SQL

David Klappholz
Department of Computer Science
Stevens Institute of Technology

1. Introduction

SQL is the most widely used query language for relational databases; it has gone through two standardizations, with a third in progress, and exists in numerous implementations/dialects which go well beyond the standards. Because of the proliferation of SQL's dialects, we will concentrate here on the basics common to all or most versions of the language rather than on a review of the the numerous implementations. Our approach will be more tutorial than exhaustive. We will elaborate on those aspects of the language which are the most powerful, and, therefore, the most interesting, and will simply make mention of some aspects of the language which, though they may be critically necessary in practice, are fundamentally simple, easy to learn, and not particularly interesting.

Because SQL is closely tailored to the relational approach to database management, an understanding of SQL requires at least an elementary understanding of that approach and of its historical motivation. The two major approaches which prevailed before the invention of the relational approach, the *hierarchical* approach and the *network* approach are both relatively cumbersome; both render the writing of query code relatively difficult as they require the programmer to deal directly with the complex data structures used to store the contents of a database. The relational approach to data storage and retrieval was proposed by E.F. Codd in 1970 to alleviate precisely this problem. The success of the relational approach, which currently dominates the market for database software, is due, at least in part, to the development of SQL, and to the development of techniques for the optimization of SQL code.

Codd's genius, as it is now viewed, was in proposing that data be stored in very simple data structures (*relations*), in showing how relations can be conceptually linked to one another in very simple ways (*foreign key references*), and, finally, in proposing a very simple yet powerful language for retrieving data (*relational algebra*). We will begin our discussion of the relational approach, in Section 1.1, by considering the details of Codd's first major contribution, the notions of:

- a relation scheme
- a relational database scheme
- a relational database
- a candidate key for a relation scheme
- a foreign key reference from one relation scheme to another

Since SQL is directly derivative of Codd's second major contribution, relational algebra, we will devote a significant part of this introductory section to a detailed consideration of that language, of its expressive power, and of the limits to its expressive power which constituted the motivation for the development of SQL (Section 1.2).

1.1 Relational Database Fundamentals

Codd proposed that data be stored as *relations*: sets of ordered tuples of values, with values of positionally-identical components in different tuples of the same relation all coming from the same *domain*. For reasons which we need not go into here, Codd soon adopted the convention that domains be sets of *atomic* values; i.e., neither were complex values with sub-parts of different types to be allowed as components of tuples, nor were sets with members of a single type.

As it turns out, it is simpler to view Codd's relations as *tables* with named columns -- column names being referred to as *attributes* -- than to think of them as relations, and it is simpler to think of tuples as *rows* of such tables. In this view, the predominant one among practitioners, a *domain* is simply an elementary datatype such as *integer*, *floating point*, *character*, *string*, etc. For the duration of our discussion of Codd's contributions, we will assume, with Codd, that a table (relation) is a *set* of rows (tuples), i.e., that no table has two or more identical rows.

1.1.1 Database/Table Schemes and Instances
Since concrete examples are already called for, and will become increasingly necessary as we proceed through our further discussion of the relational approach, we point the reader to Figure 1.1 wherein is shown the *relational database scheme* (type declaration) for a database designed to hold information about American manufacturing companies. (As the relational scheme of a "real" database is typically quite large, the example shown in Figure 1.1 is that of a "toy" version of a manufacturing-industry database; the example has, however, been designed to illustrate all the ideas which are relevant to the subsequent discussion.)

A *relational database scheme* consists of individual *relation schemes* (type declarations), *table schemes* in the terminology which we will be using here; in the case of Figure 1.1 the schemes are for tables named *company*, *division*, *site*, *product*, *person*, *works_for*, and *skill*.

company(co_name, govt_id, ceo_ssn, hq_loc)
division(co_id, div_name, subdiv_of, dir_ssn, div_hq)
site(co_id, div_name, loc)
product(prod_id, manuf_co, manuf_div, loc, prod_descr)
person(ssn, name, address)
works_for(ssn, co_id, div_name, salary, emp_id, sup_ssn)
skill(ssn, prod_id, manuf_co)

Figure 1.1

These schemes specify that instances of the tables will hold information (attribute values) of the following sorts:

•Each row of the *company* table will hold:

•co_name: the name of a company
•govt_id: the tax identification number assigned to the company by the federal government
•ceo_ssn: the social security number of the company's chief executive officer
•hq_loc: the name of the city in which the company's headquarters are located

•Each row of the *division* table will hold:

•co_id: the tax identification number assigned to a company by the federal government
•div_name: the name of a division of the company

- •subdiv_of: the name of that division of the company of which this division is a direct subdivision. The value of *subdiv_of* will be *null* if the division is a first-level subdivision, i.e., not a subdivision of any division. Note that the single *co_id* attribute serves as the company identifier for both the division (*div_name*) and the super-division (*subdiv_of*), a fact which implies that a division of one company cannot be a subdivision of a division of a different company.
- •dir_ssn: the social security number of the division's director
- •div_hq: the city in which the division's headquarters are located

•Each row of the *site* table will hold:

- •co_id: the tax identification number assigned to a company by the federal government
- •div_name: the name of a division of the company
- •loc: the name of a city in which a site (installation) of the division of the company is located. (If we wished to distinguish among different installations located in the same city, then we would add, say, an *address* attribute.)

•Each row of the *product* table will hold:

- •manuf_co: the tax identification number assigned to a company by the federal government
- •prod_id: the company-assigned identifier of a product that the company manufactures
- •manuf_div: the division of the company by which the product is manufactured
- •loc: the location (site) of the division of the company at which the product is manufactured
- •prod_descr: the company-assigned description of the product

•Each row of the *person* table will hold:

- •ssn: a social security number
- •name: the name of the person with that social security number
- •address: the address of the person with that social security number

•Each row of the *works_for* table will hold:

- •ssn: a social security number
- •co_id: the tax identification number assigned, by the federal government, to a company which employs the person with that social security number
- •div_name: the name of the division of the company to which the person is assigned
- •salary: the person's yearly salary from the company
- •emp_id: the identification number assigned to the person by the company
- •sup_ssn: the social security number of the person's direct supervisor in the company

•Each row of the *skill* table will hold:

- •manuf_co: the tax identification number assigned to a company by the federal government
- •prod_id: the identifier of a product manufactured by the company

•ssn: the social security number of a person trained in the manufacture of the product for the company

A *relational database* is an *instance* of a relational database scheme which, in turn, is a collection of *instances* of the database scheme's table schemes -- one of each. A sample (partial) instance of the manufacturing company database scheme is shown in Figures 1.2a-g.

co_name	govt_id	ceo_ssn	hq_loc
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	999-00-0002	Trenton
Southern Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Ohio Windshield	888-99-0000	999-00-0007	Toledo
⋮	⋮	⋮	⋮

***company* table**
Figure 1.2a

co_id	div_name	subdiv_of	dir_ssn	div_hq
666-77-8888	innovation	null	888-00-0001	Valdosta
666-77-8888	research	innovation	888-00-0001	Boston
666-77-8888	development	innovation	888-00-0002	Nashville
333-44-5555	personnel	null	888-00-0003	New York
333-44-5555	manufacturing	null	888-00-0005	Boston
333-44-5555	axle	manufacturing	888-00-0006	Cambridge
333-44-5555	front axle	axle	888-00-0007	Mattapan
333-44-5555	rear axle	axle	888-00-0008	Roxbury
444-55-6666	communications	null	888-00-0010	West Chester
444-55-6666	purchasing	null	888-00-0011	Bensalem
⋮	⋮	⋮	⋮	⋮

***division* table**
Figure 1.2b

co_id	div_name	loc
333-44-5555	front axle	Atlanta
333-44-5555	front axle	Valdosta
333-44-5555	front axle	Memphis
333-44-5555	manufacturing	Cleveland
333-44-5555	manufacturing	Nashville
333-44-5555	manufacturing	New Orleans
888-99-0000	purchasing	Toledo
888-99-0000	communications	Columbus
888-99-0000	communications	Dayton
⋮	⋮	⋮

***site* table**
Figure 1.2c

product_id	manuf_co	manuf_div	loc	prod_descr
FA-S/1	333-44-5555	front axle	Valdosta	small light-weight front axle
FA-S/2	333-44-5555	front axle	Valdosta	small heavy weight front axle
FA-L/1	333-44-5555	front axle	Atlanta	large light-weight rust-proof front axle
FA-L/2	333-44-5555	front axle	Atlanta	large light-weight plain front axle
FA-L/3	333-44-5555	front axle	Atlanta	large medium-weight rust-proof front axle
⋮	⋮	⋮	⋮	⋮

***product* table**
Figure 1.2d

ssn	name	address
999-00-0001	Joseph Jones	76 Main Street, Atlanta, GA
999-00-0002	Daniel Smith	42 Main Street, Atlanta, GA
999-00-0003	Donald Smith	76 Main Street, Atlanta, GA
999-00-0004	Denis Smith	90 Main Street, Atlanta, GA
777-00-0001	Jehosaphat Jones	32 Main Street, Atlanta, GA
777-00-0002	Jennifer Jones	29 Main Street, Atlanta, GA
777-00-0003	Janet Jones	87 Main Street, Atlanta, GA
777-00-0004	Janice Jones	63 Main Street, Atlanta, GA
777-00-0005	Joline Jones	29 Main Street, Atlanta, GA
777-00-0006	Jud Jones	29 Main Street, Atlanta, GA
777-00-0007	Jerusha Jones	29 Main Street, Atlanta, GA
⋮	⋮	⋮

person **table**
Figure 1.2e

ssn	co_id	div_name	salary	emp_id	sup_ssn
777-00-001	333-44-5555	front axle	51000	EMP/FA-1	888-00-0004
777-00-002	333-44-5555	front axle	52000	EMP/FA-2	888-00-0004
777-00-003	333-44-5555	front axle	53000	EMP/FA-3	888-00-0004
777-00-004	333-44-5555	front axle	54000	EMP/FA-4	888-00-0004
777-00-005	333-44-5555	null	600000	EMP/1	null
777-00-006	333-44-5555	front axle	56000	EMP/FA-6	888-00-0004
⋮	⋮	⋮	⋮	⋮	⋮

works_for **table**
Figure 1.2f

ssn	prod_id	manuf_co
777-00-0005	FA-S/1	333-44-5555
777-00-0005	FA-S/2	333-44-5555
777-00-0005	FA-L/1	333-44-5555
777-00-0005	FA-L/2	333-44-5555
777-00-0006	FA-L/1	333-44-5555
777-00-0006	FA-L/2	333-44-5555
777-00-0006	FA-L/3	333-44-5555
⋮	⋮	⋮

skill **table**
Figure 1.2g

While we have indicated that a relational database's scheme may be thought of as its type declaration and that the individual schemes for the database's tables may be thought of as the type declarations for those tables, we must note one proviso: an instance of a database scheme consists of exactly *one* instance of each of its relation schemes whereas in the context of programming languages, *multiple* instances of a user-declared type are always allowed.

1.1.2 Keys and Their Implications

In the listings of the attributes of the various tables of the manufacturing-company database we have used the articles *a*, *an*, and *the* in such way as to indicate that certain items must be unique and that others need not be; for example, in indicating that attribute *co_name* of *company* will hold *the* name of a company, we have implied that a company cannot have more than one name.

