Relational Database and SQL

Prepared by Salvatore T. March, Jungpil Hahn and Jinsoo Park
For Classroom Use Only

Introduction

The database layer of an information system provides its requisite persistence mechanism. That is, an information system utilizes a database management system (DBMS) to maintain its data, stored permanently in secondary memory (disk). Relational database technology is the dominant type of DBMS in commercial information system applications.

A Relational DBMS (RRDBMS) defines its data as a collection of *tables* each having well-defined *rows* and *columns*. Each table holds data about a singe type of "thing" about which data is maintained. The rows represent the "things" (e.g., customers, products, orders, employees, accounts, and transactions). The columns represent the "facts" about those things that need to be maintained (e.g., name, address and phone number for customers; product number, description and price for products). Each column in each row can hold only one value. Hence column in each row in a table holds the data value for the fact represented in that column about the "thing" represented in that row.

Suppose, for example, it was necessary to store data about the departments of a company and the employees who work for them. Since departments are different types of things from employees, one table would be defined for departments and a separate table would be defined for employees. The department table would have one row for each department and one column for each fact about departments the needed to be maintained about departments. Depending on the information system requirements, the department table would have columns such as, department number, department name, department budget, and department manager. Similarly, the employee table would have one row for each employee and one column for each fact about employees that needed to be maintained. The employee table would have columns such as employee number, employee name, social security number, number of exemptions, weekly salary, year-to-date gross pay, and department for which the employee works.

Figure 1 shows a pictorial representation of this database, including sample data. These tables must be defined within the RDBMS, data must be entered into them and, of course, data must be accessed from them to meet the requirements of the information system. To do so, the RDBMS must have a language to define the tables and one to update and access their data. The dominant language implemented in commercial RDBMSs is the Structured Query Language (SQL). SQL provides both data definition and data manipulation capabilities. This module provides an introduction to relational database concepts and to SQL.

Department Table

deptNo	deptName	deptBudget	deptManager
1	Marketing	220,000	4
2	Operations	400,000	6
3	Accounting	100,000	3

Employee Table

empNo	empName	ssn	exemptions	weeklySalary	ytdGross	deptNo
1	Smith, Joseph	111-11-1111	1	4000	40000	3
2	Jones, David	222-22-2222	2	3200	32000	2
3	Olson, Jane	333-33-3333	1	3800	38000	3
4	Neff, Arnold	444-44-4444	3	2300	23000	1
5	Homes, Denise	555-55-5555	2	1400	14000	1
6	Naumi, Susan	666-66-6666	2	3500	35000	2
7	Young, John	777-77-7777	1	3000	21000	3

Figure 1 - An Example Relational Database

In addition to implementing SQL, many commercial RDBMSs include a graphical, or direct manipulation interface through which data definition and data manipulation commands can be entered. The DBMS translates them into SQL. RDBMSs may also include graphical screen and report "painters" which enable visual programming of these interfaces. These also generate SQL retrieval queries to obtain data for display, and, in the case of input screens generate SQL update queries and commands to write data back to the database. These SQL queries are typically embedded in generated procedural code in the *host language* supported by the DBMS. The host language may be a general purpose programming language such as COBOL, C, C++, or Java or it may be a proprietary language as in Microsoft Access (which supports a variation of Visual Basic) or PowerBuilder.

Client-side development environments such as VisualBasic and PowerBuilder similarly use SQL or a graphical interface that produces SQL to obtain data for display, and, in the case of update screens, generate SQL update queries and commands to write data back to the database. For example, the `RecordSource` property of a VisualBasic data control specifies an SQL query. Furthermore, SQL queries are often embedded in modules or functions implemented in such environments to obtain data for procedural processing and to execute updates to the database. For example, the VisualBasic methods `CreateDynaset` and `ExecuteSQL` can use SQL queries to specify database retrievals and updates, respectively.

Background

The Relational Data Model (RDM) was posed by Edward Codd, a researcher at IBM's San Jose Research center, in a seminal article published in the *Communications of the ACM* in 1970 [Codd, 1970]. The RDM was presented as a conceptually simple, but mathematically sound, representation of the way in which data are stored within a

computer database. It was refined and extended by numerous researchers during the following thirty years and continues to be the subject of ongoing research.

The RDM is the basis for Relational Database Management Systems (RDBMSs) which have become extremely popular in commercial information systems. The de-facto standard language used in commercial RDBMSs is the Structured Query Language (SQL). Originally called SEQUEL (*S*tructured *E*nglish *QUE*ry *L*anguage), and still pronounced that way, SQL was designed and implemented by IBM in the mid 1970s as the interface for an research-oriented relational database system called SYSTEM R.

SQL has two components: (1) a Data Definition Language (DDL) in which the structure of a database is defined and (2) a Data Manipulation Language (DML) in which retrieval and update *queries* are specified. The SQL DML is *non-procedural* in the sense that it specifies the content of the output (i.e., what), not the procedure for producing it (i.e., how). RDBMSs typically have a *query optimizer*, that determines an efficient algorithm for producing the result, depending on the database structure, volume, and existing indexes. One of the important differentiating factors among commercial RDBMSs is the quality of the query optimizer.

Relational Database Structure (DDL)

As discussed above, a relational database is defined as a set of *tables* (relations), each having *rows* (tuples, records) and *columns* (attributes, data items). One table is typically defined for each *class* or type of *entity* in the domain of the information system. Rows correspond to the *objects* or *instances* in that class. Columns define the data elements in the table. These correspond to *descriptors* or *attributes* of the objects in that class. Each row contains one data value for each column. That is, columns contain *single-valued* attributes. Each column in a table may be specified to draw values from a *domain* defined within the database or to be a predefined *data type* such as integer or string. A column may also be specified to be *not null*, disallowing rows with a null value in that column. If a column is not specified to be, not null, then null values are allowed (i.e., empty cells are allowed).

The database illustrated in Figure 1 has two tables, `Department` and `Employee`. The `Department` table has columns for Department Number (`deptNo`), Name (`deptName`) Budget (`deptBudget`), and the Employee Number of the employee who manages the department (`deptManager`). Similarly, the `Employee` table has columns for Employee Number (`empNo`), Name (`empName`), Social Security Number (`ssn`), Number of Exemptions (`exemptions`), Weekly Salary (`weeklySalary`), Year To Date Gross pay (`ytdGross`) and the Department Number of the department for which the employee works (`deptNo`). Each department has a corresponding row in the `Department` table with one value in each column containing the value of that attribute for that department. Similarly, each employee has a corresponding row in the `Employee` table.

