental modification.

Client object. An object making a request for a service.

Complex objects. Complex objects, also referred to as aggregate objects, are formed by applying object constructors to simple objects such as integers, characters, and floats that are provided by all systems. The object space can be considered to be built on top of atomic objects of the types integer, float, character, boolean, string etc.

Data abstraction. Viewing data objects in terms of the operations with which they can be manipulated rather than as elements of a set. The representation of the data object is irrelevant.

Data independence. Physical storage of data is isolated from the user's perception of it.

Dynamic binding. Binding that is performed at run time, after the request has been issued.

Encapsulated. Data and methods isolated from external access and indiscriminate modification (private, protected, public).

Hypermedia. Using a computer to integrate diverse communication channels (for example text, sound, graphics, animation and motion video) generated by the computer or other devices.

Inheritance. Ability to define descendent classes that have common data or methods that can be incrementally altered to form another object class.

Instance. A specific case of occurrence in a class of objects.

Late binding. Delaying the linking of program code until the program is executing.

Metaobject. An object that represents a type, operation, class, method, or other object model entity that describes objects.

Message. The process of invoking an operation on an object. In response to a message, the corresponding method is executed in the object. A message to an object specifies what should be done. A message can be sent by clients of the object-application programs, another object, or another method within the same object.

Methods. Implementations of the operations relevant to a class of objects. The part of an object that performs an operation is termed a method. Methods are invoked in response to messages.

Multimedia. Using a computer to control and integrate different devices such as videodisc, videotape, CD-audio or CD-ROM into a single project.

Multiple inheritance. When a class inherts from more than one base class. C++ allows multiple inheritance.

Objects. Chunks of programming code and data that can behave like things in the real world. An object could be an animal, a shed, a stomach, or a business form. A combination of data and the collection of operations that are implemented on the data. Also a collection operations that share a state. The representation of a real-world entity. An object is used to model a person, place, thing, or event from the real world. It encapsulates data and operations that can be used to manipulate the data and responds to requests for service.

Object identity. Each object in the real world has an existence or identity which is independent of its actual values. A system-supported identity is maintained by assigning a so-called object identifier to each object.

Object-oriented system. In pure form contains only objects and methods. Objects provide the nouns against which the message verbs act. Permits viewing things as interrelationships between and among system components.

OODB. Object-oriented database.

OODBMS. Object-oriented database management system that can be used to store and retrieve objects.

ORB. Object request broker, the facility that provides the means by which objects make and receive requests and responses.

Persistence. The ability of data to exist beyond the scope of the program that created it. Persistence can be visualized in two ways. Values are persistent, implying that they retain their identity and their persistence independent of who points to them. Variables are persistent, which implies that persistence is retained only for values pointed to be persistent variables and, perhaps, that identity is associated more with the source of the pointer than with the destination of the pointer.

Persistent data. Data remains in storage even when DBMS software is not operating.

Polymorphism. Ability to issue the same command to different objects with the object then determining the appropriate action.

Shared data. Ability to have multiple simultaneous use of the data.

Operator overloading. The ability of operators to be polymorphic and define actions congruent with its usage context.

6.4 Properties of an Object-Oriented Database

The features of object-oriented databases should be:

Support for complex objects. The ability to build objects by applying constructors to basic objects. The minimum set of constructors are set, list and tuple(i.e. record).

Object identity. An object has an existence independent of its value.

Encapsulation. A mechanism whereby the specification and the implementation of an object can be separated. This is achieved if the data and the method implementations are hidden and only the operations are visible to the programmer.

Support for types or classes. A type summarizes the common features of a set of objects with the same characteristics and can be used at compilation time to check for programming errors. A class has similar functionality for run-time execution.

Class or type hierarchies. Any subclass or subtype will inherit attributes and methods from its superclass or supertype.

Overriding, overloading and late binding. Operations should apply to objects of different types (overloading). The implementation of an operation will depend on the type of object it is applied to (overriding). The implementation code can't be referenced until run-time (late binding).

Computational completeness. The data manipulation language of the database should allow a user to express any computable function (a contrast from SQL).

Extensibility. The system should provide facilities to define new types or classes which can be manipulated in exactly the same manner as those built into the system.

Persistence. Data must remain after the creation and transformation processes have terminated.

Secondary storage management. Techniques that transparently manage secondary storage to improve system performance and make the handling of large volumes of data possible.

Concurrency. Multiple simultaneous use of objects should be permitted.

Recovery. The system should provide a mechanism to handle contingencies.

Ad hoc query facility. The database should provide a high-level, efficient, application independent query facility but not necessarily a query language.

Although other definition criteria are likely to emerge, this list provides a relatively complete set of feature requirements for defining an object oriented database.

Object-oriented database languages are expected to fulfil the same requirements as languages based on other data models, particular the following.

1) Universality. The language should not be designed with a view of being applied for a specific purpose; instead it should be universally applicable.

2) Descriptivity. The language should be characterized by a high level of abstraction independent of implementation details, and in particular it should support set-oriented access.

3) Optimizability. For an underlying system it must be possible to optimize expressions of the language prior to execution; therefore, appropriate optimization rules and strategies must exist.

4) Closedness. It must be possible to describe every result of a language expression within the given data or object model; this ensures that the result of a query can be used as input for a subsequent query.

5) Completeness. Every concept of the data model in question must have a processing counterpart.

6) Genericity. The language should contain generic operators (for example, selection, projection and set operations) which can be applied to values depending only on the type structure.

7) Expressive power. The language should surmount the restrictions imposed on relational languages; that is, recursive traversing of objects sets should be possible and Turing-completness should also be guaranteed.

8) Extensibility. The language should support both user-defined and system-defined types.

6.5 Example

As an example consider data related to an individual cow. The data that is to be maintained on each cow (a record) is identified in the columns (fields) of the table. If additional data are to be kept on a cow, more columns are added. In this case the elementary table entity is the cow and the data items related to each cow (the columns) are the attributes.

The rows of the table are the data for any cow. For each cow added to the database, a row is added to the table. Each row is a record (i.e. tuple). Within the constraints imposed by the hardware and software, the table can be as large as needed.

In the example illustrated in table 1 with cow information, the attributes of the cow entity are fixed in nature (e.g., birth date). How is cow data handled when the number of observations (e.g., freshening dates or milk production by milking) can be quite variable? One approach would be to add columns to the cow record to allow for the most extreme case. However, this is not a very effective approach because the record structure is inefficiently used and data access and use becomes very cumbersome. Thus, it is better to establish a new entity that has links to the desired tracking entity, in this case the cow, for the entity that has multiple observations. Table 2 illustrates a milking entity in which there would be multiple observations for each cow. In this case all the desired data related to the milking (e.g., cow I.D., date, milk production, etc.) are the attributes of the milking entity. The link to the cow information data is the cow I.D. This database is relational in nature because the data is expressed in tabular form. Since the data is linked, it becomes hierarchial in nature.

For other information related to the cow, such as calving information, health data, and so forth, additional tables or files would be created. All the tables related to a particular topic constitute a database.

COW - INFORMATION FILE

COW_ID	BARN_NAME	BIRTH_DATE	SIRE	DAM ...
123	Mary	12/1/87	Q	418 88...
434	Susie	4/1/86	K	88 34...
111	Blackie	6/1/88	Q	433 106..
.....				
136	Jane	6/4/88	Q	178 91...

Table 1. Relational Data File of Cow Information

To retrieve the desired information from its persistent storage location, usually several tables or files are employed; thus to determine the total production for a particular cow in a selected lactation, the cow information file is needed to supply the code (i.e., cow I.D.) needed for linking the data together and the lactation file would be used to obtain the beginning and ending dates of the lactation and finally milking information file would supply all the milking records for that cow within the dates of the lactation selected. Relational systems can perform this type of operation through the use of commands. However, to extract the same information from files integrated with a unique software package, it would require the writing of special computer codes to accomplish the task. Furthermore, each new information request would involve writing new computer codes. Thus, one can easily see the value of database management systems.

MILKING INFORMATION FILE

COW_ID	DATE	TIME	MLK_MIN	LBS_MLK	QT_COND..
434	11/4/91	5:16	14.1	38	13..
123	11/4/91	5:17	16.2	41	18..
111	11/4/91	5:17	17.1	32	14..
.....					
123	11/4/91	18:15	15.1	41	14..
111	11/4/91	18:15	16.6	42	19..
434	11/4/91	18:16	16.4	36	21..
.....					

Table 2. Relational Data File of Milking Information

Relational databases have become more refined through time. They have more features and are capable of processing increasingly complex commands. With the widening adoption of SQL as a query language for database management systems, relational systems have become more standardized. All these trends reflect advances with respect to data management and control. Once the data has been captured in the database, it can be accessed and used for multiple purposes. Whereas those tightly coupled to a software package have their data use restricted to the primary purpose for which the software package was originally developed.

Relational systems, although they have many strengths, also have some shortcomings and limitations. They tend to be rather limited in the type of data that they can store, retrieve and manipulate. They are often limited to number and character strings (as illustrated in Tables 1 and 2). Also, the field size is often fixed which limits their ability to handle free form text and numbers. For example, for the cow information file discussed earlier, if the farm keeping this type of information is producing purebred cattle, it may be desirable to have photographic images of

the cow as part of the database. For the milk record, it may be desirable to have the recent pattern of conductivity measurements from the quarters stored as an estimable function (e.g., a spline function) which could be easily used with artificial intelligence techniques such as neural networks to recognize an emerging mastitis problem before it reaches a serious stage. Most of the relational systems do not possess the capabilities to support schema containing these data types.

Finally, the data fields of relational systems are quite rigid. For example, the milk production illustrated in Table 2 is expressed in pounds. If the data are needed in the form of kilograms by another software package, it would need to explicitly be converted. Most relational systems do not handle these conversions transparently.

To address the shortcomings and deficiencies of relational systems, object-oriented databases have started to emerge. They differ from the relational systems in the sense that the real-world or abstract entities are modeled in the database. These entities or objects could be animals, pieces of equipment, documents, etc. Objects can contain behavioral components as well as structured data. The behavioral dimension of objects is concerned with methods and encapsulation of object data.