While this particular detail may be obvious, both it and others, some of them anything but obvious, must be formally specified as part of a database's/table's scheme. Details of this sort are specified by the database designer through the choice of *candidate keys* -- often referred to simply as *keys* -- and *foreign key* references. A key is specified in a database scheme through the use of underlining as in Figure 1.1; thus, for example, the fact that *govt_id* is underlined in the scheme for *company* records the database designer's designation of *govt_id* as the key for *company*. Note that a key may consist of more than one attribute, e.g., {*co_id*, *div_name*} of *division*; a key may even consist of *all* the attributes of a scheme, e.g., {*ssn*, *prod_id*, *manuf_co*} of *skill*. Finally, a table scheme may have more than one key, in which case the database designer often chooses one as the *primary key*. (In fact, in the notation for database schemes used in Figure 1.1, if a table has multiple keys it is possible to indicate only the primary key.)

The designation, by a database designer, of a set of attributes as a key for a table constitutes the imposition, by the designer, of the constraint that an instance of the table is not to be deemed *legal* if any two rows of the instance agree on the values of these attributes. Thus, for example, the designation of *govt_id* as the key for *company* represents the imposition of the constraint that an instance of *company* will not be deemed legal if two rows have the same *govt_id* value; the fact that *co_name* was not designated as part of the key represents the lack of any such constraint on that attribute; i.e., the designer has decided that different companies will be allowed to have the same name. The designation of {*co_id*, *div_name*} as the key for *division* represents the imposition of the constraint that an instance of *division* will not be deemed legal if two rows agree on the values of both the *co_id* attribute and the *div_name* attribute. (There would be no problem however if two rows agreed on one of the two attributes but not the other, i.e., if a company had two or more differently-named divisions and/or if different companies had divisions with the same name.)

An alternative way of interpreting the key constraint on *company* would be to say that a company *is uniquely identified by* its *govt_id*; but note that the phrase "is uniquely identified by" suggests an assertion about the real world, whereas the designation of a key represents only the database designer's assertion of a constraint on instances of tables, i.e., an aspect of the designer's decision as to how the database is to *model* the real world.

The implications, both obvious and non-obvious of the choices of keys for the manufacturing-industry database are as follows:

- in the case of *company(co_name, <u>govt_id</u>, ceo_ssn, hq_loc)*:
    - a company <u>cannot</u> have more than one name
    - different companies <u>can</u> have the same name

•a company cannot have more than one tax identification number
•different companies cannot have the same tax identification number
•a company cannot have more than one chief executive officer
•different companies can have the same person as chief executive officer
•a company cannot have more than one headquarters location
•different companies can have the same headquarters location

•in the case of *division(co_id, div_name, subdiv_of, dir_ssn, div_hq)*:
•a company cannot have different divisions with the same name
•different companies can have divisions with identical names
•A division of a company cannot be a subdivision of a division of a different company.
•a division can be a subdivision of itself (If we wish to rule this out, we must do so in some fashion other than through the designation of key constraints.)
•a division of a company cannot be a (direct) subdivision of more than one division
•different divisions of a company can be (direct) subdivisions of the same division
•a person can be the director of more than one division of a company
•a person can be the director of divisions of different companies
•a division of a company cannot have more than one division headquarters city
•different divisions of a company can have the same division headquarters city
•divisions of different companies can have the same division headquarters city

•in the case of *site(co_id, div_name, loc)*:
•different divisions of a company can have sites in the same city
•divisions of different companies can have sites in the same city

•in the case of *product(prod_id, manuf_co, manuf_div, loc, prod_descr)*:
•different companies can manufacture products with the same product identifier
•if different companies manufacture products with the same product identifier, those products can be manufactured by divisions with the same name
•if different companies manufacture products with the same product identifier, those products can be manufactured by divisions with different names
•if different companies manufacture products with the same product identifier, those products can have identical descriptions
•if different companies manufacture products with the same product identifier, those products can have different descriptions
•if a product is manufactured by a division of a company at a specific site of that division of the company, then it cannot be manufactured at any other of the division's sites.

•in the case of *person(ssn, name, address)*:
•different people cannot have the same social security number
•different people can have the same name
•different people can have the same address

•in the case of *works_for(ssn, co_id, div_name, salary, emp_id, sup_ssn)*:
•a person can work for more than one company. (If we wished to rule out this possibility, we could do so by designating just *ssn* as the key of *works_for*.)
•a company can have more than one person working for it

- a person cannot work for more than one division of a company
- a person cannot have more than one salary from a company
- a person can have the same salary from different companies
- a person can have different salaries from different companies
- a person cannot have more than one employee identifier from a company
- a person can have the same employee identifier from different companies
- a person can have different employee identifiers from different companies
- a person cannot have more than one supervisor in a company. (Note that the choice of key for *works_for* does not imply that an employee's supervisor need work for the same division of the company, nor that the supervisor even work for the company; nor can such a restriction be imposed on the scheme of Figure 1.1 through the use of foreign key references. If such a restriction is desired, then it must be imposed in some other fashion.)
- a person can have the same supervisor in different companies
- a person can have different supervisors in different companies

- in the case of *skill(ssn, prod_id, manuf_co)*:
  - a person can be trained in the manufacture of different products for the same company
  - a person can be trained in the manufacture of products with the same product identifiers for different companies
  - a person can be trained in the manufacture of products with different product identifiers for different companies
  - different people can be trained in the manufacture of a product for a company

1.1.3 Foreign Key References and Their Implications

As indicated at the beginning of this introductory section, Codd proposed the notion of a *foreign key reference* to enable the database designer to specify conceptual connections between tables. By way of introducing the notion of a *foreign key reference*, we point the reader to the *govt_id* attribute of *company* and the *co_id* attribute of *division*. The reader will recall that the *govt_id* attribute of *company* is intended to hold a company's government-assigned tax identification number, a company's unique identification insofar as the manufacturing-industry database is concerned. The database designer has also specified that *co_id* in *division* hold the identifier of a company; i.e., *division*'s *co_id* attribute will hold government-assigned tax identification numbers.

The database designer must, of course, communicate to the direct users of the database the fact that an attribute of one table is "the same as" a, possibly same-named, possibly differently-named, attribute of a different table[1]. Without such information, the application programmer would be unable to write code for queries which require reference to tables which are related in this way[2].

Codd's terminology for the same-ness of two sets of attributes is that one is a *foreign key reference* to the other; in the case of the example which we have been pursuing, *co_id* of *division* would be a foreign key reference to *govt_id* of *company*, rather that the other way around, because *govt_id* of *company* is its table's key and *co_id* of *division* is not its table's key. The direction of a foreign key reference is always *to* a key because of the

[1]or of a differently-named attribute of the same table.

[2]An example of such a query is "Find every company which has its heaquarters in Atlanta and has a division with divisional headquarters in Boston," a query which requires reference to the *company* and *division* tables and which requires awareness of the same-ness of *govt_id* in the former and *co_id* in the latter.

implication, discussed in the immediately following paragraph, of a foreign key reference as a constraint on legal instances of database tables.

Just as the database designer's choice of a key for a table scheme amounts to the imposition of a constraint on what constitutes a legal instance of that table, the choice of two sets of attributes from two different tables and the assertion of a foreign key reference from one of the sets of attributes to the other constitutes the designer's imposition of a constraint on the legality of table instances. In the example which we have been considering, the constraint is that a *division* table instance is not deemed to be legal if it contains a row with a *co_id* value which is not also the *govt_id* value of some row in the *company* table; i.e., a *division* table may not contain information about a division of a company about which the database has no *company*-table information.

The foreign key references for the manufacturing-industry database scheme are shown in Figure 1.3, in which the directions of the arrows indicate the directions of the references. (One foreign key reference, a reference which is a bit difficult to represent in the notation of Figure 1.3, has been omitted, but is, nevertheless, intended to hold; that reference is from {*co_id*, *subdiv_of*} of *division* to {*co_id*, *div_name*} of *division*; i.e., the database is not to be allowed to hold information about one division's being a super-division of another, unless the super-division is recorded in the database as being a division of the same company as the sub-division.) Note that just as a key need not be a single attribute, a foreign key reference may be from a non-singleton set of attributes to another set of attributes of the same cardinality.

Having discussed Codd's first major contribution, the invention of the notion of a relational database and the definition of its basic components, it is time to proceed to a discussion of Codd's second major contribution, relational algebra. Before doing so, we consider the more general subject of querying a relational database.

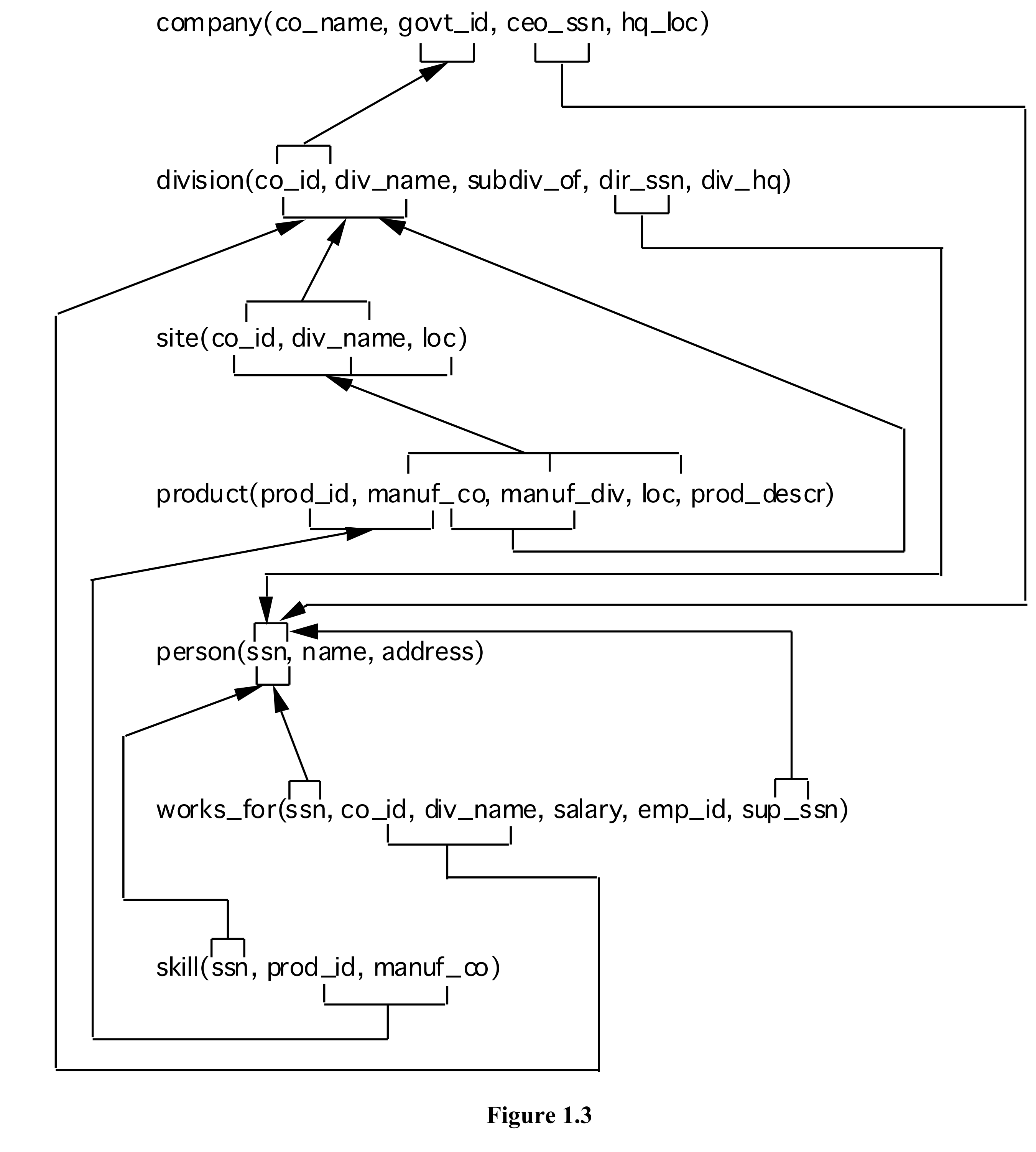

company(co_name, govt_id, ceo_ssn, hq_loc)
division(co_id, div_name, subdiv_of, dir_ssn, div_hq)
site(co_id, div_name, loc)
product(prod_id, manuf_co, manuf_div, loc, prod_descr)
person(ssn, name, address)
works_for(ssn, co_id, div_name, salary, emp_id, sup_ssn)
skill(ssn, prod_id, manuf_co)