A table may have column or set of columns designated as its *primary key*. Each row must have a unique value for its primary key (or a unique combination of values if there

are multiple columns in the primary key – a composite key). `deptNo` is the primary key of the `Department` table; `empNo` is the primary key of the `Employee` table. Primary keys represent the *identifier* of the type of thing (*class, entity*) represented in the table. Primary keys are typically underlined in the textual representation of a table, as done in Figure 1. Primary key columns **must** be specified as, not null. Relational database theory has introduced the concept of *normalization* or *normal forms* to define "well-formed" tables. These are rules aimed at eliminating redundancy within the database structure[1].

Different types of things represented in a database are often related to each other and it is often important to capture those relationships in the database. Employees and departments, for example relate to each other in two different ways. Each employee **works for** a department and each department has an employee who **manages** it. The first relationship is termed a *one-to-many* with Department on the *one side* and Employee on the *many side*. That is, one department may have many employees who work for it but each employee works for only one department. The second relationship is termed a *one-to-one* relationship. That is, each department may have one employee who manages it and an employee can manage one department.

Relationships must actually be more precisely defined if the DBMS is to enforce appropriate data integrity. Is it possible, for example, for an employee not to work for any department? Is it possible for a department to not have a manager? That is, the minimum and maximum cardinalities must be specified for each table in the relationship. For the first relationship assume that, in this business, each employee must work for exactly one department (a minimum and maximum of one) and a department may have any number of employees, but is not required to have any (minimum of zero and an unlimited maximum). For the second relationship assumes that a department has at most one manager, but does not need to have a manager specified (minimum of zero and maximum of one) and an employee may manage zero or one department (minimum of zero and maximum of one). The ER (Entity-Relationship) schema of the relational database is presented in Figure 2.

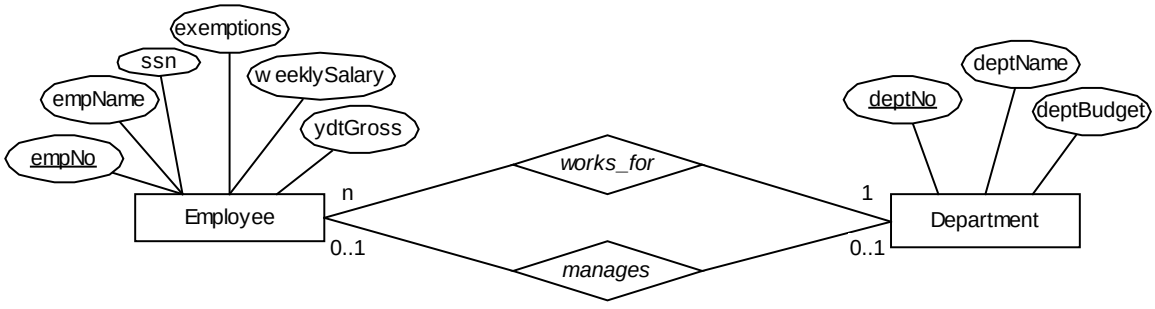

Figure 2 - ER Schema of the Relational Database in Figure 1

Relationships such as these are represented in a relational database using corresponding column(s) in each related table. The column(s) in one table must be the primary key of

1 Refer to Kent (1983) for more details on database normalization.

that table; the column(s) in the related table is (are) termed a *foreign key*. Although primary keys, and thus, foreign keys may include more than one column, to avoid repeatedly writing, "column or columns," the following discussion assumes that primary keys, and thus foreign keys, are each a single column.

In a relational database, a one-to-many relationship is represented using a foreign key column in the table on the *many* side. The values in that column must come from the primary key column of the table on the *one* side. A one-to-one relationship may be represented in either table, using a foreign key column in one table holding values from the primary key column of the table on the other side. If the minimum for the table on the one side is one, then the foreign key column must be designated as "NOT NULL". The two relationships between departments and employees described above, for example, are represented in the tables of Figure 1 as follows:

(1) Employee works for Department[2] – `deptNo` is a foreign key column in the `Employee` table, corresponding to the primary key column, `deptNo`, in the `Department` table, designated to be `NOT NULL`, and
(2) Employee manages Department[3] – `deptManager` is a foreign key in `Department` table corresponding to the primary key column, `empNo` in the Employee table, not designated to be `NOT NULL`.

As shown in Figure 1, David Jones, employee number 2, works for the Operations department, department number 2. The employee, Susan Naumi, having employee number 6, manages this department.

The "Employee manages Department" relationship is represented by a foreign key in the Department table both for efficiency reasons and for conceptual reasons. Even though the one-to-one relationship can be represented in either table, it is generally represented by putting the primary key of a table into a table that has *total participation* in the relationship. Relatively few employees manage departments, but most, if not all, departments have managers. Thus if this relationship was represented in the `Employee` table by a foreign key, all rows corresponding to employees who do not manage departments would have a null value for that column. Furthermore, it is conceptually more natural to associate the managing function with departments than with employees.

If the minimum on the *one* side of a relationship is one, i.e., the foreign key is `NOT NULL`, then the relationship is said to represent a *referential integrity constraint* or an *existence dependency constraint*. That is, a row cannot exist in the table on the many side unless there is a related row in the table on the one side. In the above specification, an employee (row) cannot exist in the database without a related department (row).

If a relationship represents a referential integrity constraint, then the DBMS **must not** allow row insertions, deletions and updates that would violate that constraint. For

[2] More precisely, you can read the ER schema in Figure 2 as follows: "*Each* Employee works for *one* Department (left-to-right)," and "*Each* Department hires (or is worked by) *any* (zero or more) Employees (right-to-left)."
[3] Similarly, you can interpret the ER schema in Figure 2 as follows: "*Each* Employee manages *at most one* (zero or one) Department (left-to-right)," and "*Each* Department is managed by *at most one* Employee (right-to-left)."

example, the DBMS must reject an attempt to add a row in the `Employee` table whose `deptNo` column contains a value not in the `deptNo` column of some row in the `Department` table. Similarly, the DBMS must reject an update to the `deptNo` column of an existing employee row that would violate this constraint.