What should the increased functionality of object-oriented databases permit? For example, the sire selection process could be enhanced by having "Bull Books" incorporated into a database that could be incrementally updated and conveniently shared and/or distributed. The user could request a visual image of a bull and his daughters, a text or graphical illustration of pedigree and genetic profile information showing the magnitude and precision of selected trait estimates and decision support assistance in applying user tailored screening criteria and rules. Another application possibility might be an animal sales catalog. In addition to text, graphics and image display, the playback of digitized –maybe even animated!– voice messages providing a descriptive narrative of the animal(s) can also be envisioned as being a useful communication and, therefore, marketing tool. In more mundane –but more typical– transactions recording, manipulating, storing and retrieving applications, having persistent facsimiles of transaction related documents in addition to traditional data types (i.e., numbers, strings, etc.) might also be highly useful in some instances. Finally, the frequently cited computer-aided applications realm (e.g. computer-aided software engineering (CASE), mechanical computer- aided design (MCAD), electronics computer-aided design (ECAD), computer-aided publishing (CAP), etc.) contains many enhanced functionality potentials.

Another possible area of application of object-oriented databases (OODB) is the task of storing and organizing the large amounts of diverse forms of information related to a particular subject matter area. The impact on extension and teaching programs could be vast. For example, in a National Pork Producers Council publication, a set of standards regarding the definition and calculation methods of numerous performance measures for the swine industry has been promulgated. These standards are critical for communication in the industry. These metrics could be objects or instances of performance measures classes in an OODB that would contain the data and methods for calculating each performance measure, as well as, descriptive or other related information about them.

What advantages are to be gained from handling information as objects? To address this question, assume a knowledge-based system –one built using object-oriented programming techniques– is being used to evaluate the production and economic performance of an integrated swine operation. One criterion considered by the knowledge-based system is "pigs weaned per litter." Since the formula for calculating this value is encapsulated, the user does not need to be concerned with developing computer codes to make this calculation. If a user of the software needs to know more about how certain factors are calculated, or unsure of a data item used in the analysis, or how to recognize a certain condition, the user can easily get assistance since the context sensitive "help messager" could extract the needed information, such as definitions or graphic images, from other objects that are part of the OODB.

Similarly, the recent report on future applications of communication technology in Extension stresses the need of tailoring information to clients' needs and comprehension skills. This implies that the old approaches (e.g., a bulletin), will likely not be acceptable. By using "print-on-demand" technologies coupled with an OODB, information can be extracted and tailored for each request. Again, these objects have originated from traditional sources and are more accessible (indeed they may be some of the same objects used to develop the knowledge based system) they can be selectively utilized to meet each unique information request. Furthermore, since the information is now encapsulated in a persistent database, it can be conveniently refined and updated. Thus, the information obsolescence problem can be reduced relative to our more traditional methods such as bulletins or video tapes.

6.6 C++

C++ is the most popular object-oriented programming language (OOPL). It is an object-oriented extension to C. C++ supports the OOP concepts of

objects, classes, inheritence, multiple inheritence, polymorphims, parametrized types, abstract classes, exception handling.

The C++ class concept can be considered as the generalization of the C feature of a `struct`.

C++ provides an access control mechanism for the operations on objects. The operations are called member functions. Member functions can have one of the following three modes of access:

1) `public`

2) `private`

3) `protected`

Public member functions are accessible by all clients of the object.

Private member functions are accessible only by other member functions of the class.

Protected member functions are accessible only by other member functions and the member functions of classes derived from that class.

The Standard Template Library (STL) adds a number of useful classes, for example the container classes

`vector, list, set, map`

for database manipulations. It also provides algorithm for searching and sorting.

The model of C++ objects can be summarized in the following points.

1) Class instances are objects

2) Pointers to class instances are object names

3) Pointers reveal object identity to clients

4) Base components of derived class instances are not objects

5) Public data members correspond to special operations that return pointers

6) All function invocations are request forms, except for invocations that suppress virtual function lookup, which are direct method invocations

7) Except for virtual functions, each function is a distinct operation

8) An overriding virtual function is a new method for one or more existing operations

9) Virtual functions are generic operations; overloaded functions are not

10) C++ values of of nonclass types are values (request parameters)

11) C++ nonclass types are types

12) Subtyping is defined between pointer types based on class derivation

13) A class whose members are pure virtual functions is an interface

14) A pointer type to an interface class is an interface type

15) Operations are values

16) A C++ function type is an operation signature

6.7 The Object Query Language

An Object Query Language is proposed by the ODMG. The language is based on the following principles and assumptions:

1) It uses the ODMG-93 object model.

2) OQL is a declarative and optimizable language, but not Turing-complete. Therefore, OQL is a classical data query language, and in contrast to SQL3, it is designed as a full programming language.

3) The syntax of OQL is similar to SQL, which means the basic query syntax is the

```
SELECT-FROM-WHERE
```

construct. The syntax of OQL is still subject to refinements and enhancements. In particular, the integration into programming languages (including Java, C++, Smalltalk) needs to be finalized.

4) Unlike SQL, OQL does not favour the set as its primary query medium, but treats tuple structures or lists in the same way as sets.

5) OQL has no explicit update commands or operations, but appropiate methods are supplied for the purpose of updating (that is, inserting, deleting or modifying) objects.

With respect to the third point it should be noted that the syntax of OQL is neither stable nor has it been implemented yet. In its current state Version 1.2 OQL is closely related and based on O_2SQL, the query language of O_2.

The object-oriented database systems Versant

```
www.versant.com
```

and Poet (Fast-Object)

```
www.poet.com
```

include the object query language.

6.8 SQL3 Object Model

6.8.1 Basic Concepts

We describe SQL3, a development of the relational language standard SQL, which will reflect object-oriented properties for the first time. SQL3 may be considered as moving towards object-relational systems. The enhancements included in SQL3 can be divided into four main areas:

relational enhancement

procedural extensions

object-oriented support

call level interface.

ANSI (X3H2) and ISO (ISO/IEC JTC1/SC21/WG3) SQL standardization committees have for some time been adding features to the SQL specification to support object-oriented data management. The current version of SQL in progress including these extensions is often referred to as SQL3 [ISO96a,b]. SQL3 object facilities primarily involve extensions to SQL's type facilities; however, extensions to SQL table facilities can also be considered relevant. Additional facilities include control structures to make SQL a computationally complete language for creating, managing, and querying persistent object-like data structures. The added facilities are intended to be upward compatible with the current SQL92 standard (SQL92). This and other sections of the Features Matrix describing SQL3 concentrate primarily on the SQL3 extensions relevant to object modeling. However, numerous other enhancements have been made in SQL as well. In addition, it should be noted that SQL3 continues to undergo development, and thus the description of SQL3 in this Features Matrix does not necessarily represent the final, approved language specifications.

The parts of SQL3 that provide the primary basis for supporting object-oriented structures are:

user-defined types (ADTs, named row types, and distinct types)

type constructors for row types and reference types

type constructors for collection types (sets, lists, and multisets)

user-defined functions and procedures

support for large objects (BLOBs and CLOBs)

One of the basic ideas behind the object facilities is that, in addition to the normal built-in types defined by SQL, user-defined types may also be defined. These types may be used in the same way as built-in types. For example, columns in relational tables may be defined as taking values of user-defined types, as well as built-in types. A user-defined abstract data type (ADT) definition encapsulates attributes and operations in a single entity. In SQL3, an abstract data type (ADT) is defined by specifying a set of declarations of the stored attributes that represent the value of the ADT, the operations that define the equality and ordering relationships of the ADT, and the operations that define the behavior (and any virtual attributes) of the ADT. Operations are implemented by procedures called routines. ADTs can also be defined as subtypes of other ADTs. A subtype inherits the structure and behavior of its supertypes (multiple inheritance is supported). Instances of ADTs can be persistently stored in the database only by storing them in columns of tables.

Collection types for

sets

lists

multisets

have also been defined. Using these types, columns of tables can contain sets, lists, or multisets, in addition to individual values.

6.8.2 Objects

One of the basic ideas behind the object extensions in SQL3 is that, in addition to the normal built-in types defined by SQL, user-defined types may also be defined. These types may be used in the same way as built-in types. For example, columns in relational tables may be defined as taking values of user-defined types, as well as built-in types. A user-defined abstract data type (ADT) definition encapsulates attributes and operations in a single entity. In SQL3, an abstract data type (ADT) is defined by specifying a set of declarations of the stored attributes that represent the value of the ADT, the operations that define the equality and ordering relationships of the ADT, and the operations that define the behavior (and any virtual attributes) of the ADT. Operations are implemented by procedures called routines. ADTs can also be defined as subtypes of other ADTs. A subtype inherits the structure and behavior of its supertypes (multiple inheritance is supported). Instances of ADTs can be persistently stored in the database only by storing them in columns of tables.

A row type is a sequence of field name/data type pairs resembling a table definition. Two rows are type-equivalent if both have the same number of fields and every pair of fields in the same position have compatible types. The row type provides a data type that can represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines, and returned as return values from function invocations. This facility also allows columns in tables to contain row values. A named row type is a row type with a name assigned to it. A named row type is effectively a user-defined data type with a non-encapsulated internal structure (consisting of its fields). A named row type can be used to specify the types of rows in table definitions. A named row type can also be used to define a reference type. A value of the reference type defined for a specific row type is a unique value which identifies a specific instance of the row type within some (top level) database table. A reference type value can be stored in one table and used as a direct reference (pointer) to a specific row in another table, just as an object identifier in other object models allows one object to directly reference another object. The same reference type value can be stored in multiple rows, thus allowing the referenced row to be shared by those rows.

Tables have also been enhanced with a subtable facility. A table can be declared as a subtable of one or more supertables (it is then a direct subtable of these supertables), using an `UNDER` clause associated with the table definition. When a subtable is defined, the subtable inherits every column from its supertables, and may also define columns of its own. The subtable facility is completely independent from the ADT subtype facility.

6.8.3 Operations

Operations that may be invoked in SQL include defined operations on tables

`SELECT, INSERT, UPDATE, DELETE`

the implicitly defined functions defined for ADT attributes, and routines either explicitly associated with ADTs or defined separately.

Routines associated with ADTs are `FUNCTION` definitions for type-specific user-defined behavior. The `FUNCTION` definitions specify the operations on the ADT and return a single value of a defined data type. Functions may either be SQL functions, completely defined in an SQL schema definition, or external functions, defined in standard programming languages.