Figure 1.3

1.2 Introduction to Querying a Relational database

Given that a relational database consists of multiple tables, it should come as no surprise that the retrieval of information from such a database sometimes requires reference to a single table and sometimes requires reference to multiple tables. (In what follows we will use the word *table*, rather than the phrase *table scheme* or the phrase *table instance* when no confusion is expected to result; in the present case, of course, we have in mind *table instance*; schemes cannot be queried.) Consider as examples the queries of Figures 1.4a-g. The first query requires reference to only *company*; the second requires reference to *company* and *division*; each of the subsequent queries requires reference to one more table than does the previous one so that the seventh requires reference to all seven tables.

Find every company with headquarters in Atlanta.

Figure 1.4a

Find every company with headquarters in Atlanta which has a division whose divisional headquarters are in Boston.

Figure 1.4b

Find every company with headquarters in Atlanta which has a division whose divisional headquarters are in Boston and and which division has a site in Cleveland.

Figure 1.4c

Find every company with headquarters in Atlanta which has a division whose divisional headquarters are in Boston and which division has a site in Cleveland, and which division manufactures a product whose id is *FA-S/2*.

Figure 1.4d

Find every company with headquarters in Atlanta whose chief executive officer is named Joline Jones, and which has a division whose divisional headquarters are in Boston and which division has a site in Cleveland, and which division manufactures a product whose id is *FA-S/2*.

Figure 1.4e

Find every company with headquarters in Atlanta whose chief executive officer, Joline Jones, earns more than $500,000 per year for her work for the company, and which has a division whose divisional headquarters are in Boston and which division has a site in Cleveland, and which division manufactures a product whose id is *FA-S/2*.

Figure 1.4f

Find every company with headquarters in Atlanta whose chief executive officer, Joline Jones, earns more than $500,000 per year for her work with the company, and which has a division whose divisional headquarters are in Boston and which division has a site in Cleveland, and which division manufactures a product whose id is *FA-S/2* in whose manufacture the chief executive officer happens to be trained.

Figure 1.4g

1.2.1 Nested-Loop Queries

Single-table queries can usually be implemented as single-level loops, and multiple-table queries can often be implemented as nested loops, the level of nesting often being identical to the number of tables involved.

The pseudo-code loop nests of Figures 1.5a-g are potential implementations of the queries of Figures 1.4a-g respectively. Note that in Figures 1.5b thru 1.5g, each simple condition in the *if* statement tests for an attribute value's being equal to a scalar value mentioned in the English query, for two rows' being related by a foreign key reference, or for two rows' being related by an *indirect* foreign key reference.

```
for i = 1 to no_rows(company) do
    if company[i][hq_loc] = "Atlanta"
    then output(company[i][govt_id])
```

Figure 1.5a

```
for i = 1 to no_rows(company) do
        for j = 1 to no_rows(division) do
            if (company[i][hq_loc] = "Atlanta")
                and (division[j][div_hq] = "Boston)
                and (division[j][co_id] = company[i][govt_id])
            then output(company[i][govt_id])
```

Figure 1.5b

```
for i = 1 to no_rows(company) do
    for j = 1 to no_rows(division) do
        for k = 1 to no_rows(site) do
            if (company[i][hq_loc] = "Atlanta")
                and (division[j][div_hq] = "Boston)
                and (division[j][co_id] = company[i][govt_id])
                and (site[k][loc] = "Cleveland")
                and (site[k][co_id] = division[j][co_id])
                and (site[k][div_name] = division[j][div_name])
            then output(company[i][govt_id])
```

Figure 1.5c

```
for i = 1 to no_rows(company) do
    for j = 1 to no_rows(division) do
        for k = 1 to no_rows(site) do
            for m = 1 to no_rows(product) do
                if (company[i][hq_loc] = "Atlanta")
                    and (division[j][div_hq] = "Boston)
                    and (division[j][co_id] = company[i][govt_id])
                    and (site[k][loc] = "Cleveland")
                    and (site[k][co_id] = division[j][co_id])
                    and (site[k][div_name] = division[j][div_name])
                    and (product[m][prod_id] = "FA-S/2")
                    and (product[m][manuf_co] = site[k][co_id])
                    and (product[m][manuf_div] = site[k][div_name])
                then output(company[i][govt_id])
```

Figure 1.5d

```
for i = 1 to no_rows(company) do
    for j = 1 to no_rows(division) do
        for k = 1 to no_rows(site) do
            for m = 1 to no_rows(product) do
                for n = 1 to no_rows(person) do
                    if (company[i][hq_loc] = "Atlanta")
                        and (division[j][div_hq] = "Boston)
                        and (division[j][co_id] = company[i][govt_id])
                        and (site[k][loc] = "Cleveland")
                        and (site[k][co_id] = division[j][co_id])
                        and (site[k][div_name] = division[j][div_name])
                        and (product[m][prod_id] = "FA-S/2")
                        and (product[m][manuf_co] = site[k][co_id])
                        and (product[m][manuf_div] = site[k][div_name])
                        and (person[n][name] = "Joline Jones")
                        and (person[n][ssn] = company[i][ceo_ssn])
                    then output(company[i][govt_id])
```

Figure 1.5e

```
for i = 1 to no_rows(company) do
    for j = 1 to no_rows(division) do
        for k = 1 to no_rows(site) do
            for m = 1 to no_rows(product) do
                for n = 1 to no_rows(person) do
                    for p = 1 to no_rows(works_for) do
                        if (company[i][hq_loc] = "Atlanta")
                            and (division[j][div_hq] = "Boston)
                            and (division[j][co_id] = company[i][govt_id])
                            and (site[k][loc] = "Cleveland")
                            and (site[k][co_id] = division[j][co_id])
                            and (site[k][div_name] = division[j][div_name])
                            and (product[m][prod_id] = "FA-S/2")
                            and (product[m][manuf_co] = site[k][co_id])
                            and (product[m][manuf_div] = site[k][div_name])
                            and (person[n][name] = "Joline Jones")
                            and (person[n][ssn] = company[i][ceo_ssn])
                            and (works_for[p][salary] > 500000)
                            and (works_for[p][ssn]  = person[n][ssn])
                            and (works_for[p][co_id]  = company[i][govt_id])
                        then output(company[i][govt_id])
```

Figure 1.5f

```
for i = 1 to no_rows(company) do
    for j = 1 to no_rows(division) do
        for k = 1 to no_rows(site) do
            for m = 1 to no_rows(product) do
                for n = 1 to no_rows(person) do
                    for p = 1 to no_rows(works_for) do
                        for q = 1 to no_rows(skill) do
                            if (company[i][hq_loc] = "Atlanta")
                                and (division[j][div_hq] = "Boston)
                                and (division[j][co_id] = company[i][govt_id])
                                and (site[k][loc] = "Cleveland")
                                and (site[k][co_id] = division[j][co_id])
                                and (site[k][div_name] = division[j][div_name])
                                and (product[m][prod_id] = "FA-S/2")
                                and (product[m][manuf_co] = site[k][co_id])
                                and (product[m][manuf_div] = site[k][div_name])
                                and (person[n][name] = "Joline Jones")
                                and (person[n][ssn] = company[i][ceo_ssn])
                                and (works_for[p][salary] > 500000)
                                and (works_for[p][ssn]  = person[n][ssn])
                                and (works_for[p][co_id]  = company[i][govt_id])
                                and (skill[q][prod_id] = "FA-S/2")
                                and (skill[q][ssn] = person[n][ssn])
                                and (skill[q][manuf_co] = company[i][govt_id])
                            then output(company[i][govt_id])
```

Figure 1.5g

In all of the above example queries it was rows of *different* tables that were compared to one another. In cases in which rows of the *same* table must be compared with one another, the depth of loop nesting required is usually incremented by one for each additional row which must be examined. For example, the query of Figure 1.6a requires that pairs of rows of *skill* be examined, a fact which is reflected in the two-level loop nest in its potential implementation in Figure 1.6b.

Find every person who is trained in the manufacture of at least two different products for the company whose government id is 111-22-3333.

Figure 1.6a

```
for i = 1 to no_rows(skill) do
    if (skill[i][co_id] = "111-22-3333") then
        for j = 1 to no_rows(skill) do
            if (skill[j][co_id] = "111-22-3333")
                and (skill[i][ssn] = skill[j][ssn])
                and (skill[i][prod_id] <> skill[j][prod_id])
            then output(skill[i][ssn])
```

Figure 1.6b

Codd, realizing that the expression of queries in a highly stylized algebraic language would be far more likely to lead to the development of methods for query optimization than their expression in a general-purpose programming language, invented just such a language, *relational algebra*.

1.2.2 Basics of Relational Algebra

While relational algebra is, indeed, a query language, it has never been standardized and is not directly available to the user of any serious commercial database software product. Justification for its consideration here is two-fold: first, from a pedagogical point of view, an understanding of the fundamentals of relational algebra renders SQL much easier to learn; second, many commercial database products translate SQL into (proprietarily modified/extended) relational algebra as a first step in the query optimization process, making an understanding of relational algebra critical to an understanding of that process.

Relational algebra is a system of expressions, all of whose operators take tables as operands and produce tables as results (values). The basic operators of relational algebra are:

- *select*, denoted σ, a unary operator which takes the place of the *if* in the nested-loop query implementations of Figures 1.5a-g
- *project*, denoted π, a unary operator which, roughly speaking, takes the place of *output()* in the nested-loop query implementations of Figures 1.5a-g
- *cartesian product*, denoted $\times$, a binary operator, familiar from set theory, which takes the place of loop nesting in the nested-loop query implementations of Figures 1.5b-g
- *rename*, denoted ρ, a unary operator which renames a table, a technical necessity in relational algebra which doesn't have a parallel in the nested-loop query implementations of Figures 1.5a-g
- the binary set operators $\cup$, $\cap$, and - (set union, set intersection, and set difference)

1.2.2.1 The *select* Operator

The *select* operator returns the table which results from deleting from its operand all rows which do not satisfy the subscripted condition (see Figure 1.7 which shows the use of the *select* operator in expressing the query of Figure 1.4a.)