Deletions that violate a referential integrity constraint can be handled in one of two ways. First, as with insertions, they can be rejected. Alternately, the deletion could be *cascaded* to the rows in the related table. That is, all rows related to the deleted row could also be deleted, thus retaining the integrity of the database. This specification is appropriately termed, *cascade delete*. While somewhat dangerous, cascade delete can be quite useful. For example, if a payroll system keeps a record of an employee's dependents in a `Dependents` table related to the `Employee` table, referential integrity with cascade delete is probably appropriate. That is, a dependent cannot be added without a related employee and when an employee is deleted all of the employee's dependents are also deleted. If cascade delete is not specified then the dependents must be deleted separately, before the employee can be deleted.

SQL has seven basic types of data definition statements,

1. **Create Database** – to create and name a database,
2. **Create Domain** – to define legal values for columns, in addition to the system defined data types,
3. **Create Table** – to define a table and all of its columns and relationships,
4. **Create View** – to define virtual tables derived from existing tables based on a query,
5. **Alter Table** – to modify the column and relationship definitions of an existing table,
6. **Drop Table** – to remove a table from a database,
7. **Drop View** – to remove a view from a database.

Different commercial RDBMSs implement slightly different variations of SQL. This is particularly true with data types. Microsoft Access, for example, has the data types, `text`, `integer`, `double`, and `timestamp`, while Microsoft SQLServer has the corresponding data types, `char` and `varchar`, `int`, `smallint` and `tinyint`, `float` and `numeric`, and `datetime` and `smalldatetime`, among others. Furthermore, Access does not support the definition of domains or views or database triggers. Nor does it support the `Create Database` statement. Databases must be created using the Access command menu (File => New => Database).

Specification of the complete SQL DDL syntax is beyond the scope of this module. However, following is the basic SQL syntax for creating the `Department` and `Employee` tables. First, a database named, `Emp-Dept` is created using,

```
CREATE DATABASE Emp-Dept;
```

Next, the Department and Employee tables are created in that database,

```
CREATE TABLE Department
(    deptNo          INTEGER NOT NULL,
     deptName        VARCHAR(20),
     deptBudget      DOUBLE,
     deptManager     INTEGER,
     PRIMARY KEY     (deptNo)
);

CREATE TABLE Employee
(    empNo           INTEGER NOT NULL,
     empName         VARCHAR(20),
     ssn             CHAR(11),
     exemptions      INTEGER,
     weeklySalary    DOUBLE,
     ytdGross        DOUBLE,
     deptNo          INTEGER NOT NULL,
     PRIMARY KEY (empNo),
     FOREIGN KEY (deptNo) REFERENCES Department (deptNo)
);
```

Primary keys are specified in parentheses following the keyword, `PRIMARY KEY`. Foreign keys are specified in a phrase of the form,

```
FOREIGN KEY (column-name(s)) REFERENCES table-name (column-name(s))
```

The parameter following the `FOREIGN KEY` keyword is the name of the column containing the foreign key (in the current table). It must be followed by the keyword, `REFERENCES`, followed by the name of the related table, followed, in parentheses, by the name of the primary key column in that table.

Since the `Department` table is created *before* the `Employee` table, `deptManager` cannot be specified as a foreign key in the initial definition of the `Department` table. It must be specified as such after the `Employee` table is created using an `ALTER TABLE` command.

```
ALTER TABLE Department
ADD FOREIGN KEY (deptManager) REFERENCES Employee (empNo);
```

A referential integrity constraint is established by specifying a foreign key column to be, not null. For example, `deptNo` in the `Employee` table is specified to be not null and specified to be a foreign key. This establishes the *referential integrity constraint* that each employee must work for exactly one department. Since `deptManager` is not specified to be, `NOT NULL`, the database definition allows departments to exist without a manager. In fact, it is not possible to specify both working for and managing to include referential integrity constraints. Doing so would cause the system to reject all attempts to add rows to either table! That is, when the database is created, both tables are empty. Adding an employee would require the department for which the employee works to exist in the `Department` table, but that table is empty. However, adding a department would require the employee who manages the department to exist in the `Employee` table, but that table is also empty.

Hence, only one of these constraints can be defined in the database structure. Enforcing the other constraint requires the creation of a *transaction* in which rows can be added to both tables before the constraint is tested. If the constraint is met, then the changes are *committed* to the database. If not, then all changes made in the transaction are *rolled back,* restoring the database to its state prior to the transaction.

Microsoft access supports SQL DDL. To execute DDL commands use the cascading menu Query => SQL Specific => Data Definition in its Query module. Then enter the desired DDL commands and execute the query (Query => Run). However, data definition is more easily accomplished using its graphical Table Design interface and its graphical Relationships interface.

Relational Data Manipulation (DML)

Data is obtained from a relational database using SQL retrieval queries. It is written to the database using SQL update queries or using SQL commands that support "record at a time" processing via the concept of a *cursor*. This section will address SQL retrieval and update queries. Processing via cursors is beyond the scope of this module, as are the details of stored procedures and database triggers.

Query specifications in SQL are based on *relational algebra* or *relational calculus*. What is important about that concept is that the result of a query is itself, another relation, which can be stored or used in other queries. There are four basic types of SQL queries,

1. **Select** – to retrieve selected rows and columns from selected tables, performing specified calculations on specified columns,
2. **Insert** – to add rows to a table,
3. **Update** – to modify the values in a table,
4. **Delete** – to remove rows from a table.

Select Queries

`SELECT` queries are used to obtain data from a database. Again, once retrieved, the data may be further processed and formatted for display on a screen or in a report. The purpose of the query is to obtain the necessary data. The syntax of a `SELECT` query is as follows. For clarity, keywords are capitalized and bold.

```
SELECT [ALL | DISTINCT] <column expressions>
FROM <table names and optional join specifications>
WHERE <row inclusion conditions>
GROUP BY <grouping expressions>
HAVING <group selection conditions>
ORDER BY <sorting expressions>;
```

Each line in the above specification, beginning with an SQL keyword is termed a *clause*. Clauses do not need to begin on a separate line. They will be written so for clarity. A `SELECT` query must have at least two clauses. It must begin with a **Select** clause that lists the column expressions (columns and column calculations) needed, followed by a **From**

clause that names the tables from which those columns are accessed and specifies any Join conditions if the columns are from more than one table. It must end with a semicolon. Hence, a minimal SQL query has the form:

```
SELECT [ALL | DISTINCT] <column expressions>
FROM <table names and optional join specifications>;
```

where `[ALL | DISTINCT]` means that, optionally, one of the keywords `ALL` or `DISTINCT` may be specified. If `ALL` is specified, then, all rows selected in the query are included in the result, even if there are duplicate rows. If `DISTINCT` is specified then, any duplicate rows are removed from the query result. If neither is specified, `ALL` is assumed.