SQL functions associated with ADTs are invoked using either a functional notation or a dot notation (the dot notation is syntactic sugar for the functional notation). For example:

```
BEGIN
DECLARE r real_estate
...
SET r..area = 2540;           /* same as area(r,2540)
SET ... = r..area;            /* same as area(r)
...
SET ... = r..location..state; /* same as state(location(r))
SET r..location..city = 'LA'; /* same as city(location(r),'LA')
END;
```

Routines (procedures and functions) that define aspects of the behavior of the ADT may be encapsulated within the ADT definition (these routines have access to the ADT's `PRIVATE` attributes; routines may also be defined outside an ADT definition). A number of these routines have predefined names. For example, when an ADT is defined, a constructor function is automatically defined to create new instances of the type. The constructor function has the same name as the type and takes zero arguments. It returns a new instance of the type whose attributes are set to their default values. The constructor function is `PUBLIC`. For every attribute, observer and mutator functions are also automatically defined (these functions may also be explicitly defined by the user). These functions are used to read or modify the ADT attribute values.

`EQUAL` and `LESS THAN` functions may be defined to specify type-specific functions for comparing ADT instances. `RELATIVE` and `HASH` functions can be specified to control ordering of ADT instances. `CAST` functions can also be specified to provide user-specified conversion functions between different ADTs.

Other routines associated with ADTs include function definitions for type-specific user-defined behavior. ADT function definitions return either `BOOLEAN`, if the result is to be used as a truth value in a Boolean predicate, or a single value of a defined data type, if the result is to be used as a value specification. Functions may either be SQL functions, completely defined in an SQL schema definition, or external function calls to functions defined in standard programming languages.

6.8.4 Methods

An SQL routine is basically a subprogram. A routine may be either a `FUNCTION` or a `PROCEDURE`. A routine reads or updates components of an ADT instance or accesses any other parameter declared in its parameter list. A routine is specified by giving its name, its parameters, a `RETURNS` clause if it is a function, and a body. A parameter in the parameter list consists of a parameter name, its data type, and whether it is `IN`, `OUT`, or `INOUT` (for functions, the parameters are always `IN`; the `RETURNS` clause specifies the data type of the result returned).

A routine may be either an SQL routine or an external routine. An SQL routine has a body that is written completely in SQL. An external routine has an externally-provided body written in some standard programming language. If the function is an SQL routine, its body is any SQL statement, including compound statements and control statements. A number of new statement types have been added in SQL3 in order to make SQL computationally-complete enough so that ADT behavior can be completely specified in SQL.

SQL3 supports state in the form of the values of the various SQL3 data types. For example, the state of an ADT instance is the ordered sequence of stored components of an ADT instance; the state of a row is the ordered set of values of its columns; and so on. Values can only be stored persistently by storing them in the columns of database tables.

An ADT instance can exist in any location that an ADT name can be referenced. However, the only way that any ADT instance can be stored persistently in the database is to be stored as the column value of a table. For example, in order to store instances of an `employee_t` ADT persistently in a database, a table would have to be created with a column having the ADT as its data type, such as the `emp_data` column in

```
CREATE TABLE employees
( emp_data employee_t );
```

There is no facility in SQL3 to name individual instances of an ADT, and to store them persistently in the database using only that name. Similarly, there is no central place that all instances of a given ADT will exist (a built-in type extent), unless the user explicitly creates such a place, i.e., by defining a table in which all instances are stored. Thus, in SQL3 it is not necessarily possible to apply SQL query operations to all instances of a given ADT. The instances must first be stored in one or more tables (as column values).

A row in a table exists until it is deleted. Deletion of an ADT instance is done by deleting the row in which it is stored.

SQL3 routines may be defined within ADT definitions, or independently of them. SQL3 supports a generalized object model in terms of dispatching. However, there is no concept of a generic function which groups routines with a common signature. A routine defined within an ADT has access to that ADT's `PRIVATE` members.

6.8.5 Events

In SQL, a trigger is a named database construct that is implicitly activated whenever a triggering event occurs. When a trigger is activated, the specified action is executed if the specified condition is satisfied. An example is

```
CREATE TRIGGER update_balance
BEFORE INSERT ON account_history              /* event */
REFERENCING NEW AS ta
FOR EACH ROW
WHEN (ta.TA_type = 'W')                       /* condition */
UPDATE accounts                               /* action */
  SET balance = balance - ta.amount
  WHERE account_# = ta.account_#;
```

Triggers can be used for a number of purposes, such as validating input data, reading from other tables for cross-referencing purposes, or supporting alerts (e.g., through electronic mail messages). Triggering events include

insertion, deletion, update of tables and columns.

A condition can be any SQL condition (including those that involve complex queries), and an action can be any SQL statement (including compound statements, and those that invoke SQL routines). The trigger can also specify whether the trigger should be activated `BEFORE` the triggering SQL operation is performed, or `AFTER`. The condition and action can refer to both old and new values of rows affected by the SQL statement. The trigger condition and action can be executed

`FOR EACH ROW`

affected by the triggering statement, or only once for the whole triggering statement

`FOR EACH STATEMENT`

6.8.6 Binding and Polymorphism

Different routines may have the same name. This is referred to as overloading, and may be required, for example, to allow an ADT subtype to redefine an operation inherited from a supertype. SQL3 implements what is sometimes known as a generalized object model, meaning that the types of all arguments of a routine are taken into consideration when determining what routine to invoke, rather than using only a single type specified in the invocation as, for example, in C++ or Smalltalk. As a result, the rules for determining which routine to invoke for a given invocation can be fairly complex. The instance of the routine that is chosen for execution is the best match given the types of the actual arguments of the invocation at run time.

Each component (attribute or function) of an ADT has an encapsulation level of either

`PUBLIC, PRIVATE, PROTECTED`

`PUBLIC` components form the interface of the ADT and are visible to all authorized users of the ADT.

`PRIVATE` components are totally encapsulated, and are visible only within the definition of the ADT that contains them.

`PROTECTED` components are partially encapsulated; they are visible both within their own ADT and within the definition of all subtypes of the ADT.

SQL3 also supports encapsulation for tables to the extent that views (derived tables) are considered as providing encapsulation.

By default, testing corresponding attribute values for equality serves to test for the equality of two ADT instances. Alternatively, the specification of an ADT supports declaration of a function to be used to determine equality of two ADT instances.

Two values are said to be not distinct if either: both are the null value, or they compare equal according to [the SQL3] `"<comparison predicate>"`. Otherwise they are distinct. Two rows (or partial rows) are distinct if at least one of their pairs of respective values is distinct. Otherwise they are not distinct. The result of evaluating whether or not two values or two rows are distinct is never unknown.

6.8.7 Types and Classes

The parts of SQL3 that provide the primary basis for supporting object-oriented structures are extensions to its type facilities, specifically:

user-defined types (ADTs, named row types, and distinct types)
type constructors for row types and reference types
type constructors for collection types (sets, lists, and multisets)
user-defined functions and procedures
support for large objects (BLOBs and CLOBs)
SQL3 also supports a number of built-in scalar types.

One of the basic ideas behind the object facilities is that, in addition to the normal built-in types defined by SQL, user-defined types may also be defined. These types may be used in the same way as built-in types. For example, columns in relational tables may be defined as taking values of user-defined types, as well as built-in types.

The simplest form of user-defined type in SQL3 is the distinct type, which provides a facility for the user to declare that two otherwise equivalent type declarations are to be treated as separate data types. The keyword

```
DISTINCT
```

used in an declaration indicates that the resulting type is to be treated as distinct from any other declaration of the same type. For example, if two new types are declared as:

```
CREATE DISTINCT TYPE us_dollar AS DECIMAL(9,2)

CREATE DISTINCT TYPE canadian_dollar AS DECIMAL(9,2)
```

any attempt to treat an instance of one type as an instance of the other would result in an error, even though each type has the same representation.

A user-defined abstract data type (ADT) definition encapsulates attributes and operations in a single entity. In SQL3, an abstract data type (ADT) is defined by specifying a set of declarations of the stored attributes that represent the value of the ADT, the operations that define the equality and ordering relationships of the ADT, and the operations that define the behavior (and any virtual attributes) of the ADT. Operations are implemented by procedures called routines. ADTs can also be defined as subtypes of other ADTs. A subtype inherits the structure and behavior of its supertypes (multiple inheritance is supported). Instances of ADTs can be persistently stored in the database only by storing them in columns of tables.

An example ADT declaration is

```
CREATE TYPE employee_t
(PUBLIC
 name CHAR(20),
 b_address address_t,
 manager employee_t,
 hiredate DATE,
PRIVATE
 base_salary DECIMAL(7,2),
 commission DECIMAL(7,2),
PUBLIC
 FUNCTION working_years (p employee_t) RETURNS INTEGER
  <code to calculate number of working years>,
PUBLIC
 FUNCTION working_years (p employee_t,y years) RETURNS employee_t
  <code to update number of working years>,
PUBLIC
 FUNCTION salary (p,employee_t) RETURNS DECIMAL
  <code to calculate salary>
);
```

ADTs are completely encapsulated. Only attributes and functions defined as `PUBLIC` are accessible from outside the ADT definition. For each attribute (such as `name`), an observer and mutator function is automatically defined. Virtual attributes (such as `working_years`) can also be defined. These do not have stored values; their behavior is provided by user-defined observer and mutator functions that read and define their values (`salary` is a read-only virtual attribute). ADT instances are created by system-defined constructor functions. The instances created in this way have their attributes initialized with their default values, and can be further initialized by the user by invoking mutator functions, as in:

```
  BEGIN
  DECLARE e employee_t;
  SET e..working_years = 10;
  SET y = e..working_years;
  SET z = e..salary;
END;
```

The expression `e..working_years` illustrates the dot notation used to invoke the `working_years` function of the ADT instance denoted by `e`. Users can also define specialized constructor functions which take parameters to initialize attributes.