$\sigma_{hq\_loc = "Atlanta"}(company)$

Figure 1.7

The relational algebra expression of Figure 1.7 has as its value (i.e., it returns) a table with four columns, the columns of *company*. It contains all and only those rows of the instance of *company* shown in Figure 1.2a which have the value "Atlanta" in their *hq_loc* attributes as shown in Figure 1.8.

co_name	**govt_id**	**ceo_ssn**	**hq_loc**
Southern Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Georgia Brake	666-77-8888	999-00-0005	Atlanta
⋮	⋮	⋮	⋮

Figure 1.8

1.2.2.2 The *project* Operator
If, on the other hand, the result desired is a single-column table of *govt_id*'s (of companies headquartered in Atlanta), then we apply the *project* operator to the result of the *select* operator. The *project* operator returns the result of deleting from its operand all columns except those listed in the subscripted attribute list. The relational algebra expression of Figure 1.9 produces precisely the desired result; that part of the result which can be gleaned from the tables of Figures 1.2a-1.2g is shown in Figure 1.10.

$\pi_{govt\_id}(\sigma_{hq\_loc = "Atlanta"}(company))$

Figure 1.9

govt_id
333-44-5555
444-55-6666
666-77-8888
⋮

Figure 1.10

1.2.2.3 The *cartesian product* Operator
When a query requires reference to more than one table, the *cartesian product* operator is used. The cartesian product operator takes two tables, T1 and T2, and produces a table each of whose rows consists of a row of T1 concatenated with a row of T2; *cartesian product* produces all possible such rows. *Cartesian product* can be used to write relational algebra versions of the queries of Figures 1.4b-g as we will see presently; The query of

Figure 1.4b requires a two-way cartesian product, that of Figure 1.4c a three-way cartesian product, ... , and that of Figure 1.4g a seven-way cartesian product. Even the two-way example results in a table containing eighty rows, making it too bulky to show in its entirety; the seven-way example results in a table with

no_rows(company)*no_rows(division)*no_rows(site)*no_rows(product)
*no_rows(person)*no_rows(works_for)*no_rows(skill)

= 8*10*9*5*11*6*7 = 1,663,200

rows. While this large a table can easily be managed by database software, it is implausibly large to be shown here. We will, therefore, illustrate the results of *cartesian product*, i.e., the entire table produced by a *cartesian product*, through the use of a small free-standing example. Consider, therefore, the tables of Figures 1.11 and 1.12; the result of computing their cartesian product is the table of Figure 1.13.

ssn	name
000-00-0000	John Smith
000-00-0001	Jane Smith
000-00-0002	Jodie Smith

Figure 1.11

author	isbn	title
000-00-0000	67543	"Stuff"
000-00-0000	98763	"Things"
000-00-0001	84213456	"Things & Stuff"
000-00-0001	983215	"Stuff & Things"

Figure 1.12

ssn	name	author	isbn	title
000-00-0000	John Smith	000-00-0000	67543	"Stuff"
000-00-0000	John Smith	000-00-0000	98763	"Things"
000-00-0000	John Smith	000-00-0001	84213456	"Things & Stuff"
000-00-0000	John Smith	000-00-0001	983215	"Stuff & Things"
000-00-0001	Jane Smith	000-00-0000	67543	"Stuff"
000-00-0001	Jane Smith	000-00-0000	98763	"Things"
000-00-0001	Jane Smith	000-00-0001	84213456	"Things & Stuff"
000-00-0001	Jane Smith	000-00-0001	983215	"Stuff & Things"
000-00-0002	Jodie Smith	000-00-0000	67543	"Stuff"
000-00-0002	Jodie Smith	000-00-0000	98763	"Things"
000-00-0002	Jodie Smith	000-00-0001	84213456	"Things & Stuff"
000-00-0002	Jodie Smith	000-00-0001	983215	"Stuff & Things"

Figure 1.13

As an example, of the use of *cartesian product* in writing relational algebra queries, consider Figure 1.14 which shows its use in expressing the query of Figure 1.4b.

$$\pi_{\text{company.govt\_id}}(\sigma_{\substack{\text{(company.govt\_id = division.co\_id)} \\ \text{and(company.hq\_loc = "Atlanta")} \\ \text{and(division.div\_hq = "Boston")}}} [\text{company} \times \text{division}])$$

Figure 1.14

In the relational algebra expression of Figure 1.14 we have utilized conventional record-field *dot* notation to indicate which table an attribute is associated with; though doing so is not actually required unless two relations which occur in an expression have same-named attributes, we will persist in the use of this notation in order to render relational algebra expressions easier to read. Figures 1.15-1.19 show relational algebra expressions for the queries of Figures 1.4c-g.

$$\pi_{\text{company.govt\_id}}(\sigma_{\substack{\text{(company.govt\_id = division.co\_id)} \\ \text{and(company.hq\_loc = "Atlanta")} \\ \text{and(division.div\_hq = "Boston")} \\ \text{and(site.co\_id = company.govt\_id)} \\ \text{and(site.div\_name = division.div\_name)} \\ \text{and(site.loc = "Cleveland")}}} [\text{company} \times \text{division} \times \text{site}])$$

Figure 1.15

$$\pi_{\text{company.govt\_id}}(\sigma_{\substack{\text{(company.govt\_id = division.co\_id)} \\ \text{and(company.hq\_loc = "Atlanta")} \\ \text{and(division.div\_hq = "Boston")} \\ \text{and(site.co\_id = company.govt\_id)} \\ \text{and(site.div\_name = division.div\_name)} \\ \text{and(site.loc = "Cleveland")} \\ \text{and(product.manuf\_co = company.govt\_id)} \\ \text{and(product.manuf.div = site.div\_name)} \\ \text{and(product.prod\_id = FA-S/2)}}} [\text{company} \times \text{division} \times \text{site} \times \text{product}])$$

Figure 1.16

$\pi_{company.govt\_id}(\sigma$ [company $\times$ division $\times$ site $\times$ product $\times$ person])

```
(company.govt_id = division.co_id)
  and(company.hq_loc = "Atlanta")
  and(division.div_hq = "Boston")
  and(site.co_id = company.govt_id)
  and(site.div_name = division.div_name)
  and(site.loc = "Cleveland")
  and(product.manuf_co = company.govt_id)
  and(product.manuf.div = site.div_name)
  and(product.prod_id = FA-S/2)
  and(person.ssn = company.ceo_ssn)
  and(person.name = "Joline Jones")
```

Figure 1.17

$\pi_{company.govt\_id}(\sigma$ [company $\times$ division $\times$ site $\times$ product $\times$ person $\times$ works_for])

```
(company.govt_id = division.co_id)
  and(company.hq_loc = "Atlanta")
  and(division.div_hq = "Boston")
  and(site.co_id = company.govt_id)
  and(site.div_name = division.div_name)
  and(site.loc = "Cleveland")
  and(product.manuf_co = company.govt_id)
  and(product.manuf.div = site.div_name)
  and(product.prod_id = FA-S/2)
  and(person.ssn = company.ceo_ssn)
  and(person.name = "Joline Jones")
  and(works_for.ssn = person.ssn)
  and(works_for,co_id = company.govt_id)
  and(works_for.salary > 500000)
```

Figure 1.18

$$\pi_{\text{company.govt\_id}}\left(\sigma_{\substack{\text{(company.govt\_id = division.co\_id)}\\ \text{and(company.hq\_loc = "Atlanta")}\\ \text{and(division.div\_hq = "Boston")}\\ \text{and(site.co\_id = company.govt\_id)}\\ \text{and(site.div\_name = division.div\_name)}\\ \text{and(site.loc = "Cleveland")}\\ \text{and(product.manuf\_co = company.govt\_id)}\\ \text{and(product.manuf\_div = site.div\_name)}\\ \text{and(product.prod\_id = FA-S/2)}\\ \text{and(person.ssn = company.ceo\_ssn)}\\ \text{and(person.name = "Joline Jones")}\\ \text{and(works\_for.ssn = person.ssn)}\\ \text{and(works\_for,co\_id = company.govt\_id)}\\ \text{and(works\_for.salary > 500000)}\\ \text{and(skill.ssn = person.ssn)}\\ \text{and(skill.prod\_id = product.prod\_id)}\\ \text{and(skill.manuf\_co = company.govt\_id)}}}\left[\text{company} \times \text{division} \times \text{site} \times \text{product} \times \text{person} \times \text{works\_for} \times \text{skill}\right]\right)$$

Figure 1.19

Note that since the relational algebra expressions of Figures 1.14-1.19 are relational algebra expressions for the queries of Figures 1.4a-g, and since the nested-loop programs of Figures 1.5a-g are potential implementations of the very same queries, the nested-loop programs may be used to evaluate the relational algebra expressions of Figures 1.14-1.19; i.e., cartesian products need not be computed; note, however, the the number of iterations of each of the nested-loop programs is exactly the same as the number of rows of the corresponding cartesian product so that while the nested-loop implementations save storage, the amount of time they require is still proportional to the product of the numbers of rows of all the tables involved.

1.2.2.4 The *rename* Operator

Cartesian product is also used, but with the aid of the *rename* operator, when different rows of the same table must be compared. What is actually done in such cases, however, is not literally the comparison of two different rows of the *same* table, but, rather, the comparison of rows from two renamed copies of the table. The *rename* operator assigns the name found in its subscript to its single operand. Thus, for example, the relational algebra expression of Figure 1.20 is an implementation of the query of Figure 1.6a. Figure 1.21 shows the cartesian product of the two renamed tables.

$$\pi_{\text{skill1.ssn}}\left(\sigma_{\substack{\text{(skill1.co\_id = 111-22-3333)}\\ \text{and(skill2.co\_id = skill1.co\_id)}\\ \text{and(skill1.ssn = skill2.ssn)}\\ \text{and(skill1.prod\_id <> skill2.prod\_id)}}}\left[\rho_{\text{skill1}}(\text{skill}) \times \rho_{\text{skill2}}(\text{skill})\right]\right)$$

Figure 1.20

skill1.ssn	**skill1. prod_id**	**skill1. manuf_co**	**skill2.ssn**	**skill2. prod_id**	**skill2. manuf_co**
777-00-0005	FA-S/1	333-44-5555	777-00-0005	FA-S/1	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0005	FA-S/2	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0005	FA-L/1	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0005	FA-L/2	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0006	FA-L/1	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0006	FA-L/2	333-44-5555
777-00-0005	FA-S/1	333-44-5555	777-00-0006	FA-L/3	333-44-5555
777-00-0005	FA-S/2	333-44-5555	777-00-0005	FA-S/1	333-44-5555
777-00-0005	FA-S/2	333-44-5555	777-00-0005	FA-S/2	333-44-5555
777-00-0005	FA-S/2	333-44-5555	777-00-0005	FA-L/1	333-44-5555
777-00-0005	FA-S/2	333-44-5555	777-00-0005	FA-L/2	333-44-5555
⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.21

1.2.2.5 The Set Operators

The relational algebra expressions which we have seen thus far have involved the operators σ, π, ×, and ρ; none have utilized set union, intersection, or difference. Examples of queries which can be writing in relational algebra utilizing the binary set operators are shown in Figures 1.22a-c, with corresponding relational algebra expressions in Figures 1.23a-c.