`<column expressions>` is a list of columns and column calculations, separated by commas, specifying the columns to include in the query result. `<table names and optional join specifications>` is simply a table name for single table queries or an expression specifying how the tables are to be combined for multi-table (or `JOIN`) queries. `JOIN` queries, which typically combine tables along relationships, are discussed in the next section.

The following simple `SELECT` query:

```
SELECT empNo, empName, ssn, exemptions, weeklySalary, ytdGross,
     deptNo
FROM Employee;
```

produces a result table, termed a *recordset*, containing the entire Employee table:

empNo	empName	ssn	exemptions	weeklySalary	ytdGross	deptNo
1	Smith, Joseph	111-11-1111	1	4000	40000	3
2	Jones, David	222-22-2222	2	3200	32000	2
3	Olson, Jane	333-33-3333	1	3800	38000	3
4	Neff, Arnold	444-44-4444	3	2300	23000	1
5	Homes, Denise	555-55-5555	2	1400	14000	1
6	Naumi, Susan	666-66-6666	2	3500	35000	2
7	Young, John	777-77-7777	1	3000	21000	3

Columns appear in the order specified in the Select clause. Column headings default to the column names. Note that the column headings are for display purposes only and are not a separate row of the recordset which may, for example, be sent across a network to a *Client* computer for display and manipulation[4]. A different heading can be specified for a column expression using the keyword `AS`, followed by the desired column heading. In Microsoft Access, if the desired heading is a reserved word or contains embedded blanks, it must be enclosed in brackets. For example, the following query changes the column headings for the `empNo` and exemptions columns, but leaves the other column heading at their default values.

[4] Note that if the client computer changes any values in the record set, these changes are not automatically reflected in the database. To change the database, the client computer must compose an update query and request its execution by the DBMS on the Server computer.

```
SELECT empNo AS [Emp Number], empName, ssn, exemptions AS Exempt,
     weeklySalary, ytdGross, deptNo
FROM Employee;
```

Since there is no `ORDER BY` clause, the row ordering is arbitrary. If row ordering is desired, it must be specified in an `ORDER BY` clause. Often a DBMS will use either chronological or primary key order if an `ORDER BY` clause is not specified. However, this varies between different DBMSs.

Column names must be qualified by their table name if there is any ambiguity (i.e., if there are duplicate column names in the tables involved in the query). A fully qualified column name is `<table name>.<column name>`. For example, the `empNo` column in the `Employee` table has the fully qualified name, `Employee.empNo`. A shorthand notation for "all columns" in a table is `<table name>.*`, or just `*`. Hence an equivalent query to that given above is,

```
SELECT Employee.*
FROM Employee;
```

A `SELECT` query may have additional clauses to specify row selection and ordering in the result table. A `WHERE` clause specifies row selection criteria. An `ORDER BY` clause specifies the ordering criteria for the result table. The basic syntax for these clauses is:

```
WHERE <condition list>
ORDER BY <sorting list>
```

where `<condition list>` defines rows to be included in the result table; `<sorting list>` lists the columns used to sort the result, each optionally followed by `ACS` for ascending order or `DESC` for descending order. If neither is specified, `ASC` is assumed. Again, `SELECT` and `FROM` clauses are required, `WHERE` and `ORDER BY` clauses are optional. A query may contain either, both, or neither clause depending on the retrieval requirements.

The following query retrieves the Employee Number, Name, and Salary of all employees with salary greater than $3,500, sorted by employee name.

```
SELECT empNo, empName, weeklySalary
FROM Employee
WHERE weeklySalary > 3500
ORDER BY empName;
```

The result of this query would be displayed as:

empNo	EmpName	weeklySalary
3	Olson, Jane	3800
1	Smith, Joseph	4000

The `<condition list>` in a `WHERE` clause may include the comparison operators `=`, `>`, `>=`, `<`, `<=` and the logical operators, `AND`, `OR` and `NOT` as well as any built-in functions

supplied by the DBMS. Built-in functions vary widely in commercial DBMSs, particularly with respect to date and text manipulation and type conversions. Parentheses may be used as necessary and are recommended for complex `WHERE` clauses to enhance readability. For example to include employees in department 3 who have weekly salary over $3,500 and employees in department 2 who have weekly salary over 3000 and all employees in department 1, use the following `WHERE` clause,

```
SELECT empNo, empName, weeklySalary
FROM Employee
WHERE (deptNo = 3 AND weeklySalary > 3500)
     OR (deptNo = 2 AND weeklySalary > 3000)
     OR (deptNo = 1)
```

which would result in:

empNo	EmpName	weeklySalary
1	Smith, Joseph	4000
2	Jones, David	3200
3	Olson, Jane	3800
4	Neff, Arnold	2300
5	Homes, Denise	1400
6	Naumi, Susan	3500

The `<sorting list>` of an `ORDER BY` clause lists columns from major to minor precedence, separated by commas. Each may specify `ASC` for ascending or `DESC` for descending. For example, to sort the result table in the above query alphabetically (ascending) within descending department numbers, use the following `ORDER BY` clause,

```
SELECT empNo, empName, weeklySalary
FROM Employee
WHERE (deptNo = 3 AND weeklySalary > 3500)
     OR (deptNo = 2 AND weeklySalary > 3000)
     OR (deptNo = 1)
ORDER BY deptNo DESC, empName
```

which would result in:

empNo	EmpName	weeklySalary
3	Olson, Jane	3800
1	Smith, Joseph	4000
2	Jones, David	3200
6	Naumi, Susan	3500
5	Homes, Denise	1400
4	Neff, Arnold	2300

More complex queries can perform column aggregations such as averages, counts, maximums, minimums, and sums (*aggregation* queries) and groupings (*Group By* queries), access data from multiple tables (e.g., *join* queries), and use `SELECT` queries within the `WHERE` clause to define conditions for row inclusion (*Nested* queries). Aggregate and nested queries are discussed below. `JOIN` queries are discussed in the

next section. Nested queries are discussed in the section following the section on join queries.