A row type is a sequence of field name/data type pairs resembling a table definition. Two rows are type-equivalent if both have the same number of fields and every pair

of fields in the same position have compatible types. The row type provides a data type that can represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines, and returned as return values from function invocations. This facility also allows columns in tables to contain row values. An example is:

```
 CREATE TABLE employees
 (name CHAR(40),
  address ROW(street CHAR(30),
              city CHAR(20),
              zip ROW(original CHAR(5),
                      plus4 CHAR(4))));
INSERT INTO employees
VALUES('John Wu',('2225 Coral Drive','San Jose',('95124','2347'))));
```

A named row type is a row type with a name assigned to it. A named row type is effectively a user-defined data type with a non-encapsulated internal structure (consisting of its fields). A named row type can be used to specify the types of rows in table definitions. For example:

```
CREATE ROW TYPE account_t
(acctno INT,
 cust REF(customer_t),
 type CHAR(1),
 opened DATE,
 rate DOUBLE PRECISION,
 balance DOUBLE PRECISION,
);

CREATE TABLE account OF account_t
(PRIMARY KEY acctno
);
```

A named row type can also be used to define a reference type. A value of the reference type defined for a specific row type is a unique value which identifies a specific instance of the row type within some base (top level) database table. A reference type value can be stored in one table and used as a direct reference (pointer) to a specific row in another table, just as an object identifier in other object models allows one object to directly reference another object. The same reference type value can be stored in multiple rows, thus allowing the referenced row to be shared by those rows. For example, the `account_t` row type defined above contains a `cust` column with the reference type `REF(customer_t)`. A value of this column identifies a specific row of type `customer_t`. The value of a reference type is unique within the database, never changes as long as the corresponding row exists in the database,

and is never reused.

In general, the value of a reference type such as `REF(customer_t)` can refer to a row in any table having rows of type `customer_t`. If a `SCOPE` clause is specified in the definition of a table, such references are restricted to rows in a single table, as in:

```
CREATE TABLE account OF account_t
(PRIMARY KEY acctno,
 SCOPE FOR cust IS customer
);
```

In this case `customer_t` rows referenced in the `cust` column must be stored in the `customer` table. Use of `SCOPE` does not imply any referential integrity constraint.

References can be used in path expressions (similar to those used in some other object query languages), that permit traversal of object references to navigate from one row to another. Such expressions can also include the invocation of functions on ADT instances. An example is:

```
SELECT a.cust -> name
FROM account a
WHERE a.cust -> address..city = Hollywood
AND a.balance > 1000000;
```

In the `SELECT` statement, `a.cust -> name` represents:

1. the selection of the `cust` column's value (an instance of type `REF(customer_t)`) from the row denoted by `a` (a row of type `account_t`)

2. the traversal (dereference) of that instance of type `REF(customer_t)` to the row of type `customer_t` it refers to (`->` is a dereferencing operator)

3. the selection of the name column from the referenced `customer_t` row.

In the `WHERE` clause, `a.cust -> address..city` represents a similar process, identifying the `address` column of the referenced `customer_t` row, and then applying the `city` observer function to the ADT instance found in the `address` column.

Collection types for sets, lists, and multisets have also been defined. Using these types, columns of tables can contain sets, lists, or multisets, in addition to individual values.

For example

```
 CREATE TABLE employees
(id INTEGER PRIMARY KEY,
 name VARCHAR(30),
 address ROW(street VARCHAR(40),
             city CHAR(20),
             start CHAR(2),
             zip INTEGER),
 projects SET(INTEGER),
 children LIST(person),
 hobbies SET(VARCHAR(20))
);
```

The `BLOB` (Binary Large Object) and `CLOB` (Character Large Object) types have been defined to support very large objects. Instances of these types are stored directly in the database (rather than being maintained in external files). For example:

```
CREATE TABLE employees
(id INTEGER,
 name VARCHAR(30),
 salary us_dollar,
...
 resume CLOB(75K),
 signature BLOB(1M),
 picture BLOB(12M));
```

LOB types are excluded from some operations, such as greater and less than operators, but are supported by other operations, such as value retrieval, and the `LIKE` predicate.

6.8.8 Inheritance and Delegation

An ADT can be defined as a subtype of one or more ADTs by defining it as `UNDER` those ADTs (multiple inheritance is supported). In this case, the ADT is referred to as a direct subtype of the ADTs specified in the

`UNDER` clause,

and these ADTs are direct supertypes. A type can have more than one subtype and more than one supertype. A subtype inherits all the attributes and behavior of its supertypes; additional attributes and behavior can also be defined. An instance of a subtype is considered an instance of all of its supertypes. An instance of a subtype can be used wherever an instance of any of its supertypes is expected.

Every instance is associated with a most specific type that corresponds to the lowest subtype assigned to the instance. At any given time, an instance must have exactly one most specific type (in some cases, multiple inheritance must be used to ensure this is true). The most specific type of an instance need not be a leaf type in the type hierarchy. For example, a type hierarchy might consist of a maximal supertype `person`, with `student` and `employee` as subtypes. `student` might have two direct subtypes `undergrad` and `grad`. An instance may be created with a most specific type of `student`, even though it is not a leaf type in the hierarchy. A `TYPE` predicate allows for the type of an ADT instance to be tested at run time.

A subtype definition has access to the representation of all of its direct supertypes (but only within the ADT definition that defines the subtype of that supertype), but it has no access to the representation of its sibling types. Effectively, components of all direct supertype representations are copied to the subtype's representation with the same name and data type. To avoid name clashes, a subtype can rename selected components of the representation inherited from its direct supertypes.

A subtype can define operations like any other ADT. A subtype can also define operations which have the same name as operations defined for other types, including its supertypes (overriding).

A table can be declared as a subtable of one or more supertables (it is then a direct subtable of these supertables), using an `UNDER` clause associated with the table definition. An example is:

```
CREATE TABLE person
(name CHAR(20),
 sex CHAR(1),
 age INTEGER);

CREATE TABLE employee UNDER person
(salary FLOAT);
```

```
CREATE TABLE customer UNDER person
(account INTEGER);
```

The subtable facility is completely independent from the ADT subtype facility. When a subtable is defined, the subtable inherits every column from its supertables, and may also define columns of its own. A maximal supertable (a supertable that is not a subtable of any other table) together with all its subtables (direct and indirect) makes up a subtable family. A subtable family must always have exactly one maximal supertable. Any row of a subtable must correspond to exactly one row of each direct supertable. Any row of a supertable corresponds to at most one row of a direct subtable.

The rules for the SQL `INSERT`, `DELETE`, and `UPDATE` DML statements are defined in such a way as to keep the rows in the tables of a subtable family consistent with each other, in accordance with the rules described above.

Specifically:

If a row is inserted into a subtable T, then a corresponding row (with the same row identifier, and the same values as any values provided for inherited columns of T) is inserted into each supertable of T, cascading upward in the table hierarchy. If T is a maximal supertable, a row is inserted only into T.

If a row is updated in a supertable, then all inherited columns in all corresponding rows of the direct and indirect subtables are correspondingly changed.

If a row is updated in a subtable, then every corresponding row is changed so that their column values match the newly updated values.

If a row in a table that belongs to a subtable family is deleted, then every corresponding row is also deleted.

The semantics maintained are those of containment; a row in a subtable is effectively contained in its supertables. This means that, for example, a row could exist for a person in the `person` table without a corresponding row in the `employee` table (if the person is not also an employee). A row for a new employee, not corresponding to any existing person, could be inserted into the `employee` table, and this would automatically create a corresponding row in the `person` table.

6.8.9 Noteworthy Objects

Relations (tables) can be used to define generalized n-ary relationships, as in SQL92; referential and other integrity constraints can be defined on these tables. Columns whose types are reference types also allow modeling of relationships in SQL3. References to groups of objects can be specified using rows containing (directly or indirectly) instances of the SQL3

```
MULTISET(..),  LIST(..),  SET(..)
```

collection types.

There are two types of ADT attributes, stored attributes and virtual attributes. A stored attribute is specified by giving an attribute name and a data type. The data type of a stored attribute can be any known data type, including another ADT. Each stored attribute implicitly declares a pair of functions to get (observer function) and set (mutator function) the attribute value.

A virtual attribute has a value that is derived or computed by a user-defined observer function. Because ADTs are encapsulated, and because the syntax for function invocation is the same for any attribute, only the type owner and subtype definers would ever be aware of this distinction. Columns of tables can also be used to represent attributes, as in SQL92.

In SQL3, literals are used to specify non-null values. The rules for forming literals for the various built-in types are contained in the draft standard [ISO96a]. ADTs do not have literal values. Row type literals are formed by concatenating values for the individual columns, as in:

```
   CREATE TABLE employees
   (name CHAR(40),
    address ROW(street CHAR(30),
                city CHAR(20),
                zip ROW(original CHAR(5),
                        plus4 CHAR(4))));
INSERT INTO employees
VALUES('John Doe',('2225 Coral Drive','San Jose',('95124','2347'))));
```

SQL3 supports the concept of values being contained within values (e.g., instances of row types, or collections of such instances, can be contained in a column of a row) or within ADTs. A form of containment semantics can also be implemented by specifying triggers to enforce cascaded manipulations of a collection of data structures when one of them is manipulated. This kind of containment must be specified by the user.

SQL3 provides row types as literal structures. Instances of row types can be used as values in tables; row types can also be nested. A number of predefined parameterized collection types are also defined. A collection may be specified as

`SET(<type>), MULTISET(<type>), LIST(<type>)` .

In each case, the `<type>` parameter (called the element type) can be a predefined type, an ADT, a row type, or another collection type. For example `SET(INTEGER)` and `SET(LIST(INTEGER))` would both be valid declarations, as would `SET(movie)` and `SET(LIST(movie))`, where `movie` is some previously defined ADT. At present, the element type cannot be a reference type, nor can it be a named row type containing a field whose type is a reference type.

A collection can be used as a simple table in queries. In this case, each element of the collection corresponds to a row in the table. The table is treated as having a single column whose type is defined by the type of the instances of the collection. Since collection types are data types, they must be declared as the types of table columns in order to store instances of collections persistently in the database.

6.8.10 Extensibility

New tables and types (ADTs, row types, collection types, etc.) can be defined based on existing types. Existing types may be modified to add new operations, attributes, or constraints. Existing instances may not acquire or lose type without creating a new instance and destroying the old one.