Find all companies which do not have a division named *research*.

Figure 1.22a

Find all companies which have neither a division named *research* nor a division named *development*.

Figure 1.22b

Find all companies which are either lacking a division named *research* or lacking a division named *development*.

Figure 1.22c

$$\pi_{company.govt\_id}[company] \;-\; \pi_{co\_id}(\sigma_{division.div\_name = "research"}[division])$$

Figure 1.23a

$$\left[\pi_{company.govt\_id}(company) \;-\; \pi_{co\_id}(\sigma_{division.div\_name = "research"}(division))\right]$$
$$\cap\; \left[\pi_{company.govt\_id}(company) \;-\; \pi_{co\_id}(\sigma_{division.div\_name = "development"}(division))\right]$$

Figure 1.23b

$$\left[\pi_{company.govt\_id}(company) \;-\; \pi_{co\_id}(\sigma_{division.div\_name = "research"}(division))\right]$$
$$\cup\; \left[\pi_{company.govt\_id}(company) \;-\; \pi_{co\_id}(\sigma_{division.div\_name = "development"}(division))\right]$$

Figure 1.23c

1.2.2.6 Reprise of Relational Algebra Operators

Additional details regarding the relational algebra operators are as follows:

- $\sigma_{<condition>}$<expression>:
  - <condition> must not refer to any attributes other than those in the value of <expression>
  - <condition> may be a simple condition, i.e., a comparison of two values, either of which may be the value of a named attribute or a literal value
  - The comparison operators are <, <=, =, <>, >=, >
  - <condition> may also be a compound condition, i.e., a Boolean expression involving simple conditions combined using *and*'s, *or*'s and *not*'s
- $\pi_{<attribute\ list>}$<expression>:
  - <attribute list> must not contain any attributes other than those in the value of <expression>

- •if the deletion from the value of <expression> of all columns in <attribute list> results in two or more rows' being identical, then the final result contains only one instance of that row

•<expression1> $\times$ <expression2>:

- •if the value of <expression1> has n columns and r1 rows and the value of <expression2> has m columns and r2 rows, then the result is an (n+m)-column table with r1*r2 rows
- •for each row in the value of <expression1> and for each row in the value of <expression2>, the result contains a row whose first n attributes' values are those of the row of <expression1> and whose remaining m attributes' values are those of the row of <expression2> -- and no other rows

•$\rho_{\text{<new name>(<attribute list>)}}$<expression>:

- •the number of attributes in <attribute list> must be the same as the number of attributes in the value of <expression>
- •the parenthesized <attribute list> is optional; if it is present, then the final result is a table identical to the value of <expression> except that its name is <new name> and the names of its attributes are those of <attribute list>
- •if the parenthesized <attribute list> is missing, then the result is exactly the same as above, except that the names of the attributes of the final result are those of the value of <expression>

•<expression1> $\cup$ <expression2>, <expression1> $\cap$ <expression2>, expression1> - <expression2>:

- •the value of <expression1> must have the same number of columns as the value of <expression2>
- •the types of the attributes of the value of <expression1> must be the same as the types of the attributes of the value of <expression2> respectively
- •if the application of $\cup$ results in two or more rows' being identical, then the final result contains only one instance of that row

Before we leave our discussion of the expressive power of relational algebra, and proceed to consider the type of query that can't be expressed, we revisit the question of the use of the dot (path-expression-style) notation to indicate which table an attribute comes from; recall, in this regard, that up to this point we have prefixed every attribute name with a table name. In general, though, a table name need not be appended unless there is ambiguity as to which table a particular occurrence of an attribute name belongs to. Thus, for example, the query of Figure 1.19 could be written as in Figure 1.24 with no change in its meaning.

$\pi_{govt\_id}(\sigma$ [company $\times$ division $\times$ site $\times$ product $\times$ person $\times$ works_for $\times$ skill])

```
(govt_id = division.co_id)
   and(hq_loc = "Atlanta")
   and(div_hq = "Boston")
   and(site.co_id = govt_id)
   and(site.div_name = division.div_name)
   and(site.loc = "Cleveland")
   and(product.manuf_co = govt_id)
   and(manuf_div = site.div_name)
   and(product.prod_id = "FA-S/2")
   and(person.ssn = ceo_ssn)
   and(name = "Joline Jones")
   and(works_for.ssn = person.ssn)
   and(works_for,co_id = govt_id)
   and(salary > 500000)
   and(skill.ssn = person.ssn)
   and(skill.prod_id = product.prod_id)
   and(skill.manuf_co = company.govt_id)
```

Figure 1.24

1.2.3 Limitations on the Expressive Power of Relational Algebra

The most basic SQL statement has essentially the same conceptual form and meaning as the skeletal relational algebra expression shown in Figure 1.25, albeit with considerably different syntax:

$\pi_{<\text{attribute list}>}(\sigma_{<\text{condition}>}[<\text{table}_1> \times <\text{table}_2> \times \ldots \times <\text{table}_n>])$

Figure 1.25

In addition, SQL statements of this basic form may be combined using set union, set intersection, and set difference. What, then, is missing from relational algebra that is required in a commercial query language? The most obvious limitation of relational algebra is that selection conditions can involve only attribute names and literal constants. The query of Figure 1.26a, for example, cannot be formulated in relational algebra, but could easily be formulated if relational algebra were extended to allow arithmetic expressions in selection conditions as in Figure 1.26b.

Find every employee of the company with *govt_id* 333-44-5555 who earns twice as much working for that company as does the supervisor of the division of the company to which the employee is assigned

Figure 1.26a

$$\pi_{emp.ssn}(\sigma_{\substack{(emp.co\_id = 3\text{-}444\text{-}5555)\\ and(sup.co\_id = emp.co\_id)\\ and(emp.sup\_ssn = sup.ssn)\\ and(emp.salary = 2*sup.salary)}} [\rho_{emp}(works\_for) \times \rho_{sup}(works\_for)])$$

Figure 1.26b

But the select operator in relational algebra is limited in more ways than just this one; even if relational algebra were extended to allow arithmetic expressions in selection conditions, there would appear to be no way, in relational algebra, to decide whether a row should be included in the value of an expression on the basis of summary statistics of an entire table. For example, it would appear to be impossible to write the queries of Figures 1.27a-f in relational algebra.

Find every company that has exactly three divisions

Figure 1.27a

Find every company that has more divisions with headquarters in Baltimore than divisions with headquarters in Saint Louis

Figure 1.27b

Find every division whose total salary budget exceeds $3,000,000 per year

Figure 1.27c

Find every division whose average salary exceeds $50,000 per year

Figure 1.27d

Find the person(s) who is(are) paid the highest salary by the company whose govt_id is 666-77-8888

Figure 1.27e

Find the person(s) who is(are) paid the lowest salary by the company whose govt_id is 666-77-8888

Figure 1.27f

The summary-statistics functions which appear to be necessary for the expression of the queries of Figures 1.27a-f are:

- COUNT the number of rows in a table
- SUM the values of an attribute
- AVERAGE the values of an attribute
- find the MINIMUM value of an attribute
- find the MAXIMUM value of an attribute

The reader who is already acquainted with SQL will recognize COUNT, SUM, AVERAGE, MAXIMUM and MINIMUM as the so-called *aggregate* functions of SQL. The COUNT query of Figure 1.27b, the SUM query of Figure 1.27c, and the AVERAGE query of Figure 1.27d cannot be written in relational algebra; the COUNT query of Figure 1.27a can, however, be written in relational algebra as, somewhat surprisingly[3], can the MAXIMUM and MINIMUM queries of Figures 1.27e and 1.27f -- for which see Figures 1.28a-c respectively. In order to understand how it is that the relational algebra expression of Figure 1.28a implements the query of Figure 1.27a, note that:

- the first major sub-expression of Figure 1.28a, i.e., the one before the difference operator, finds the *co_id*'s of companies that have three or more divisions
- the second major sub-expression of Figure 1.28a, i.e., the one after the difference operator, finds the *co_id*'s of companies that have four or more divisions

The relational algebra expression of Figure 1.28b is best understood by noting that:

- the first major sub-expression of Figure 1.28b, i.e., the one before the difference operator, finds the *ssn*'s of all people who work for the company in question
- the second major sub-expression of Figure 1.28b, i.e., the one after the difference operator, finds the *ssn*'s of people who work for the company in question and for whom there is someone working for the company who is paid more
- the entire expression therefore finds the ssn's of people who work for the company and for whom there is no one working for the company who is paid more

The relational algebra expression of Figure 1.28c is best understood by noting that:

- the first major sub-expression of Figure 1.28c, i.e., the one before the difference operator, finds the *ssn*'s of all people who work for the company in question
- the second major sub-expression of Figure 1.28c, i.e., the one after the difference operator, finds the *ssn*'s of people who work for the company in question and for whom there is someone working for the company who is paid less

[3]The authors of at least one popular textbook indicate that queries of this sort cannot be written in relational algebra.

- the entire expression therefore finds the ssn's of people who work for the company and for whom there is no one working for the company who is paid less

Note, however, that even if a query can be expressed in relational algebra, its expression may be extremely verbose. Consider as an example, the length of the relational algebra expression which would be required if the query of Figure 1.27a were changed so as to request companies with exactly *thirty* divisions.

$$\pi_{div1.co\_id}(\sigma_{\substack{(div1.co\_id = div2.co\_id) \\ and(div2.co\_id = div3.co\_id) \\ and(div1.div\_name <> div2.div\_name) \\ and(div1.div\_name <> div3.div\_name) \\ and(div2.div\_name <> div3.div\_name)}} [\rho_{div1}(division) \times \rho_{div2}(division) \times \rho_{div3}(division)])$$

$$- \; \pi_{div1.co\_id}(\sigma_{\substack{(div1.co\_id = div2.co\_id) \\ and(div2.co\_id = div3.co\_id) \\ and(div3.co\_id = div4.co\_id) \\ and(div1.div\_name <> div2.div\_name) \\ and(div1.div\_name <> div3.div\_name) \\ and(div1.div\_name <> div4.div\_name) \\ and(div2.div\_name <> div3.div\_name) \\ and(div2.div\_name <> div4.div\_name) \\ and(div3.div\_name <> div4.div\_name)}} [\rho_{div1}(division) \times \rho_{div2}(division) \times \rho_{div3}(division) \times \rho_{div4}(division)])$$

Figure 1.28a

$$\pi_{ssn}(\sigma_{co\_id = 555\text{-}55\text{-}5555} [works\_for]) \quad - \quad \pi_{wf\_1.ssn}(\sigma_{\substack{(w\_f1.co\_id = 555\text{-}55\text{-}5555) \\ and(w\_f1.co\_id = w\_f2.co\_id) \\ and(w\_f1.salary < w\_f2.salary)}} [\rho_{w\_f1}(works\_for) \times \rho_{w\_f2}(works\_for)])$$

Figure 1.28b

$$\pi_{ssn}(\sigma_{co\_id = 555\text{-}55\text{-}5555} [works\_for]) \quad - \quad \pi_{wf\_1.ssn}(\sigma_{\substack{(w\_f1.co\_id = 555\text{-}55\text{-}5555) \\ and(w\_f1.co\_id = w\_f2.co\_id) \\ and(w\_f1.salary > w\_f2.salary)}} [\rho_{w\_f1}(works\_for) \times \rho_{w\_f2}(works\_for)])$$

Figure 1.28c

Each of COUNT, AVG, SUM, MIN, and MAX, as described above, takes a table as its operand and produces a scalar value as its result -- or, if we care to look at it that way, a table with a single row and a single column. If we were to add these functions to relational algebra, an appropriate place to add them would be the subscripted list of a project operator. (It is, of course, in SQL's analog of the project operator that these functions are permitted to occur in SQL.)