Aggregation Queries

An aggregation query aggregates or computes a result from all specified rows in the result table. The syntax for an aggregation is,

```
<aggregate function> ([ALL | DISTINCT] <column expression>)
```

where, `<aggregate function>` is one of the aggregate functions, `AVG()` for average, `COUNT()` for number of columns, `MAX()` for maximum value, `MIN()`for minimum value or `SUM()` for sum of all values. Again, `ALL` and `DISTINCT` are optional, specifying if duplicates are to be included (`ALL`) or excluded (`DISTINCT`) in the aggregation operation. The default is `ALL` if neither is specified. The `<column exprexssion>` is any valid expression involving columns in the table. All commercial RDBMSs provide the above aggregation functions. Microsoft Access includes the additional aggregation functions, `FIRST()` for first value, `LAST()` for last value, `STDEV()`for standard deviation and `VAR()` for variance.

Consider, for example, the following aggregation query,

```
SELECT Count(empNo), Avg(weeklySalary), Max(weeklySalary),
    Sum(weeklySalary), Avg(weeklySalary * 1.1)
FROM Employee
```

It produces the following, one row, result table:

CountOf empNo	AvgOf weeklySalary	MaxOf weeklySalary	SumOf weeklySalary	Expr1
7	3028.571428	4000	1400	3331.428571

Aggregation queries, can, of course, include selection criteria to restrict the rows included in the aggregation functions, as illustrated by the following aggregation query and result:

```
SELECT Count(empNo), Avg(weeklySalary), Max(weeklySalary),
    Sum(weeklySalary)
FROM Employee
WHERE deptNo = 3
```

CountOf empNo	AvgOf weeklySalary	MaxOf weeklySalary	SumOf weeklySalary
3	3600	4000	10800

Group By Queries

Rather than having only one row containing the aggregation results for the entire query, a `GROUP BY` query produces a row containing sub-aggregations (e.g., subtotals) for each value in the specified `GROUP BY` column(s). For example, a `GROUP BY` query could be used to compute the aggregate values for each department, displayed in a separate row of the result table. Each combination of values in the `GROUP BY` columns defines a separate row in the result table. The clause, `GROUP BY deptNo`, for example, produces a result table with one row per department. Each row has a column for each aggregate function

for containing the sub-aggregation for that department. Such an aggregate query and its result are shown below.

```
SELECT deptNo, Count(empNo), Avg(weeklySalary), Sum(weeklySalary)
FROM Employee
GROUP BY deptNo
ORDER BY deptNo
```

deptNo	CountOf empNo	AvgOf weeklySalary	SumOf weeklySalary
1	2	1850	3700
2	2	3350	6700
3	3	3600	10800

More than one column can be specified in a `GROUP BY` clause, column names are separated by commas. In this case, one row appears in the result table for each unique combination of values in the `GROUP BY` columns. Although not necessary, it is often useful to sort the result by the grouping column(s).

A syntactic rule for SQL `GROUP BY` queries is that any columns that appear in the `SELECT` clause without being in an aggregation function must be in the `GROUP BY` clause. All others must be aggregated. In the above example, all columns in the Select clause except `deptNo` must appear in aggregate functions. To illustrate why this is true, consider the following, syntactically incorrect, aggregate query:

```
SELECT deptNo, empNo, Avg(weeklySalary), Sum(weeklySalary)
FROM Employee
GROUP BY deptNo
ORDER BY deptNo
```

The `GROUP BY` clause specifies that there is to be one row per department in the result table. However, there are many employees in each department. Hence, there are many possible values of `empNo` for that column in each row of the result table. Since the DBMS cannot determine which value of `empNo` to use, it must reject the query as being syntactically incorrect. Indeed, an attempt to execute this query will result in an error.

Optionally, selection criteria of two different types can be specified for `GROUP BY` queries. Selecting rows from the original table to be included in the aggregation is specified using a Where clause, as in any other type of `SELECT` query. However, SQL also provides a means of selecting rows from the result table *after* the aggregation has been performed. This is specified using a `HAVING` clause. A single query can include both `WHERE` and `HAVING` clauses.

To illustrate the difference between Where and having, consider the following query.

```
SELECT deptNo, Count(Empno), Avg(weeklySalary), Sum(weeklySalary)
FROM Employee
WHERE weeklySalary <= 3500
GROUP BY deptNo
ORDER BY deptNo
```

The result table is as follows.

deptNo	CountOf empNo	AvgOf weeklySalary	SumOf weeklySalary
1	2	1850	3700
2	2	3350	6700
3	1	3000	3000

This query first selects rows from the `Employee` table whose `weeklySalary` is less than or equal to (≤) $3,500. It then calculates the aggregations, count, average and sum for those rows. Hence, employees 1 and 3, whose weekly salaries are $4,000 and $3,800, respectively, are not included in the aggregations. Since both of these employees work for department 3, that department shows a count of 1 employee rather than 3 (i.e., only one of the three employees with `deptNo` equal to 3 is included in the aggregations).

Now consider the following `GROUP BY` query that uses a `HAVING` clause rather than a `WHERE` clause.

```
SELECT deptNo, Count(Empno), Avg(weeklySalary), Sum(weeklySalary)
FROM Employee
GROUP BY deptNo
HAVING Avg(weeklySalary) <= 3500
ORDER BY deptNo
```

This query calculates the aggregations count, sum, and average using all rows from the `Employee` table. It then uses the conditions specified in the `HAVING` clause to determine which rows should be included in the result table. Its result table is shown below.

deptNo	CountOf empNo	AvgOf weeklySalary	SumOf weeklySalary
1	2	1850	3700
2	2	3350	6700

It only includes those departments whose average weekly salary is less than or equal to 3500. Since the average weekly salary for department 3 is $3,600, (($4,000 + $3,800 + $3,000) / 3), that department is not included in the result table. The average salary for each of the other departments is less than $3,500; therefore they are included in the result table.

Of course, the important issue is to first determine what data the system users require and then to develop the appropriate SQL query to produce it. If the users need aggregations that include only employees who have a weekly salary less than $3,500, then the former query is appropriate. If they need aggregations for departments whose average salary is less than $3,500, then the latter query is appropriate. The determination of these requirements is, again, a principle concern in systems analysis and design activities.

Join Queries

To satisfy user information requirements, it is often necessary to obtain data from two or more tables. *Join queries* are used for this purpose. They combine rows from different tables into a single row in the result table where column values in those rows meet the specified *join conditions*.