Limited schema evolution is possible by applying the `ALTER` statement to a base table. Actions that can be taken using the `ALTER` statement include adding, altering, and dropping columns, and adding and dropping supertables, and table constraints. Data types can also be added and dropped.

SQL3 has no notion of metaclass and its semantics are not extensible.

Some metadata is maintained in SQL3 tables (this is a requirement of SQL92) and can be read by the user.

A number of new statement types have been added in SQL3 in order to make SQL computationally-complete enough so that object behavior can be completely specified in SQL. Some of the additional statements provided for writing SQL functions include:

An assignment statement that allows the result of an SQL value expression to be assigned to a free standing local variable, a column, or an attribute of an ADT.

A `CALL` statement that allows invocation of an SQL procedure.

A `RETURNS` statement that allows the result of an SQL value expression to be returned as the `RETURNS` value of the SQL function.

A `CASE` statement to allow selection of an execution path based on alternative choices.

An `IF` statement with `THEN`, `ELSE`, and `ELSEIF` alternatives to allow selection of an execution path based on the truth value of one or more conditions.

Statements for `LOOP`, `WHILE`, and `REPEAT` to allow repeated execution of a block of SQL statements. `WHILE` checks a search condition; before execution of the block, and `REPEAT` checks it afterwards. All three statements are allowed to have a statement label.

Additional control facilities available include compound statements and exception handling. A compound statement is a statement that allows a collection of SQL statements to be grouped together into a block. A compound statement may declare its own local variables and specify exception handling for an exception that occurs during execution of any statement in the group. For exception handling, a `CONDITION` declaration establishes a one-to-one correspondence between an SQLSTATE error condition and a user-defined exception name. `HANDLER` declarations associate user-defined exception handlers with specific exceptions.

The SQL92 standard defines language bindings for a number of standard languages. A key aspect of the individual language bindings is the definitions of correspondences between SQL data types and host language data types. In some cases, these are relatively straightforward; e.g., the SQL `CHARACTER` data type maps to a C `char`. In other cases, the mapping is not so straightforward. For example, SQL92 has a `TIMESTAMP` data type, but standard programming languages do not contain a corresponding built-in type. In these cases, SQL requires the use of a `CAST` function to convert database `TIMESTAMP` data to character data in the program, and vice-versa. In SQL92, these type correspondences are defined only at the level of elementary scalar data types. There are no type correspondences defined for structured types, e.g., between a row of an SQL table and a flat record or structure in a programming language (although some such correspondences would be relatively straightforward to define).

There are currently no bindings defined between the SQL3 ADT extensions (or rows containing them) and object classes or types in object-oriented programming languages such as C++, Java or Smalltalk, although these are under investigation.

6.9 SQL3 Datatypes and Java

The datatypes commonly referred to as SQL3 types are the new datatypes being adopted in the next version of the ANSI/ISO SQL standard. The JDBC 2.0 API provides interfaces that represent the mapping of these SQL3 datatypes into the Java programming language. With these new interfaces, we can work with SQL3 datatypes the same way we do other datatypes.

The new SQL3 datatypes give a relational database more flexibility in what can be used as a type for a table column. For example, a column may now be used to store the new type `BLOB` (Binary Large Object), which can store very large amounts of data as raw bytes. A column may also be of type `CLOB` (Character Large Object), which is capable of storing very large amounts of data in character format. The new type `ARRAY` makes it possible to use an array as a column value. Even the new SQL user-defined types (UDTs), structured types and distinct types, can now be stored as column values.

The following list gives the JDBC 2.0 interfaces that map the SQL3 types. We will discuss them in more detail later.

1.) A `Blob` instance maps an SQL `BLOB` instance

2.) A `Clob` instance maps an SQL `CLOB` instance

3.) A `Array` instance maps an SQL `ARRAY` instance

4.) A `Struct` instance maps an SQL structured type instance

5.) A `Ref` instance maps an SQL `REF` instance

Using SQL3 Datatypes

We retrieve, store, and update SQL3 datatypes the same way we do other datatypes. We use either

```
ResultSet.getXXX
```

or

```
CallableStatement.getXXX
```

methods to retrieve them,

```
PreparedStatement.setXXX
```

methods to store them, and

```
updateXXX
```

to update them. Most of the operations performed on SQL3 types involve using the `getXXX`, `setXXX`, and `updateXXX` methods. The following table shows which methods to use:

SQL3 type	getXXX method	setXXX method	updateXXX method
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
ARRAY	getArray	setArray	updateArray
Structured type	getObject	setObject	updateObject
REF (structured type)	getRef	setRef	updateRef

For example, the following code fragment retrieves an SQL `ARRAY` value. For this example, the column `SCORES` in the table `STUDENTS` contains values of type `ARRAY`. The variable `stmt` is a `Statement` object.

```
ResultSet rs = stmt.executeQuery(
"SELECT SCORES FROM STUDENTS WHERE ID = 2238");
rs.next();
Array scores = rs.getArray("SCORES");
```

The variable `scores` is a logical pointer to the SQL `ARRAY` object stored in the table `STUDENTS` in the row for student 2238. If we want to store a value in the database, we use the appropriate `setXXX` method. For example, the following code fragment, in which `rs` is a `ResultSet` object, stores a `Clob` object:

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE MARKETS SET COMMENTS = ? WHERE SALES < 1000000",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
pstmt.setClob(1,notes);
```

This code sets `notes` as the first parameter in the update statement being sent to the database. The `CLOB` value designated by `notes` will be stored in the table `MARKETS` in column `COMMENTS` in every row where the value in the column `SALES` is less than one million.

Blob, Clob, and Array Objects

An important feature about `Blob`, `Clob`, and `Array` objects is that we can manipulate them without having to bring all of the data from the database server to our client machine. An instance of any of these types is actually a logical pointer to the object in the database that the instance represents. Because an SQL `BLOB`, `CLOB`, or `ARRAY` object may be very large, this feature can improve performance dramatically.

We can use SQL commands and the JDBC 1.0 and 2.0 API with `Blob`, `Clob`, and `Array` objects just as if we were operating on the actual object in the database. If we want to work with any of them as an object in the Java programming language, however, we need to bring all their data over to the client, which we refer to as materializing the object. For example, if we want to use an SQL `ARRAY` object in an application as if it were an array in the Java programming language, we need to materialize the `ARRAY` object on the client and then convert it to an array in the Java programming language. Then we can use array methods in the Java programming language to operate on the elements of the array. The interfaces `Blob`, `Clob`, and `Array` all have methods for materializing the objects they represent. Refer to the second edition of JDBC Database Access with Java if we want more details or examples.

Struct and Distinct Types

SQL structured types and distinct types are the two datatypes that a user can define in SQL. They are often referred to as UDTs (user-defined types), and we create them with an SQL `CREATE TYPE` statement.

An SQL structured type is similar to structured types in the Java programming language in that it has members, called attributes, that may be of any datatype. In fact, an attribute may itself be another structured type. Here is an example of a simple definition creating a new SQL datatype:

```
CREATE TYPE PLANE_POINT
(
X FLOAT,
Y FLOAT
)
```

Unlike `Blob`, `Clob`, and `Array` objects, a `Struct` object contains values for each of the attributes in the SQL structured type and is not just a logical pointer to the object in the database. For example, suppose that a `PLANE_POINT` object is stored in column `POINTS` of table `PRICES`.

```
ResultSet rs = stmt.executeQuery(
    "SELECT POINTS FROM PRICES WHERE PRICE > 3000.00");
while (rs.next()) {
Struct point = (Struct)rs.getObject("POINTS");
// do something with point
}
```

If the `PLANE_POINT` object retrieved has an X value of 3 and a Y value of -5, the `Struct` object `point` will contain the values 3 and -5.

We note that `Struct` is the only type not to have a `getXXX` and `setXXX` method with its name as `XXX`. We must use `getObject` and `setObject` with `Struct` instances. This means that when we retrieve a value using the method `getObject`, we will get an `Object` in the Java programming language that we must explicitly cast to a `Struct`, as was done in the previous code example.

The second SQL type that a user can define in an SQL `CREATE TYPE` statement is a distinct type. An SQL distinct type is similar to a `typedef` in C or C++ in that it is a new type based on an existing type.

Here is an example of creating a distinct type

```
CREATE TYPE MONEY AS NUMERIC(10,2)
```

This definition creates the new type called `MONEY`, which is a number of type `NUMERIC` that is always base 10 with two digits after the decimal point. `MONEY` is now a datatype in the schema in which it was defined, and we can store instances of `MONEY` in a table that has a column of type `MONEY`. An SQL distinct type is mapped to the type in the Java programming language to which its underlying type would be mapped. For example, `NUMERIC` maps to `java.math.BigDecimal`, so the type `MONEY` maps to `java.math.BigDecimal`. To retrieve a `MONEY` object, we use