If we did so, then the query of Figure 1.29a might be written as in Figure 1.29b. Note, however, that the wording of queries 1.27a-f suggests that the results of these functions are to be used in selection conditions, i.e., in places where relational algebra allows only attribute names and literal constants. The logical extension of the addition of the six aggregate functions to a query language would therefore be to allow single-value-producing expressions to appear in (be embedded in) select conditions. If we took this logical next step in the case of relational algebra, then we could write the queries of Figures 1.27a-f as in Figures 1.30a-f, in which the original English queries have been repeated for the sake of convenience. Naturally, SQL handles this construct in an analogous fashion.

Find the number of employees of the company whose govt_id is 333-44-5555, the average of the salaries paid by the company to its employees, company's total salary budget, and the minimum and maximum salaries paid by the company to its employees

Figure 1.29a

$$\pi_{\text{COUNT(ssn), AVG(salary), SUM(salary), MIN(salary), MAX(salary)}}(\sigma_{\text{co\_id = "333-44-5555"}}(\text{works\_for})$$

Figure 1.29b

Find every company that has exactly three divisions

$$\pi_{\text{company.govt\_id}}(\sigma_{(\pi_{\text{COUNT(division.div\_name)}}(\sigma_{\text{company.govt\_id = division.co\_id}}(\text{division}))) = 3}(\text{company}))$$

Figure 1.30a

Find every company that has more divisions with headquarters in Baltimore than divisions with headquarters in Saint Louis

$$\pi_{company.govt\_id}(\sigma(company))$$

$$(\pi_{COUNT(division.div\_name)}(\sigma_{\substack{(company.govt\_id\ =\ division.co\_id)\\ \text{and } (division.div\_hq\ =\ \text{"Baltimore"})}}(division)))$$

$$>$$

$$(\pi_{COUNT(division.div\_name)}(\sigma_{\substack{(company.govt\_id\ =\ division.co\_id)\\ \text{and } (division.div\_hq\ =\ \text{"Saint Louis"})}}(division)))$$

Figure 1.30b

Find every division whose total salary budget exceeds $3,000,000 per year

$$\pi_{division.co\_id,\ division.div\_name}(\sigma\ (division))$$

$$\pi_{SUM(works\_for.salary)}(\sigma_{\substack{(works\_for.co\_id\ =\ division.co\_id)\\ \text{and } (works\_for.div\_name\ =\\ division.div\_name)}}(works\_for))) > 3000000$$

Figure 1.30c

Find every division whose average salary exceeds $50,000 per year

$$\pi_{division.co\_id,\ division.div\_name}(\sigma\ (division))$$

$$\pi_{AVG(works\_for.salary)}(\sigma_{\substack{(works\_for.co\_id\ =\ division.co\_id)\\ \text{and } (works\_for.div\_name\ =\\ division.div\_name)}}\ (works\_for)) > 50000$$

Figure 1.30d

Find the person(s) who is(are) paid the highest salary by the company whose govt_id is 666-77-8888

$$\pi_{ssn}(\sigma_{(w\_f1.govt\_id = "666\text{-}77\text{-}8888") \text{ and } \pi_{MAX(w\_f2.salary)}(\sigma(\rho_{w\_f2}(works\_for))) = w\_f1.salary}(\rho_{wf1}(works\_for)))$$

Figure 1.30e

Find the person(s) who is (are) paid the lowest salary by the company whose govt_id is 666-77-8888

$$\pi_{ssn}(\sigma_{(w\_f1.govt\_id = "666\text{-}77\text{-}8888") \text{ and } \pi_{MIN(w\_f2.salary)}(\sigma(\rho_{w\_f2}(works\_for))) = w\_f1.salary}(\rho_{w\_f1}(works\_for)))$$

Figure 1.30f

Suppose, then, that we have added the functions COUNT, AVG, SUM, MIN, and MAX to relational algebra as above. One final type of query which was of interest to the designers of SQL would still be inexpressible in relational algebra, i.e., that type of query which requires that a table be partitioned into sub-tables, and that summary statistics be computed for each of the sub-tables. An example of such a query is shown in Figure 1.31, in which the *works_for* table must be partitioned into sub-tables each of which contains rows relating to exactly one company, i.e., rows which all agree on their values of *co_id*; the average of the *salary* fields must then be computed and output for each of the sub-tables.

For each company,[4] find the average salary paid by the company to its employees.

Figure 1.31

Finally, but not exhaustively, it is impossible to express *transitive closure* queries in relational algebra, queries such as that of Figure 1.32; this is also, by the way, a limitation of SQL, one which is being addressed in current standardization discussions.

Find all *ancestor* divisions of the "research" division of the company whose co_id is 111-11-2222, i.e., the division of which the research division is a sub--division (the company's *immediate* ancestor), the division of which the research division's immediate ancestor is a sub-division, etc., etc.

Figure 1.32

[4]I.e., for each company with a listing (row) in the *company* table.

The reader whose curiosity has been whetted sufficiently to be desirous of developing a clearer, more formal, view of the sorts of queries which can be written in relational algebra and those which can't, may wish to consult one of the many relational database textbooks which discuss the other of Codd's two proposed query languages, *relational calculus*, a language based upon first-order predicate calculus and equivalent in expressive power to relational algebra. Relational calculus is not, however, recommended for those with a purely practical interest in relational database programming; nor is it recommended as part of an introductory course in relational database systems as it's non-procedural, i.e., non-computational, nature is extremely confusing to most students, even those with strong programming backgrounds.

2. SQL

The presentation of a programming language typically proceeds through discussions of the following topics:

- the system of built-in data types
- the system of type constructors
- the type declaration statement
- the variable declaration statement
- the system of built-in operations on variables
- the system of expressions
- the scope rules
- the assignment statement
- the control-flow statements

SQL is a very-high-level special-purpose language so that not all of these categories will be directly applicable to it; it is, nevertheless, instructive to compare SQL's constructs to those of a lower-level, general purpose programming language.

2.1 Basics

Each of the SQL standards specifies a list of built-in data types which include ones for character strings, of both fixed and varying lengths (CHAR and VARCHAR); bit strings, of both fixed and varying lengths (BIT and BIT VARYING); integers, of both standard length and short (INT and SMALLINT); fixed point numbers with user declarable numbers of digits and numbers of digits after the decimal point (DECIMAL); reals and double-precision reals (FLOAT, REAL, and DOUBLE PRECISION); and dates (DATE), times (TIME), and time intervals (INTERVAL).

Each standard also specifies a number of built-in functions on built-in types, functions which compute the lengths of variable-length character (CHAR_LENGTH) and bit strings (BIT_LENGTH); functions which cast values of one type to another type (CAST); functions which return the current time (CURRENT_TIME) and date (CURRENT_DATE), etc., etc., etc. Most commercial database products add their own built-in types and built-in operations on these types -- in some cases extremely long lists of both. In line with our expressed intention to discuss the most powerful/interesting aspects of SQL, rather than to be exhaustive, we will omit any further discussion of built-in types and operations on built-in types, except to note that SQL enables the user to declare and name restricted versions of built-in types using the **CREATE DOMAIN** statement[5]. . There are also **ALTER DOMAIN** and **DROP DOMAIN** statements with obvious meanings.

---

[5]For example, ***CREATE DOMAIN*** *foobar* ***AS VARCHAR****(30)* declares *foobar* as a name of the type *VARCHAR(30). VARCHAR(30)* is the designation of the built-in type "variable length character string with maximum length of 30 characters." We say that "**CREATE DOMAIN** foobar **AS VARCHAR**(30) declares foobar as *a* name of the type VARCHAR(30)," because it is perfectly legal to declare additional domains with specifications of *VARCHAR(30)*.

SQL's only type constructor is *table of <list of built-in types>*, with a type declaration statement whose introductory key words are, not surprisingly, **CREATE TABLE**; the **CREATE TABLE** statements necessary to declare the table schemes of Figure 1.1, together with their keys and foreign key references, is shown in Figures 2.1a-g. (Note that for every table scheme we have indicated that none of the attributes in the scheme's key may be NULL, a fairly common practice.) As we indicated in our discussion of relational algebra, only a single instance of each user-declared type is permitted, so **CREATE TABLE** is simultaneously SQL's type declaration statement and its variable declaration statement.

```
CREATE TABLE company
   (    co_name              VARCHAR(30),
        govt_id                  CHAR(11)      NOT NULL,
        ceo_ssn                  CHAR(11),
        hq_loc               VARCHAR(30),
        PRIMARY KEY(govt_id),
        FOREIGN KEY(ceo_ssn) REFERENCES person(ssn))
```

Figure 2.1a

```
CREATE TABLE division
   (    co_id                CHAR(11),                NOT NULL,
        divname              VARCHAR(30)         NOT NULL,
        subdiv_of            VARCHAR(30),
        dir_ssn              CHAR(11),
        div_hq               VARCHAR(30),
        PRIMARY KEY(co_id, div_name),
        FOREIGN KEY(co_id) REFERENCES company(govt_id),
        FOREIGN KEY(dir_ssn) REFERENCES person(ssn))
```

Figure 2.1b

```
CREATE TABLE site
   (    co_id                CHAR(11),                NOT NULL,
        divname              VARCHAR(30)         NOT NULL,
        loc                  VARCHAR(30)         NOT NULL,
        PRIMARY KEY(co_id, div_name, loc),
        FOREIGN KEY(co_id, div_name)  REFERENCES division(co_id, div_name),
        FOREIGN KEY(dir_ssn)               REFERENCES person(ssn))
```

Figure 2.1c

```
CREATE TABLE product
    (   prod_id             VARCHAR(10)        NOT NULL,
        manuf_co            CHAR(11)           NOT NULL,
        manuf_div           VARCHAR(30),
        loc                 VARCHAR(30),
        prod_descr          VARCHAR(50),
        PRIMARY KEY(prod_id, manuf_co),
        FOREIGN KEY(manuf_co, manuf_div, loc)
                                               REFERENCES site(co_id, manuf_div, loc)),
        FOREIGN KEY(manuf_co, manuf_div)
                                               REFERENCES division(co_id, div_name))
```

Figure 2.1d

```
CREATE TABLE person
    (   ssn                 CHAR(11)           NOT NULL,
        name                VARCHAR(30),
        address             VARCHAR(50),
        PRIMARY KEY(ssn))
```

Figure 2.1e

```
CREATE TABLE works_for
    (   ssn                 CHAR(11)           NOT NULL,
        co_id               CHAR(11)           NOT NULL,
        div_name            VARCHAR(30),
        salary              DECIMAL(9,2),
        emp_id              VARCHAR(15),
        sup_ssn             CHAR(11),
        PRIMARY KEY(ssn, co_id),
        FOREIGN KEY(ssn)                       REFERENCES person(ssn),
        FOREIGN KEY(co_id, div_name)  REFERENCES division(co_id, div_name),
        FOREIGN KEY(sup_ssn)              REFERENCES person(ssn))
```

Figure 2.1f

```
CREATE TABLE skill
   (  ssn              CHAR(11)        NOT NULL,
      prod_id              VARCHAR(10)   NOT NULL,
      manuf_co         VARCHAR(30)   NOT NULL,
      PRIMARY KEY(ssn, prod_id, manuf_co),
      FOREIGN KEY(ssn)                          REFERENCES person(ssn),
      FOREIGN KEY(prod_id, manuf_co)    REFERENCES product(prod_id, manuf_co))
```

Figure 2.1g

Since SQL is a language intended to deal with *persistent* values, i.e., values whose existence persists beyond the execution of a single program, possibly over a large period of time, SQL also has a type re-declaration statement (**ALTER TABLE**), and a type deletion statement (**DELETE TABLE**). Naturally, there are statements for populating a table with rows (**INSERT**), for deleting rows (**DELETE**), and for modifying rows (**UPDATE**).