Tables are often joined along relationships; that is, the join condition specifies that the *primary key* in one table equals the corresponding *foreign key* in the related table. `Employee` and `Department` tables, for example, have two Primary Key-Foreign Key pairs, `Department.deptNo` (primary key) corresponding to `Employee.deptNo` (foreign key) and `Employee.empNo` (primary key) corresponding to `Department.deptManager` (foreign key). The join condition to combine employees with the departments for which they work is:

```
Employee.deptNo = Department.deptNo
```

Join conditions typically include fully qualified column names using the "dot" notation, `table.column` (some DBMSs require it). In the case of joining employees with the departments for which they work, fully qualified column names must be used since the primary key and foreign key columns have the same name (i.e., `deptNo`).

A join query must specify what to do if there are rows in one table without corresponding rows in the other table. For example, while each employee must work for a department, there may be departments for which no employees work. Should such departments be included in the result table or not? This is specified in the *type* of join as specified below:

Join Type	**Result Table**
`INNER JOIN`	Includes only rows where the `<join condition>` is explicitly satisfied (there are corresponding values in both tables).
`[LEFT | RIGHT] OUTER JOIN`	Includes all rows in the table on the left (or right) but only those rows in the table on the right (or left) where the `<join condition>` is explicitly satisfied (has corresponding values in the other table), where left and right refer to the order in which the tables are listed in the query.
`FULL OUTER JOIN`	Includes all rows in both tables.

There are two different SQL variations for specifying Join conditions. The first, termed, SQL1, has no explicit `JOIN` keyword. Join conditions are specified in the `WHERE` clause. Inner joins are specified using `=`. Outer joins are specified using DBMS specific operators such as `*=` (left), `=*` (right), or `*=*` (full outer). The second, termed SQL2 specifies join conditions in the `FROM` clause. It uses explicit keywords to specify the type of join.

The SQL1 syntax for a `JOIN` query is:

```
SELECT <column expressions>
FROM <table names separated by commas>
WHERE <join condition and other selection conditions>
```

The SQL2 syntax for a `JOIN` query is:

```
SELECT <column expressions>
FROM <table1 name> <join type> <table2 name> ON <join condition>
WHERE <selection conditions>
```

where `<join type>` is one of, `[INNER] JOIN`, `RIGHT [OUTER] JOIN`, `LEFT [OUTER] JOIN`, or `FULL [OUTER] JOIN`. Microsoft Access does not support the `FULL OUTER JOIN`. In addition, the keyword, `OUTER` is optional. If no join type is specified, the default is `INNER JOIN`. Of course, `JOIN` queries can include `ORDER BY` and `GROUP BY` clauses.

SQL1 was the original SQL specification and several RDBMSs continue to support it (virtually all RDBMSs support its syntax for `INNER JOIN`). The newer RDBMSs also support the SQL2 syntax. However, even within the SQL1 and SQL2 specifications, there are slight variations among commercial DBMSs. Both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) are responsible for specifying standards to provide consistency across different vendors' SQL implementations. Even so, numerous variations exist. Each vendor provides documentation for its specific SQL implementation.

Suppose there is a user requirement to produce an alphabetically ordered list of employees who work for departments, including the employee's name and weekly salary, along with the department name and department budget for the department for which the employee works, only including departments for which employees work. The SQL query must join a row from the `Employee` table with a row from the `Department` table when the value of `deptNo` in the `Employee` table matches the value of `deptNo` in the `Department` table. It must be an *inner join*, since it includes rows from the `Employee` table having a matching row in the `Department` table and only rows from the `Department` table having a matching row in the `Employee` table[5].

The appropriate SQL1 query is given by,

```
SELECT empName, weeklySalary, deptName, deptBudget
FROM Employee, Department
WHERE Employee.deptNo = Department.deptNo
ORDER BY empName
```

The corresponding SQL2 query is given by,

[5] In this case, assuming the Employee table is listed first, the INNER JOIN and LEFT OUTER JOIN are equivalent since each employee must work for a department (referential integrity).

```
SELECT empName, weeklySalary, deptName, deptBudget
FROM Employee INNER JOIN Department
     ON Employee.deptNo = Department.deptNo
ORDER BY empName
```

The result table is as follows. It has seven rows, one for each row in the `Employee` table, since each employee work for a department.

empName	weeklySalary	deptName	deptBudget
Homes, Denise	1400	Marketing	220000
Jones, David	3200	Operations	400000
Naumi, Susan	3500	Operations	400000
Neff, Arnold	2300	Marketing	220000
Olson, Jane	3800	Accounting	100000
Smith, Joseph	4000	Accounting	100000
Young, John	3000	Accounting	100000

Again, the *join condition*, `Employee.deptNo = Department.deptNo` is written in the fully qualified form, `<table name>.<column name>`. This form is required since the column name is ambiguous. That is, `deptNo` is a column name in both tables. Specifying,

```
deptNo = deptNo
```

does not make sense and, if the DBMS executed the query would result in every row in the `Employee` table being joined with every row in the `Department` table.

To include all rows in the `Employee` table, but only rows in the `Department` table having a matching row in the Employee table, use an *outer join*, rather than an inner join. An Outer join is specified as including all rows in the *left* or *right* table. The left table is simply the one on the left, preceding the `OUTER JOIN` keyword. The right table is the one on the right, following the `OUTER JOIN` keyword. Hence, this query is specified as,

```
SELECT empName, weeklySalary, deptName, deptBudget
FROM Employee LEFT OUTER JOIN Department
     ON Employee.deptNo = Department.deptNo
ORDER BY empName
```

It uses a *left* outer join because `Employee` is the table to the left of the `OUTER JOIN` keyword. Given the referential integrity constraint that all employees must work for a department, the inner and left outer joins produce exactly equivalent results. To include all `Departments`, even if those for which no employees work, use

```
SELECT empName, weeklySalary, deptName, deptBudget
FROM Employee RIGHT OUTER JOIN Department
     ON Employee.deptNo = Department.deptNo
ORDER BY empName
```

It uses a *right* outer join because `Department` is the table to the right of the `OUTER JOIN` keyword. To include all rows from both tables use a *full outer join* as follows.

```
SELECT empName, weeklySalary, deptName, deptBudget
FROM Employee FULL OUTER JOIN Department
     ON Employee.deptNo = Department.deptNo
ORDER BY empName
```

Again, given the *referential integrity constraint*, the full outer join produces the same result as the `RIGHT OUTER JOIN` with `Department` on the right (or the `LEFT OUTER JOIN` with `Department` on the left). In fact, for this set of data, since each department has employees that work for it, the inner and all three outer joins produce the same result.