```
ResultSet.getBigDecimal
```

or

```
CallableStatement.getBigDecimal
```

to store a `MONEY` object, we use `PreparedStatement.setBigDecimal`.

Some aspects of working with SQL3 types can get quite complex. The interface `Struct` is the standard mapping for an SQL structured type. If we want to make working with an SQL structured type easier, we can map it to a class in the Java programming language. The structured type becomes a class, and its attributes become fields. We do not have to use a custom mapping, but it can often be more convenient. Sometimes we may want to work with a logical pointer to an SQL structured type rather than with all the values contained in the structured type. This might be true, for instance, if the structured type has many attributes or if the attributes are themselves large. To reference a structured type, we can declare an SQL `REF` type that represents a particular structured type. An SQL `REF` object is mapped to a `Ref` object in the Java programming language, and we can operate on it as if we were operating on the structured type object that it represents.

6.10 Evaluations of OODBMSs

The following are some of the technical issues that must be included in OODBMS evaluations.

1. Pointer traversal
2. Application-DBMS coupling
3. Complex object support
4. Updates
5. Recovery
6. Path indexing
7. Long data items
8. Cluserting
9. Queries and optimization
10. Caching
11. Concurrency control
12. Relationhips
13. Versioning

A more comprehensive list of evaluation criteria is

1. Atomicity of transactions
2. Consistency in transactions
3. Encapsulation and isolation of internal state of transactions
4. Durability of transactions
5. Long transactions
6. Nested transactions
7. Shared transactions
8. Nonblocking read consistency
9. Concurrency control issues
10. Lock escaletion
11. Promotable locks
12. Locks set and release
13. Granularity of locks
14. Deadlock detection
15. Role assignment for security
16. Role authorization
17. Implicit role authorization
18. Positive and negative authorization
19. Strong and weak authorization
20. Authorization objects
21. Authorization by time and day
22. Multilevel security
23. Query language
24. Query optimization

25. Indexing
26. Queries in programs
27. Query language completeness
28. Specification of collections
29. SQL as a query language
30. Support for views
31. Schema modification
32. Update time
33. Changes of attributes
34. Addition of attributes from a class
35. Deleting attributes from a class
36. Changing the names of an attribute
37. Changing static members of the class
38. Adding new methods to a class
39. Dropping an existing class method
40. Changing the name of a method
41. Changing the inheritance of a method
42. Adding and removing a superclass to an existing class
43. Changes to an existing class
44. Versioning schema
45. Location independence in distributed database system
46. Local autonomy at each site
47. Fragmentation and replication independence
48. Distributed query processing
49. Distributed transaction processing
50. Hardware independence
51. Operating system and network independence
52. Object model
53. Dynamic binding and polymorphism
54. Encapsulation
55. Object identity
56. Types and classes
57. Inheritance and delegation
58. Relationship and attributes
59. Literal data types
60. Multimedia objects
61. Aggregate objects
62. Composite and complex objects
63. Integrity
64. Schema extensibility
65. Extensibility of database operations
66. Language for object manipulations
67. Persistence
68. Architecture
69. Method execution location

70. Multiple language interface
71. Object translation facilities
72. Backup and recovery facilities
73. Tools
74. 4GL (four generation language)
75. Change notification
76. Versioning
77. Configuration management
78. Standards
79. Rules
80. Platforms
81. Compilers
82. GUIs
83. Gateways
84. File systems

6.11 Summary

It is evident that object-oriented databases are a natural outcome of database evolutionary developments. Database systems are advancing from being just complex file systems to architectures that provide organized mechanisms to make sense and capture the meaning of the data. Essentially all object data management systems are developed using an object-oriented database programming language. These database programming languages, in turn, have been integrated with the Java, C++, Smalltalk or other programming languages that provide support for complex data structures that can transparently be made persistent. We have examples of four levels of extensiveness and, therefore, sophistication in object data management systems. The root system is an object manager. An object manager must provide a repository for persistent objects – and generally provide multi-user concurrence control – but lack a query or programming language. Extended database managers, the next level of sophistication, incorporates a query language. Database programming languages incorporate persistence, concurrence control and a query or programming language that executes in the application program environment, sharing the same type system and data workspace. The most lofty level of extensiveness is a database system generator. These systems permit a designer to provide specifications for a particular application for which any of the other three object data management systems can be routinely spawned.

Next generation DBMS applications will require a new set of services in different areas:

1) There will be new kinds of data, probably large and internally complex. A new type of query language might be required for large data.

2) Type management is going to be important.

3) Rules are likely to be common. They can be declarative or imperative. Rules may include elaborate constraints that the designer wants enforced.

4) They will require new concepts not found in most applications - spatial data, time, uncertainty.

5) Scaling up to the next order of magnitude in terms of size. Building an index dynamically on terabytes of information is not practical. Making a dump of such a large database might not be feasible even in case of hardware failure.

6) Ad hoc queries might take a long time to execute. Parallelism might be a solution. Users' queries would be probably need to be executed at nearly linear speedup.

7) Tertiary storage and long-duration transactions may be necessary. Part of the data being processed may still be in archives. Thus, query optimization will be

critical, and movement of data between storage media will have to be minimized.

8) Support for multiple versions and configurations will be necessary.

9) Databases are likely to operate in heterogenous and distributed environments. It may be possible to create a single worldwide database from which users can obtain information on any topics covered by data, made available by purveyors, and on which business can be transacted in a uniform way.

10) Uniform browsing capability that can be applied to any one of individual databases. The ability to interrogate the structure of the database is imperative if browsing is to be supported across several databases.

11) In a multidatabase system, the definition of data may not be consistent across all databases, leading to answers that may not be semantically consistent. Future interoperability of databases will require dramatic progress to be made on semantic issues.

12) Mediators, a class of information sources that stand between the user and the heterogenous databases, will be needed to address the problems of semantic inconsistencies and database merging.

13) Name services may need to be globally accessible, with mechanisms by which items enter and leave such name servers.

14) Security, especially in a distributed and heterogenous environment, might be a big problem. Good authentication services will be necessary for reliable identification of subjects making database access.

15) Site scale-up issues must be addressed, and this might involve design of better algorithms for query processing, concurrency control, and replication.

16) Transactions in a large distributed, heterogenous environment are a difficult issue, and will have to be supported. If each local database employs a different concurrency mechanism, integrating such systems so as to provide uniform transaction semantics will be a problem.

Chapter 7

Versant

7.1 Introduction

A Versant database, similar to a relational database, is just a collection of related data managed by a database server and administered by a DBA. Obviously, on a detailed level, there are many differences. Versant employs a conventional client/server architecture. Client applications link to and communicate with a Versant manager on the client platform through language-specific interfaces. The Versant manager communicates with the Versant database server through TCP/IP and routes the client's queries to the server. The server executes the queries and provides the requested objects to the clients via the Versant manager. The Versant manager consists of a few software modules that perform object caching, object validation, transaction and session management, schema management, and checkouts and checkins.

The Versant server performs query execution, object retrieval and updating, page caching, indexing, logging, and locking. Most RDBMSs implement a server-centric architecture in which all the database processing functions are executed by the database server process. Versant employs a "balanced" client/server implementation. Some database management functions such as transaction control are performed on the client, while other functions such as query execution and indexing are performed by the server. Certain functions, such as dual caching, are performed at both ends in an attempt to reduce network traffic.

Each Versant installation has a software root directory under which the Versant server software executables and database management utilities are stored and a database root directory under which all the databases are created. Each database is created in a database directory under the database root directory.

A Versant database consists of a number of storage volumes, which are files or raw disk devices. Each database consists of one system volume, one logical log volume, one physical log volume, and zero or more additional data volumes. The system volume is typically located in the database root directory. It is used to store class descriptions (metadata) and object instances (the data itself). The log volumes are used for transaction and longer-term rollback and recovery. They are usually located in the database root directory. The additional data volumes can be used to increase the database's capacity.

When we create a new Versant database, we have to make it, edit its profile, physically create it, and then (optionally) expand or move it. Making a database entails creating a database directory, assigning the necessary access privileges, and creating the database profiles. The Versant `makedb` utility can make a default database. If we want to change any of the database's physical characteristics, we have to edit its profile, which is stored in a file within the database root directory. There are lists and lists of parameters defined in the database profile, such as volume sizes and file locations, application process parameters, and server process parameters. The physical database is created by running the Versant `createdb` utility, which creates the system and log volumes, and updates the network database identifier file to indicate the presence of a new database. The command `DBLIST` provides a list of Versant databases.

The user that created a Versant database is considered the owner and DBA of that database. Only the DBA can perform certain administrative tasks, such as adding and removing users, and only the DBA may remove a database. There can be many different DBAs working on different databases in a Versant installation.

The directory structures Versant uses are not unlike those used by many relational DBMSs, but the creation of a database has an extra step to first "make" the database.

A Versant database models data as objects. A database object that holds data is called an instance object. Each object has a class, which defines its structure. A Versant database can have any number of classes, and any number of objects per class. When we create a class, Versant creates a corresponding class object in the database. The attributes of an object can be used to store any datatype, such as numeric and character values, images, sounds, documents, and references to other objects, as defined in the class. An object can contain other objects with their own attributes, or even arrays of other objects. The object attributes are generally defined in C++ class files or Java class files, which are then compiled and loaded in the database, where they are stored as class objects. An object also has methods, which are executable units of program code associated with the object. The methods are usually defined in C++ class files or Java class files, which are then compiled and linked with the applications. Each object has a unique logical object identifier, made up of a database identifier and an object identifier. A logical object

identifier always points to the same object, and it can never change its value.

Versant keeps track of a number of states for each object. An object can be transient or persistent, where only persistent objects are saved to the database and can be locked to guarantee access to it. It can have a dirty state, which means it has been changed, but not yet saved. An object can be checked out, which means that a user has set a long-term lock to work on the object for longer than the duration of a transaction.

We can convert an instance object to a versioned object. This instructs Versant to keep track of the changing states of the object by keeping a graph of the versions of the object. When updating a versioned object, the changes are automatically saved as a new version and linked to the version of the object to which the changes were made.

We can use the Versant Multimedia Access (VMA) module to load multimedia files into a Versant database as text, audio, image, video, URL, or application-specific C++ classes and Java classes. VMA includes a run-time version of the Verity SEARCH '97 engine, which automatically creates and maintains indexes for each managed database. We can extend the predefined classes to create custom classes, and we can create custom interfaces using the Verity Software Development Kit.

Object modeling can mean trouble to a relational DBA. We have to change our way of thinking about data modeling. It is better to design the database in terms of objects, classes, and inheritance than to find analogies for the known relational concepts. For example, it is difficult to determine the equivalent of a referential integrity constraint. The relationships between Versant objects are not trivial. An object can contain other objects, it can contain structures, such as arrays of other objects, or it can contain "is a" (inheritance) or "has a" (association) references to other objects. Object versions and object states are also two powerful and useful concepts to get used to.

Versant databases are used mostly with object-oriented programming languages that implement classes or templates and support delegation or inheritance. Versant's language-specific interfaces map the capabilities of the programming language to the object database model. Each language interface consists of several libraries and precompiled routines.

Versant has language-specific interfaces for C++, Smalltalk, and Java, but it can also be used with standard nonobject-oriented C. The C++/Versant interface implements the full object model. The C/Versant interface does not support messaging, so the methods are executed as functions, and we have to use the Versant View utility to view and edit class definitions. The Smalltalk/Versant programming interface consists of a set of Smalltalk classes that maps the Smalltalk objects directly to the Versant database objects. For example, when we retrieve an object from a Versant

database, it is placed in the Smalltalk Image as a standard Smalltalk object. The Java/Versant interface gives us access to the data storage primitives of the Versant database, including schema definition and access; object creation, deletion, and update; value-based and navigational query; full transaction support for distributed databases; and object sharing with other languages. A DBA will be able to extend the functionality of the Versant Server with user-written methods. Versant also has a pure Java interface, which allows pure Java applications to access all of the features of Versant's object database via standard or user-defined transport protocols.

A Versant query starts with a group of so-called "starting point objects," which may be a class of objects, a class and its subclasses, or an array of objects. After evaluating the starting point objects, the query can examine the attribute values in embedded and linked objects. There are basically two types of queries: dereference queries, which load objects from their links into memory, and search queries, which find objects and return their links. We can specify predicates in the search queries to locate specific objects. In the C++/Versant, Smalltalk/Versant and Java/Versant interfaces, there are quite a number functions that we can call to perform dereference and search queries. Some of the C++/Versant functions enable us to process the object instances through a cursor.

Versant (the company) is a founding member of ODMG, and Versant (the ODBMS) supports a subset of the ODMG 2.0 Object Query Language (OQL) released in mid-1997; it also supports OQL bindings to C++ and Java.

Versant also has its own proprietary high-level Versant Query Language (VQL). Queries expressed in a SQL-like syntax are assigned to strings, which are parsed and executed using C++/Versant or C/Versant routines. VQL contains a number of extensions to SQL that cater to object-oriented concepts. An example is the `ONLY` keyword in the `FROM` clause to restrict the query to the class named in the clause. Without the `ONLY` keyword, the class's subclasses will also be inspected.

Queries in Versant, like objects and classes, are a lot different from those in a relational DBMS. However, although the syntax is different, cursor processing in Versant is similar to its counterpart widely used in relational database applications. Cursors are very useful for accessing a large volume of objects without excessive locking and long wait times.

Versant/SQL Suite is an amazing knapsack of tools for relational DBAs (and users) to tackle Versant databases. Versant/SQL provides a SQL-92 standard interface to Versant databases. Versant/ISQL is an interactive tool that lets we fire SQL-92 queries interactively to a Versant database. Versant/ODBC is a client-side DLL that accepts ODBC 2.0 calls and translates them to Versant/SQL this allows any ODBC-compliant tool to access a Versant database. The Versant/SQL Mapper maps the object model stored in a Versant database to a relational data model, which can be presented to users and can be accessed using standard SQL. It can perform complex