The notion of an *expression* in SQL, or, at least, the closest thing to an expression in SQL, is based upon SQL's most interesting statement, the SELECT-FROM-WHERE statement, or SFW for short. For the purposes of this discussion, we will define an SQL expression as follows:

<exp> = SFW | <exp> EXCEPT[6] <exp> | <exp> UNION <exp> | <exp> INTERSECT <exp>

An SFW is very similar to a project-select-cartesian product in relational algebra, rendering SQL expressions very similar to those of relational algebra. We should note, however, that, in contradistinction to the assumption which we made when we discussed relational algebra, in the realm of SQL, tables may have duplicate rows.

Finally, those control-flow constructs that SQL has are buried in the SFW statement; the notion of *scope* in SQL is meaningful only in the context of (nested) SFW statements; and the closest thing that SQL has to an assignment statement, unless one considers the **INSERT** and **UPDATE** statements to be assignments, is the **CREATE VIEW** (Section 2.3), a statement which takes an entire SQL expression as a sub-statement. We therefore proceed to a discussion of the SQL expression.

2.2 The SQL Expression

The most interesting aspect of the SQL expression is the SFW, so we will devote the bulk of our discussion to a detailed discussion of it. In its most general form the SFW consists of six clauses as in Figure 2.2; the last four clauses are, however, optional, in fact independently optional except for the inter-dependence of the GROUP BY and HAVING clauses: the HAVING clause may be present only if the GROUP BY clause is present.

[6]EXCEPT is SQL's notation for *set difference*.

```
SELECT <output list>
FROM <table list>
WHERE <condition>
GROUP BY <attribute list>
HAVING <condition>
ORDER BY <attribute list>
```

Figure 2.2

The first point that should be noted for the reader familiar with relational algebra is that SELECT in the SFW corresponds to *project* (π) in relational algebra, and that WHERE is the SFW's version of *select* (σ); The second point to be noted is that the order of execution of the clauses of an SFW statement is not precisely as might be expected, but is, rather, as follows:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. ORDER BY
6. SELECT

(Since SELECT serves the function of *project* in relational algebra, and since *project* is, effectively, the operator which outputs the results of a query, it should not come as a surprise that SELECT is executed last -- though the reader might justifiably wonder why it takes first position syntactically.) In an SFW statement which is missing one or more optional clauses, each missing clause may be thought of as being replaced by a no-op, with the order of execution remaining as above.

The quickest way to understand the SFW statement is to note that the relational algebra expression of Figure 2.3a can be translated into SQL as the SFW statement of Figure 2.3b. More concretely, the queries of Figures 1.4a-g may be translated into SQL as shown in Figures 2.4a-g.

$$\pi_{\text{<attribute list>}}(\sigma_{\text{<condition>}}[\text{<table}_1\text{>} \times \text{<table}_2\text{>} \times \ldots \times \text{<table}_n\text{>}])$$

Figure 2.3a

SELECT <attribute list>
FROM <table$_1$>, <table$_2$>, ... , <table$_n$>
WHERE <condition>

Figure 2.3b

```
SELECT company.govt_id
FROM company
WHERE hq_loc = "Atlanta"
```

Figure 2.4a

```
SELECT company.govt_id
FROM company, division
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
```

Figure 2.4b

```
SELECT company.govt_id
FROM company, division, site
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
        AND (site.co_id = company.govt_id)
        AND (site.div_name = division.div_name)
        AND (site.loc = "Cleveland")
```

Figure 2.4c

```
SELECT company.govt_id
FROM company, division, site, product
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
        AND (site.co_id = company.govt_id)
        AND (site.div_name = division.div_name)
        AND (site.loc = "Cleveland")
        AND (product.manuf_co = company.govt_id)
        AND (product.manuf.div = site.div_name)
        AND (product.prod_id = FA-S/2)
```

Figure 2.4d

```
SELECT company.govt_id
FROM company, division, site, product, person
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
        AND (site.co_id = company.govt_id)
        AND (site.div_name = division.div_name)
        AND (site.loc = "Cleveland")
        AND (product.manuf_co = company.govt_id)
        AND (product.manuf.div = site.div_name)
        AND (product.prod_id = FA-S/2)
        AND (person.ssn = company.ceo_ssn)
        AND (person.name = "Joline Jones")
```

Figure 2.4e

```
SELECT company.govt_id
FROM company, division, site, product, person, works_for
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
        AND (site.co_id = company.govt_id)
        AND (site.div_name = division.div_name)
        AND (site.loc = "Cleveland")
        AND (product.manuf_co = company.govt_id)
        AND (product.manuf.div = site.div_name)
        AND (product.prod_id = FA-S/2)
        AND (person.ssn = company.ceo_ssn)
        AND (person.name = "Joline Jones")
        AND (works_for.ssn = person.ssn)
        AND (works_for,co_id = company.govt_id)
        AND (works_for.salary > 500000)
```

Figure 2.4f

```
SELECT company.govt_id
FROM company, division, site, product, person, works_for, skill
WHERE (company.govt_id = division.co_id)
        AND (company.hq_loc = "Atlanta")
        AND (division.div_hq = "Boston")
        AND (site.co_id = company.govt_id)
        AND (site.div_name = division.div_name)
        AND (site.loc = "Cleveland")
        AND (product.manuf_co = company.govt_id)
        AND (product.manuf.div = site.div_name)
        AND (product.prod_id = FA-S/2)
        AND (person.ssn = company.ceo_ssn)
        AND (person.name = "Joline Jones")
        AND (works_for.ssn = person.ssn)
        AND (works_for,co_id = company.govt_id)
        AND (works_for.salary > 500000)
        AND (skill.ssn = person.ssn)
        AND (skill.prod_id = product.prod_id)
        AND (skill.manuf_co = company.govt_id)
```

Figure 2.4g

2.2.1 The SELECT CLAUSE

If all the attributes in the table produced by the FROM clause are desired in the output, they need not all be listed; rather, the form *SELECT* * may be used with the meaning *select all attributes*. Thus, the SFW of Figure 2.5a has precisely the same meaning as that of Figure 2.5b.

```
SELECT *
FROM company
```

Figure 2.5a

```
SELECT co_name, govt_id, ceo_ssn, hq_loc
FROM company
```

Figure 2.5b

If the projection specified by the attribute list of a SELECT clause (potentially) results in a table with repeated rows, we can ask for repeated rows to be deleted by appending the key word DISTINCT after the word SELECT and before the list of attributes. (The opposite of DISTINCT is ALL, but as it is the default, it is rarely used.)

Thus, the SFW of Figure 2.6a, without the key word DISTINCT would, if applied to the tables of Figures 1.2a-g, produce the table of Figure 2.6b; the SFW of Figure 2.7a, with the key word DISTINCT would produce the table of Figure 2.7b.

```
SELECT hq_loc
FROM company
```

Figure 2.6a

hq_loc
New York
Trenton
Atlanta
Atlanta
Columbia
Atlanta
Pensacola
Toledo
• • •

Figure 2.6b

```
SELECT DISTINCT hq_loc
FROM company
```

Figure 2.7a

hq_loc
New York
Trenton
Atlanta
Columbia
Pensacola
Toledo
• • •

Figure 2.7b

The arithmetic operators which may be used in the SELECT and WHERE clauses are +, -, *, and /. Parentheses are allowed. Since a computed value in a SELECT clause has no attribute name, SQL allows the user to specify a name through the use of the key word AS. Thus, assuming a tax rate of 28%, the query of Figure 2.8a can be written as shown in Figure 2.8b, where the legend *tax* will be printed as the name of the second column of the result table.

Find the tax which must be paid by each person who works for the company whoe govt_id is 111-22-3333.

Figure 2.8a

```
SELECT works_for.ssn, .28*works_for.salary AS tax
FROM works_for
WHERE works_for.co_id = "111-22-3333"
```

Figure 2.8b

In fact, any attribute in a SELECT clause can be renamed for output; thus, if we want the legends on the two columns of the output of Figure 2.8a to be *employee* and *tax* respectively, we can write the query as in Figure 2.9

```
SELECT works_for.ssn AS employee, .28*works_for.salary AS tax
FROM works_for
WHERE works_for.co_id = "111-22-3333"
```

Figure 2.9

Query 1.29a, our first example of a query requiring the use of aggregate functions, is easily translated from the extended relational algebra of Figure 1.15b into the SQL of Figure 2.10.

```
SELECT COUNT(ssn), AVG(salary), SUM(salary), MIN(salary), MAX(salary)
FROM works_for
WHERE co_id = "333-44-5555"
```

Figure 2.10

COUNT(ssn) counts the number of occurrences of *ssn* values in the table produced by the FROM and WHERE clauses. Since the number of occurrences of *ssn* values is, like the number of occurrences of any attribute, the same as the number of rows in the table produced by the FROM and WHERE clauses, we could have used

COUNT(*) read *count all*, to return the number of rows in the table produced by the FROM and WHERE clauses.

When we want the number of distinct values which occur as values of an attribute, we add the key word DISTINCT before the name of the attribute as in *COUNT(DISTINCT ssn)*. In the case of the query of Figure 2.10, *ssn* and *co_id* together constitute a key for *works_for*, and since the WHERE clause fixes the value of *co_id* to a single value, the table produced by the FROM and WHERE clauses contains no repeated *ssn* values, thereby causing the number of rows in that table to be precisely the number of people working for the company of interest, so the key word DISTINCT would effect no change in the final result.

In the general case,

COUNT(DISTINCT <attribute$_1$> <attribute$_2$> ... <attribute$_n$>

returns the number of distinct n-tuples of values in the indicated columns of the table produced by the FROM and WHERE clauses.