As discussed above, `Department` and `Employee` have two relationships, `Employee` works for `Department` and `Employee` manages `Department`. The above queries use the "works for" relationship to join these two tables. The following query uses the "manages" relationship to obtain the department name, budget, and employee number and name of the department's manager for all departments that have a manager.

```
SELECT deptName, deptBudget, deptManager, empName
FROM Employee INNER JOIN Department
     ON Employee.empNo = Department.deptManager
ORDER BY deptName
```

The result table is follows. It has only three rows, one for each department row.

deptName	deptBudget	deptManager	empName
Accounting	100000	3	Olson, Jane
Marketing	220000	4	Neff, Arnold
Operations	400000	6	Naumi, Susan

Since all departments have managers, the `INNER JOIN` and the `RIGHT OUTER JOIN` of the above query will produce the same result. However, not all employees manage a department. Hence the `INNER JOIN` and `LEFT OUTER JOIN` produce different results. The latter includes all employees, not just those that manage departments. Rows for employees that do not manage departments contain null values for the columns, `deptName`, `deptBudget` and `deptManager` as illustrated below.

```
SELECT deptName, deptBudget, deptManager, empName
FROM Employee LEFT OUTER JOIN Department
     ON Employee.empNo = Department.deptManager
ORDER BY deptName
```

The result table is follows. It has one row for each employee.

deptName	deptBudget	deptManager	empName
			Young, John
			Homes, Denise
			Jones, David
			Smith, Joseph
Accounting	100000	3	Olson, Jane
Marketing	220000	4	Neff, Arnold
Operations	400000	6	Naumi, Susan

A JOIN query can include as many tables as necessary to obtain the needed data, each having its own join specification. Furthermore, a JOIN query can reference the same table more than once, provided it makes sense to do so. Given a set of tables, like Employee and Department, having two relationships, a JOIN query can combine rows from these tables using one relationship and then combine rows from that result (table) using the other one. To illustrate this, consider the following query to obtain the employee number, name of each employee, the department number, name of the department for which that employee works, and the employee number and name of the manager of that department.

```
SELECT Emp.empNo, Emp.empName, Department.deptNo, deptName,
     Emp2.empNo, Emp2.empName
FROM (Department INNER JOIN Employee AS Emp
     ON Department.deptNo = Emp.deptNo)
     INNER JOIN Employee AS Emp2
     ON Department.deptManager = Emp2.empNo
ORDER BY Emp.empName
```

Note the use of the *aliases*, Emp and Emp2 specified after the keyword, AS when the Employee table is referenced. These differentiate the context for the columns, empNo and empName in the query. Emp.empNo and Emp.empName refer to the empNo and empName columns when Department and Employee are joined on

```
Department.deptNo = Emp.deptNo
```

since the alias, Emp, is defined in that join specification and used to qualify the join attribute deptNo in the Employee table. Emp2.empNo and Emp2.empName refer to the empNo and empName columns when that intermediate result table is joined with Employee on

```
Department.deptManager = Emp2.empNo
```

since the alias, Emp2, is defined in that join specification and used to qualify the join attribute empNo in the Employee table.

The result table is given as follows.

Emp.empNo	Emp.empName	deptNo	deptName	Emp2.empNo	Emp2.empName
5	Homes, Denise	1	Marketing	4	Neff, Arnold
2	Jones, David	2	Operations	6	Naumi, Susan
6	Naumi, Susan	2	Operations	6	Naumi, Susan
4	Neff, Arnold	1	Marketing	4	Neff, Arnold
3	Olson, Jane	3	Accounting	3	Olson, Jane
1	Smith, Joseph	3	Accounting	3	Olson, Jane
7	Young, John	3	Accounting	3	Olson, Jane

The equivalent SQL1 query is:

```
SELECT Emp.empNo, Emp.empName, Department.deptNo, deptName,
     Emp2.empNo, Emp2.empName
FROM Department, Employee AS Emp, Employee AS Emp2
WHERE  Department.deptNo = Emp.deptNo AND
     Department.deptManager = Emp2.empNo
ORDER BY Emp.empName
```

Any number of tables can be joined in a single query, provided appropriate join conditions exist. Relationships in a Data Model or Class Diagram should document relationships that must be represented by primary key-foreign key pairs in the database. These can be used to specify meaningful joins. For example, a database used to support sales order processing would likely have tables representing *customers*, *salespeople*, *orders*, *line items* and *products*. Customers would likely have relationships with the salespeople and with the orders. Orders would have a relationship with line items, each of which would have a relationship with products. The database would need to include a primary key - foreign key pair for each relationship. Producing picking tickets or invoices would require data from all five of these entities. Hence, the query to produce picking tickets would need to join all five tables.

Nested Select Queries

The purpose of a Nested `SELECT` query is to enable selection based on the result of data values that can only be specified as the result of another query. For example, suppose the users require a list of all employees who work for the same department as employee number 5. This could be done using two separate queries, the first to determine the department number of the department for which employee number 5 works and using that result in the second query; however this requires procedural programming to store the department number and use it in the second query. It can also be done using a cross product query and aliases (joining all employees to all other employees); however, this is extremely inefficient and quite confusing. Nested `SELECT` queries provide this capability directly.

The `WHERE` clause in a Nested `SELECT` query uses the result of another, so called, *nested query*, as its selection criteria. The `WHERE` clause for such a query is of one of the following forms,

```
WHERE <column expression > [NOT] = (<select one>)
WHERE <column expression> [NOT] IN (<select list>)
WHERE <column expression> [NOT] {= | < | > |>= | <=}
      {ALL | SOME | ANY} (<select list>)
WHERE [NOT] EXISTS (<select expression>)
```

Optional parameters are in square brackets. Required, choice parameters (i.e. choose one) are in curly brackets. `<select one>` is a `SELECT` query that results in a single value, i.e., one column from one row. `<select list>` is a `SELECT` query that results in a set of values, i.e., one column from multiple rows. `<select expression>` is a `SELECT` query including `SELECT` and `FROM` clauses and optionally a `WHERE` clause. The `WHERE` clause of a `<select expression>` may, itself specify a nested query.