mappings, such as transforming a many-to-many relationship between two objects to an intersection table. Similarly, it converts a multivalued attribute to a table with a many-to-one reference to the table representing the main object.

A client can access multiple Versant databases concurrently, and the Versant server can obviously serve multiple clients concurrently. The operations on a database are grouped in transactions. An application has to start a session to access any database. When an application starts a session, Versant creates various areas in memory, such as an object cache, an object cache table, and a server page cache for each database accessed in the session. A session starts up against a default database, but the application can connect to other databases as well. The session also opens up a connection to a session database, where it logs all the transaction details. There are various types of sessions for an application, such as standard, shared, nestable transaction, high-performance, optimistic, and custom locking.

Versant supports short and long transactions in the context of a session. Short transactions are logical units of work under the application's control. An application can explicitly commit or roll back a short transaction, or set a save point for the transaction. Whenever a session starts or a short transaction is ended, a new short transaction is automatically started. However, we must have logging and locking enabled for the database. If a transaction spans more than one database, a two-phase commit is automatically invoked at commit time. Long transactions start and end by default when the sessions start and end. However, we can join to a previous long transaction - that is, make the long transaction continue in a subsequent session - by specifying the long transaction's name at session startup time. We can use long transactions to keep track of different sets of checked out objects. We switch between multiple long transactions by stopping and restarting sessions.

Short locks are set at the object level for concurrency control. There are various types of short locks, such as write locks, update locks, read locks (which can be shared), and so-called null locks (used for dirty reads). Applications waiting for locks can be set to time out. Single database deadlocks are immediately detected and disallowed by Versant. It uses a timeout mechanism to detect deadlocks between multiple databases.

Versant has a very extensive set of concurrency control mechanisms from types of locks to check-in and check-out operations in the context of multiple long transactions much more extensive than any relational DBMS currently offers. This makes it possible to tune the database server's locking behavior very closely to any application's requirements. However, DBAs must know their drill before fooling around with these lethal weapons. We really have to know our concurrency control theory and have our lock behavior models ready when starting to tune the Versant locking system.

An object DBMS should also perform its tasks as fast as possible. Although performance is evaluated based on the application requirements, most users have expeditious performance requirements. Versant has a number of mechanisms for tuning its performance. Versant uses indexes to filter objects so that query routines only fetch the objects of interest from disk. Create and update routines also use indexes to enforce uniqueness constraints on object attributes. An index is defined on a single attribute of a class and indexes the specified attribute of all the objects of that class. We can have B-tree or hash indexes, where B-tree indexes are good for range queries and sufficient for direct key queries, and hash indexes are good for direct key queries. However, there are many restrictions and peculiarities about indexes that make them very different from their relational counterparts. Indexes do not have names. An index only applies to a single attribute. An attribute can only have two indexes - one B-tree and one hash. An index only applies to a specific class and it is not inherited, so we have to define corresponding indexes for all the subclasses where it is required. Some complex attributes, such as fixed arrays and fixed length structures, cannot be indexed.

Versant has many memory-management facilities that application developers and DBAs can use to improve the applications' performance. For example, its various caches can be configured, and objects can be pinned in their respective caches. Similarly, log files can be configured, and objects can be located on multiple disks, even on raw disk devices, to improve performance. This compares well with the facilities provided by the more tunable relational DBMSs. A DBA must be skilled in the Versant-specific techniques to be able to use them advantageously. Obviously, database and application design, as in a relational database application system, play the biggest roles in achieving good database and application performance.

Versant provides two different mechanisms to cater for disasters. The DBA can make online or online incremental backups of the database. An online backup can only be used to restore a crashed database to a specific previous correct state, when the backup was taken. An incremental backup has an associated rollforward log, which keeps track of all database activity that occurs between backup operations. We can use this log to restore the database to the point of the last committed transaction. We can use the Versant Fault Tolerant Server, which is a synchronous replicator, to maintain hot standby databases. The contents of one database can be mirrored to another local or remote database to form a so-called replica pair. When one database becomes unavailable, Versant will continue to use the remaining one. When the crashed database returns to an operational state, Versant will automatically resynchronize the two databases. There are some limitations to synchronous replication. Each database can only have a single replica. Cursor queries and event notification only work on a single database and are therefore not transferred to the standby database. We cannot set and undo savepoints or use nested transactions when we use synchronous replication. Schema changes have to be performed while both databases are available. However, if we can live with these limitations, synchronous replication is excellent for maintaining a hot standby database. Alterna-

tively, we can implement asynchronous replication using Versant's event-notification mechanisms.

Versant is a powerful, tunable DBMS with all the necessary facilities to run large-scale production systems. The Versant ODBMS gives DBAs more than enough capabilities to tune the database server's behavior - more than many RDBMSs do. Versant also has extensive distributed database capabilities. Any relational DBA would need to be retrained in managing objects and methods and many other product-specific idiosyncrasies. A skilled DBA with experience in managing complex and tunable systems will appreciate the extensive facilities provided by Versant.

Versant has a pure Java interface and extensive Java support. A DBA will be able to extend the functionality of the Versant Server with user-written methods, which brings it straight into the firing line with the extendible object/relational database servers, or so-called universal servers.

For more details we refer to the web site of Versant

`http://www.versant.com/`

Chapter 8

FastObjects

8.1 Introduction

FastObjects from Poet is a object database technology for Java and C++ applications. FastObject is available for Windows NT 4.0, Windows 2000, Linux, HP-UX, and Solaris development enviroments. There are three different products:

1) FastObjects j2 for embedded real-time Java applications

2) FastObjects e7 for embedded C++ and Java applications

3) FastObjects t7 the multi-user database optimized for C++ and Java.

FastObjects is purely object-oriented. It maps objects that we created in Java or C++ onto objects in the database. Our object network is simply mapped 1:1 from main memory to the database. This kind of direct mapping enables the database schema to be generated automatically. When we implement a class in our programming language, we are therefore also defining the database schema for this class at the same time. The database knows what an object is and also recognizes the relationships (references, pointers) between objects. These are simply stored together with the objects themselves, and are therefore reproduced without doing anything next time the data is read out of the database.

For more details we refer to the web site of FastObjects:

`http://www.fastobjects.com/`

Chapter 9

Data Mining

9.1 Introduction

There are several definitions for data mining (sometimes called data or knowledge discovery). For example "search for valuable information in large volumes of data" or "exploration and analysis, by automatic or semi-automatic means, of large quantities of data in order to discover meaningful patterns and rules. Thus, generally, data mining is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both. For example, data mining makes it possible to analyze routine business transactions and glean a significant amount of information about individuals buying habits and preferences. Data mining is the search for relationships and global patterns that exist in large databases, but are hidden among the vast amount of data, such as relationship between patient data and their medical diagnosis.

Data mining is not data warehousing. In data warehousing we create stores of informational data – data that is extracted from the operational data and then transformed for end-user decision making. For example, a data warehousing tool might copy all the sales data from the operational database, perform calculations to summarize the data, and write the summarized data to a separate database from the operational data. End-users can query the separate database (the warehouse) without impacting the operational databases.

Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases.

Many models of reality are probabilistic. For example, not everyone orders crips with their beer, but a certain percentage does. Inferring such probabilistic knowledge from databases is one of the major challenges for data mining. An example of this class is: "what is the probability that a given policy-holder will file a claim with the insurance company in the next year". A risk-profile is then a description of a group of insurants that have the same probability for filing a claim.

Although data mining is a relatively new term, the technology is not. Companies have used powerful computers to sift through volumes of supermarket scanner data and analyze market research reports for years. However, continuous innovations in computer processing power, disk storage, and statistical software are dramatically increasing the accuracy of analysis while driving down the cost.

For example, one Midwest grocery chain used the data mining capacity of Oracle software to analyze local buying patterns. They discovered that when men bought diapers on Thursdays and Saturdays, they also tended to buy beer. Further analysis showed that these shoppers typically did their weekly grocery shopping on Saturdays. On Thursdays, however, they only bought a few items. The retailer concluded that they purchased the beer to have it available for the upcoming weekend. The grocery chain could use this newly discovered information in various ways to increase revenue. For example, they could move the beer display closer to the diaper display. And, they could make sure beer and diapers were sold at full price on Thursdays.

Data are any facts, numbers, or text that can be processed by a computer. Today, organizations are accumulating vast and growing amounts of data in different formats and different databases. This includes:
a) operational or transactional data such as, sales, cost, inventory, payroll, and accounting
b) nonoperational data, such as industry sales, forecast data, and macro economic data
meta data - data about the data itself, such as logical database design or data dictionary definitions

The patterns, associations, or relationships among all this data can provide information. For example, analysis of retail point of sale transaction data can yield information on which products are selling and when.

Information can be converted into knowledge about historical patterns and future trends. For example, summary information on retail supermarket sales can be analyzed in light of promotional efforts to provide knowledge of consumer buying behavior. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts.

Dramatic advances in data capture, processing power, data transmission, and storage capabilities are enabling organizations to integrate their various databases into

data warehouses. Data warehousing is defined as a process of centralized data management and retrieval. Data warehousing, like data mining, is a relatively new term although the concept itself has been around for years. Data warehousing represents an ideal vision of maintaining a central repository of all organizational data. Centralization of data is needed to maximize user access and analysis. Dramatic technological advances are making this vision a reality for many companies. And, equally dramatic advances in data analysis software are allowing users to access this data freely. The data analysis software is what supports data mining.

Data mining is primarily used today by companies with a strong consumer focus - retail, financial, communication, and marketing organizations. It enables these companies to determine relationships among internal factors such as price, product positioning, or staff skills, and external factors such as economic indicators, competition, and customer demographics. And, it enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to drill down into summary information to view detail transactional data.

With data mining, a retailer could use point-of-sale records of customer purchases to send targeted promotions based on an individual's purchase history. By mining demographic data from comment or warranty cards, the retailer could develop products and promotions to appeal to specific customer segments.

For example, Blockbuster Entertainment mines its video rental history database to recommend rentals to individual customers. American Express can suggest products to its cardholders based on analysis of their monthly expenditures.

WalMart is pioneering massive data mining to transform its supplier relationships. WalMart captures point-of-sale transactions from over 2,900 stores in 6 countries and continuously transmits this data to its massive 7.5 terabyte Teradata data warehouse. WalMart allows more than 3,500 suppliers, to access data on their products and perform data analyses. These suppliers use this data to identify customer buying patterns at the store display level. They use this information to manage local store inventory and identify new merchandising opportunities. In 1995, WalMart computers processed over 1 million complex data queries.

The National Basketball Association (NBA) is exploring a data mining application that can be used in conjunction with image recordings of basketball games. The Advanced Scout software analyzes the movements of players to help coaches orchestrate plays and strategies.

While large-scale information technology has been evolving separate transaction and analytical systems, data mining provides the link between the two. Data mining software analyzes relationships and patterns in stored transaction data based on open-ended user queries. Several types of analytical software are available: statistical, machine learning, neural networks, genetic algorithm and genetic programming.

Generally, any of four types of relationships are sought:

a) **Classes:** Stored data is used to locate data in predetermined groups. For example, a restaurant chain could mine customer purchase data to determine when customers visit and what they typically order. This information could be used to increase traffic by having daily specials.

b) **Clusters:** Data items are grouped according to logical relationships or consumer preferences. For example, data can be mined to identify market segments or consumer affinities.

c) **Associations:** Data can be mined to identify associations. The beer-diaper example is an example of associative mining.

d) **Sequential patterns:** Data is mined to anticipate behavior patterns and trends. For example, an outdoor equipment retailer could predict the likelihood of a backpack being purchased based on a consumer's purchase of sleeping bags and hiking shoes.

Data mining consists of five major elements:

1. Extract, transform, and load transaction data onto the data warehouse system.

2. Store and manage the data in a multidimensional database system.

3. Provide data access to business analysts and information technology professionals.

4. Analyze the data by application software.

5. Present the data in a useful format, such as a graph or table.

6. Different levels of analysis are available:

Artificial neural networks: Non-linear predictive models that learn through training and resemble biological neural networks in structure.

Genetic algorithms, Genetic programming and Genetic expression programing: Optimization techniques that use processes such as genetic combination, mutation, and natural selection in a design based on the concepts of natural evolution.

Decision trees: Tree-shaped structures that represent sets of decisions. These decisions generate rules for the classification of a dataset. Specific decision tree methods include Classification and Regression Trees (CART) and Chi Square Automatic Interaction Detection (CHAID). CART and CHAID are decision tree techniques used for classification of a dataset. They provide a set of rules that we can apply to a new (unclassified) dataset to predict which records will have a given outcome. CART segments a dataset by creating 2-way splits while CHAID segments using chi square tests to create multi-way splits. CART typically requires less data preparation than CHAID.

Nearest neighbor method: A technique that classifies each record in a dataset based on a combination of the classes of the k record(s) most similar to it in a historical dataset. Sometimes called the k-nearest neighbor technique.

Rule induction: The extraction of useful if-then rules from data based on statistical significance.

Data visualization: The visual interpretation of complex relationships in multidimensional data. Graphics tools are used to illustrate data relationships.

For example newer SQL implementations include beside the aggregate functions