For example, the following query results use the first form (<select one>) to produce the above mentioned result (the employee number, name and department number of all employees who work for the same department as the employee whose employee number is 5).

```
SELECT empNo, empName, deptNo
FROM Employee
WHERE deptNo = (SELECT deptNo
                FROM Employee
                WHERE empNo = 5);
```

The nested query,

```
SELECT deptNo
FROM Employee
WHERE empNo = 5
```

results in the single value, 1, the department number of the department for which employee number 5 works. The main query selects all employees who work for that department. The result table for the above Nested SELECT query is,

empNo	empName	DeptNo
4	Neff, Arnold	1
5	Homes, Denise	1

Of course, Homes is employee number 5. As mentioned above, the same result can be obtained using a cross product query and aliases as follows.

```
SELECT Emp.empNo, Emp.empName, Emp.deptNo
FROM Employee AS Emp INNER JOIN Employee AS Emp2
     ON Emp.deptNo = Emp2.deptNo
WHERE Emp2.empNo = 5
```

Suppose users require a list of the employee number, name, weekly salary and department number of all employees whose salary is greater than that of all employees in department 1. This requires the third form of nested queries as follows,

```
SELECT empNo, empName, weeklySalary, deptNo
FROM Employee
WHERE weeklySalary >  ALL (SELECT weeklySalary
                           FROM Employee
                           WHERE deptNo = 1);
```

It query result is as follows,

empNo	EmpName	WeeklySalary	deptNo
1	Smith, Joseph	4000	3
2	Jones, David	3200	2
3	Olson, Jane	3800	3
6	Naumi, Susan	3500	2
7	Young, John	3000	3

The nested query retrieves two records 2300 and 1400. Since all the other employees' weekly salaries are greater than both these numbers, the five remaining employees are retrieved in the main query.

The nested query does not need to reference the same table as the main query, it may reference any table in the database. Consider a query to obtain the employee number, name and department number of all employees who work for the department with the maximum budget (if more than one department has a budget equal to the maximum, the query will list all employees who work for any of those departments).

```
SELECT empNo, empName, deptNo
FROM Employee
WHERE deptNo IN (SELECT deptNo
                 FROM Department
                 WHERE deptBudget = (SELECT max(deptBudget)
                                     FROM Department));
```

This Nested `SELECT` query has a Nested `SELECT` query in the `WHERE` clause of its Nested `SELECT` query! The innermost query,

```
SELECT max(deptBudget)
FROM Department
```

results in the value 400000, the maximum department budget. The next outer query,

```
SELECT deptNo
FROM Department
WHERE deptBudget =(SELECT max(deptBudget)
                   FROM Department)
```

results in a list having one value, 2, the department number of the department having this budget. In this case there is only one such department. However, since it is possible for more than one department to have this same budget, this nested query results in a `<select list>` rather than a `<select value>`. The query result is as follows.

empNo	EmpName	deptNo
2	Jones, David	2
6	Naumi, Susan	2

Data Maintenance Queries

As discussed above, SQL has three types of data maintenance queries, `INSERT`, `UPDATE` and `DELETE`. `INSERT` queries add rows to a table. They can take one of two forms,

```
INSERT INTO <table name> (<column names>) VALUES (<column values>);

INSERT INTO <table1 name>
     SELECT <table2 column names>
     FROM <table2 name>
     WHERE <conditions>;
```

In the first form, column names are separated by commas, as are the corresponding column values. In the second form, values are obtained from an existing table. The second form is useful, for example, to archive old data, prior to deleting it from an operational database. The following `INSERT` query uses the first form to add a new row to the `Employee` table having the specified values.

```
INSERT INTO Employee (empNo, empName, ssn, exemptions,
     weeklySalary, deptNo) VALUES (8, "Loud, Isabel", "779-92-
     9929", 2, 3700, 2);
```

If the value specified for `deptNo` does not exist in the `deptNo` column of the `Department` table, this `INSERT` query would be rejected and an error message generated. To support complex transaction processing, SQL has `COMMIT` and `ROLLBACK` commands. After executing the above query, the insert is not permanently added to the database until a `COMMIT` command is executed. Alternately, the database can be restored to its state prior to the execution of the `INSERT` query by executing a `ROLLBACK` command. Their syntax is simply,

`COMMIT`; or **`ROLLBACK`**;

When a `COMMIT` command is executed, *all* changes made since the last `COMMIT` or `ROLLBACK` command was executed are made permanent. Changes cannot be rolled back after they have been committed. Similarly, if changes are rolled back, the queries must be re-run before they can be committed and made. Similarly, if changes are rolled back, the queries must be re-run before they can be committed and made permanent. Note that Microsoft Access does not support the `COMMIT` or `ROLLBACK` commands. Once a query is executed, the database is updated. It cannot be rolled-back to a prior state. For this reason it is crucial to make a backup copy of your database prior to testing any `UPDATE` query in Access.

An `UPDATE` query is used to modify the values of existing rows in a table. Its syntax is,

```
UPDATE <table name> SET <column name>  =  <expression>
     [,<column name>  =  <expression>]*
WHERE <conditions>
```

The `[,<column name>  =  <expression>]*` means that, additional column updates can be specified, each separated by a comma. For example, the following query increases the weekly salary of each employee in department 3 by 10%.

```
UPDATE Employee SET weeklySalary = weeklySalary * 1.1
WHERE deptNo =3;
```

Executing a `COMMIT` command would make this change permanent (it is made permanent when the query is executed in Access).

A `DELETE` query is used to remove rows from a table. Its syntax is,

```
DELETE FROM <table>
WHERE <conditions>
```

For example, the following query permanently deletes employee number 8.

```
DELETE FROM Employee
WHERE empNo = 8;
```

The following permanently deletes all employees who work for department 3

```
DELETE FROM Employee
WHERE deptNo = 3;
```

Again, executing a `COMMIT` command would make the deletions permanent (each is made permanent when the query is executed in Access). If cascade delete is specified for the `Employee` work for `Department` relationship (i.e., for the `deptNo` foreign key in the `Employee` table) and department 3 is deleted, then effect on the `Employee` table is the same as the above query and `COMMIT` command.

Bibliography

Codd, E. F. (1970) "A Relational Model for Large Shared Databanks", *Communications of the ACM*, Vol. 13, No. 6, pp. 377-390.

Kent, W. (1983) “A Simple Guide to Five Normal Forms in Relational Database Theory”, *Communications of the ACM*, Vol. 26, No. 2, pp. 120-125.