```
AVG               average
COUNT             count how many items
COUNT(DISTINCT)   count of the number of different non-null values
MAX               maximum value
MIN               minimum value
SUM               add the values
```

now also

```
STD, STDDEV   sample standard deviation
VAR, VARP     sample variance
```

For example, given the table `data`

```
number    value
======    =====
  0        0.50
  1        0.75
  2        0.40
==============
```

Using the command in mySQL

```
SELECT avg(value), std(value) FROM data;
```

we find

```
 avg(value)    std(value)
============    ==========
  0.55           0.1472
========================
```

Today, data mining applications are available on all size systems for mainframe, client/server, and PC platforms. System prices range from several thousand dollars for the smallest applications up to 1 million a terabyte for the largest. Enterprise-wide applications generally range in size from 10 gigabytes to over 11 terabytes. NCR has the capacity to deliver applications exceeding 100 terabytes. There are two critical technological drivers:

Size of the database: the more data being processed and maintained, the more powerful the system required.

Query complexity: the more complex the queries and the greater the number of queries being processed, the more powerful the system required.

Relational database storage and management technology is adequate for many data mining applications less than 50 gigabytes. However, this infrastructure needs to be significantly enhanced to support larger applications. Some vendors have added extensive indexing capabilities to improve query performance. Others use new hardware architectures such as Massively Parallel Processors to achieve order-of-magnitude improvements in query time. For example, Massively Parallel Processors systems from NCR link hundreds of high-speed Pentium processors to achieve performance levels exceeding those of the largest supercomputers.

9.2 Example

Your friend opens a shop where he sells liqueur, cigarettes, milk, mineral water, cheese, cold meat and coffee. On the first day he had 20 customers. He writes down in a table what each customer buys and also takes into account whether the customer is male or female. The table looks like this (the customers are numbered from 0 to 19)

customer	sex	liqueur	cigarettes	milk	water	cheese	meat	coffee
0	m	yes	yes	no	no	no	no	no
1	f	no	no	yes	no	yes	yes	yes
2	f	no	yes	no	yes	no	no	yes
3	f	yes	yes	no	no	no	no	yes
4	f	no	yes	yes	yes	no	yes	yes
5	m	yes	no	no	no	yes	yes	yes
6	m	yes	yes	no	no	yes	no	no
7	f	yes	no	no	no	no	yes	yes
8	f	no	no	yes	yes	yes	no	no
9	f	yes	yes	no	no	no	no	no
10	f	no	yes	yes	yes	no	no	no
11	m	no	yes	yes	yes	yes	yes	no
12	f	yes	no	no	no	no	no	yes
13	f	no	yes	no	no	no	no	yes
14	m	yes	yes	no	no	yes	yes	no
15	f	yes	no	no	yes	no	no	yes
16	f	no	yes	yes	yes	yes	yes	no
17	f	no	yes	yes	no	no	no	yes
18	m	no	yes	no	no	yes	yes	no
19	m	yes	no	no	no	yes	yes	no

Ihr Freund fraegt Sie als den fuehrenden Datenbank Experten um Rat welche Schlussfolgerung aus der Tablelle gezogen werden kann. Diskutiere.

Your friend asks you, as the leading expert in databases, for advice as to the conclusions he can draw from this table. Discuss!

Bibliography

[1] Lockmann David, *Teach Yourself Oracle8 Database Development in 21 Days*, SAMS Publishing, Indianapolis, 1997

[2] Tan Kiat Shi, Willi-Hans Steeb and Yorick Hardy, *SymbolicC++: An Introduction to Computer Algebra Using Object-Oriented Programming*, 2nd edition Springer-Verlag, London, 2000

Index

`DISTINCT`, 203
`TreeMap`, 14
`notify()`, 131
`repaint`, 120
`run`, 117
`synchronized`, 126
`wait()`, 131

Access operator, 66
ACID, 103
Associative arrays, 16
Atomicity, 103

Bookmark, 74

Cartesian product, 33
Consistency, 103
Cursor, 102

Deadlock, 104
Deadlocking, 109
Dirty read, 104
Durability, 103
Dynamic cursor, 74

Encapsulation, 179
Entity, 3, 84

Foreign key, 62
Foreign keys, 92
Full outer join, 70
Functional dependency, 34

Impedance mismatch, 177
Impedance mismatch problem, 178
Inner join, 68
Isolation, 103

Join, 66

Leaves, 31
Lock, 108

Multitasking, 117
Multithreading, 117

Normalization, 88

Outer join, 69
Owners, 92

Parentless child, 92
Phantom read, 104
Primary key, 38, 62
Priority, 123
Private member functions, 192
Protected member functions, 192
Public member functions, 192

Read locks, 108
Referential integrity, 38, 62, 66, 92
Result set, 102
Root, 31
Row lock, 108

Savepoints, 102
Serializability, 103
SQL, 35
Static cursors, 74
Stored procedure, 71
Subquery clause, 44
Synchronization problem, 131

Table lock, 108
Thread, 117
Transaction, 101
Transitive dependency, 90
Transitively, 90

View, 60

Write locks, 108