Thus, for example, if the *company* table were temporarily changed to that of Figure 2.11a, the query of Figure 2.11b would produce a result of 3 (three), whereas the query of Figure 2.11c would produce a result of 1 (one).

co_name	govt_id	ceo_ssn	hq_loc
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	999-00-0002	Trenton
Atlanta Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia
Atlanta Axle	666-77-8888	999-00-0005	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Ohio Windshield	888-99-0000	999-00-0007	Toledo
⋮	⋮	⋮	⋮

Figure 2.11a

```
SELECT COUNT(co_name, hq_loc)
FROM company
```

Figure 2.11b

```
SELECT COUNT(DISTINCT co_name, hq_loc)
FROM company
```

Figure 2.11c

Only the SELECT clause of an SFW may contain an application of an aggregate function; a query such as that shown in Figure 2.12 is, therefore, illegal -- on semantic and, as a consequence, syntactic grounds -- though the beginning student of SQL is often tempted to write statements of this form.

```
SELECT co_id
FROM works_for
WHERE COUNT(ssn) > 100
```

Figure 2.12

The reader who is not convinced of the incorrectness of the placement of the COUNT in the WHERE clause of Figure 2.12 is reminded that the condition in a WHERE clause, like that in a relational algebra *select*, is applied to *each row* of the table produced by the FROM clause -- the table to which the *select* is applied, frequently a simple cartesian product in the case of relational algebra; in the case under consideration this would mean testing each row of *works_for* for a number of *ssn*'s greater than 100.

We will delay our discussion of the use of aggregate function in the SELECT clauses of embedded SFW's until we have discussed the subject of embedding -- in our discussion of the WHERE clause.

2..2.2 The WHERE Clause

Being the SFW's version of relational algebra's *select* operator, the WHERE clause performs condition testing, on a row-by-row basis, upon the table produced by the FROM clause. We will begin by considering simple condition tests (Section 2.2.2.1), and will then proceed to a discussion of tests involving embedded expressions (Section 2.2.2.2).

2.2.2.1 Simple Conditions

The SFW's basic comparison operators include: <, <=, =, <>, >=, >. For string comparisons, SQL provides pattern-matching operators named LIKE and NOT LIKE, where % represents any sequence of one or more characters and _ represents any single character, so that the queries of Figures 2.13a-d can be written in SQL as in Figures 2.13e-h respectively.

Find all ceo's whose names start with a *K*.

Figure 2.13a

Find all ceo's whose names don't start with a *K*.

Figure 2.13b

Find all ceo's whose names are five letters long.

Figure 2.13c

Find all ceo's whose names aren't five letters long.

Figure 2.13d

```
SELECT company.ceo_ssn
FROM company, person
WHERE (company.ceo_ssn = person.ssn)
        AND (person.name LIKE 'K%'
```

Figure 2.13e

```
SELECT company.ceo_ssn
FROM company, person
WHERE (company.ceo_ssn = person.ssn)
        AND (person.name NOT LIKE 'K%'
```

Figure 2.13f

```
SELECT company.ceo_ssn
FROM company, person
WHERE (company.ceo_ssn = person.ssn)
        AND (person.name LIKE '_ _ _ _ _'
```

Figure 2.13g

```
SELECT company.ceo_ssn
FROM company, person
WHERE (company.ceo_ssn = person.ssn)
                AND (person.name NOT LIKE '_ _ _ _ _'
```

Figure 2.13h

Attribute values may be NULL, but NULL-ness of a value cannot be tested through the use of a condition of the form *<attribute name> = NULL*; rather, the proper syntax is *<attribute name> IS NULL*; similarly, non-NULL-ness is tested as *<attribute name> IS NOT NULL*.

While the value of an attribute may be tested for being between two specified values through the use of two simple comparisons AND'ed together, as in

```
(<value1> <= <attribute value>)
AND (<attribute value> <= <value 2>
```

the two (inclusive) comparisons can also be written as

```
<attribute name> BETWEEN <value 1> and <value 2>
```

Expressions of the form

```
<attribute name> NOT BETWEEN <value 1> and <value 2>
```

can also be used -- with the obvious meaning.

As we have already hinted, WHERE clause conditions can be combined using the logical operators AND, OR, and NOT. As is probably also expected, parentheses may be used in WHERE clause conditions.

2.2.2.2 Conditions Involving Embedded Expressions

The queries which we used to introduce the notion of aggregate functions in embedded relational algebra sub-expressions, queries 1.27a-f can be written in SQL through the use of aggregate functions in the SELECT clauses of embedded sub-expressions as in Figures 2.14a-f (English queries repeated).

Find every company that has exactly three divisions

```
SELECT division.co_id
FROM division div1
WHERE   ( SELECT COUNT(div_name)
                FROM division div2
                WHERE div2.co_id = div1.co_id) = 3
```

Figure 2.14a

Find every company that has more divisions with headquarters in Baltimore than divisions with headquarters in Saint Louis

```
SELECT company.govt_id
FROM company
WHERE   ( SELECT COUNT(division.div_name)
                FROM division
                WHERE company.govt_id = division.co_id)
                        AND (division.div_hq = "Baltimore")

                          >

              ( SELECT COUNT(division.div_name)
                  FROM division
                  WHERE (ompany.govt_id = division.co_id)
                          AND (division.div_hq = "Saint Louis")
```

Figure 2.14b

Find every division whose total salary budget exceeds $3,000,000 per year

```
SELECT division.co_id, division.div_name
FROM division
WHERE   ( SELECT SUM(works_for.salary)
                FROM works_for
                WHERE (works_for.co_id = division.co_id)
                      AND (works_for.div_name = division.div_name)) > 3000000
```

Figure 2.14c

Find every division whose average salary exceeds $50,000 per year

```
SELECT division.co_id, division.div_name
FROM division
WHERE   ( SELECT AVG(works_for.salary)
              FROM works_for
              WHERE (works_for.co_id = division.co_id)
                      AND (works_for.div_name = division.div_name)) > 50000
```

Figure 2.14d

Find the person(s) who is(are) paid the highest salary by the company whose govt_id is 666-77-8888

```
SELECT ssn
FROM works_for wf1
WHERE w_f1.co_id = "666-77-8888= (  SELECT MAX(w_f2..salary)
                                        FROM works_for w_f2) = w_f1.salary)
```

Figure 2.14e

Find the person(s) who is(are) paid the lowest salary by the company whose govt_id is 666-77-8888

```
SELECT ssn
FROM works_for wf1
WHERE w_f1.co_id = "666-77-8888"  = (  SELECT MIN(w_f2..salary)
                                         FROM works_for w_f2) = w_f1.salary)
```

Figure 2.14f

In addition to the simple comparison operators, there are also a few non-comparison operators, which can be used to introduce embedded sub-expressions's, viz.:

- •IN and NOT IN
- •EMPTY and NOT EMPTY
- •For each <comparison op> in {<, <=, =, <>, >, >=}:
    - •<comparison op>ALL
    - •<comparison op>SOME
    - •<comparison op>ANY.

The uses of these operators and their meanings are as follows:

The embedding expression *<value> IN <SFW>*

- •returns *true* if <SFW> returns a table with a single column, and <value> equals the value in one of the rows of the table
- •returns *false* if <SFW> returns a table with a single column, and <value> does not equal the value in any of the rows of the table
- •is a syntax error otherwise.

Thus, the query of Figure 2.15a can be written as in Figure 2.15b

Find the company headquarters location of every company that has a division headquartered in Atlanta.

Figure 2.15a

```
SELECT company.hq_loc
FROM company
WHERE company.govt_id IN (SELECT division.co_id
                          FROM division
                          WHERE division.div_hq = "Atlanta")
```

Figure 2.15b

Note, of course, that this is not the only way to write the query of Figure 2.15a; rather, it can be written, without an embedded SFW, as in Figure 2.15c

```
SELECT company.hq_loc
FROM company, division
WHERE   (company.govt_id = division.co_id)
          AND (division.div_hq = "Atlanta")
```

Figure 2.15c

In fact, SQL has been severely criticized by some because it often allows an English query to be written in numerous different ways.

The embedding expression *<value> NOT IN <SFW>*

- •returns *false* if <SFW> returns a table with a single column, and <value> equals the value in one of the rows of the table
- •returns *true* if <SFW> returns a table with a single column, and <value> does not equal the value in any of the rows of the table
- •is a syntax error otherwise.

Thus, the query of Figure 2.16a can be written as in Figure 2.16b

Find the company headquarters location of every company that does not have a division headquartered in Atlanta.

Figure 2.16a

```
SELECT company.hq_loc
FROM company
WHERE company.govt_id NOT IN (SELECT division.co_id
                              FROM division
                              WHERE division.div_hq = "Atlanta")
```

Figure 2.16b

Note that this query can be written in at least one alternative way, but not quite as easily as that of Figure 2.16a; one alternative version, that of Figure 2.16c, uses the operator *set difference*, denoted EXCEPT in SQL.

```
SELECT company.hq_loc
FROM company
WHERE company.govt_id IN   ((SELECT division.co_id
                            FROM division)

                        EXCEPT

                            ( SELECT division.co_id
                              FROM division
                              WHERE division.div_hq = "Atlanta"))
```

Figure 2.16c

The embedding expression *(<value$_1$>, <value$_2$>, ... , <value$_n$>) IN <SFW>*:

- returns *true* if <SFW> returns a table with n columns, and one of the rows of that table has column values of <value$_1$>, <value$_2$>, ... , and <value$_n$> in that order
- returns *false* if <SFW> returns a table with n columns, and none of the rows of that table has column values of <value$_1$>, <value$_2$>, ... , and <value$_n$> in that order

- is a syntax error otherwise.

Thus, the query of Figure 2.17a can be written as in Figure 2.17b

Find the division headquarters location of every division (of any company) which has a site in Atlanta

Figure 2.17a

```
SELECT division.div_hq
FROM division
WHERE   (division.co_id, division.div_name) IN ( SELECT site.co_id, site.div_name)
                                                 FROM site
                                                 WHERE site.loc = "Atlanta")
```

Figure 2.17b

The embedding expression *(<value$_1$>, <value$_2$>, ... , <value$_n$>) NOT IN <SFW>*:

- returns *false* if <SFW> returns a table with n columns, and one of the rows of that table has column values of <value$_1$>, <value$_2$>, ... , and <value$_n$> in that order
- returns *true* if <SFW> returns a table with n columns, and none of the rows of that table has column values of <value$_1$>, <value$_2$>, ... , and <value$_n$> in that order
- is a syntax error otherwise.

Thus, the query of Figure 2.18a can be written as in Figure 2.18b

Find the division headquarters location of every division (of any company) which has no site in Atlanta

Figure 2.18a

```
SELECT division.div_hq
FROM division
WHERE (division.co_id, division.div_name) NOT IN ( SELECT site.co_id, site.div_name)
                                                   FROM site
                                                   WHERE site.loc = "Atlanta")
```

Figure 2.18b

The embedding expression *EMPTY <SFW>*:

- returns *true* if <SFW> returns an empty table, i.e., a table with zero rows
- returns *false* otherwise.

Using EMPTY <SFW>, we can write the query of Figure 2.18a in yet another way. In order to understand how, we take a short digression into the subject of name scopes in nested expressions.

Once we allow an SFW statement to be embedded in another SFW statement, the question of name scopes arises. In the few cases which we have already seen, the answers were quite obvious; in the query whose *EMPTY <SFW>* version led us to this digression, the answer will be considerably less obvious. We therefore introduce the question of scopes with examples that use simple comparison operators to introduce embedding.

For example, suppose that we wrote the query of Figure 1.27f as in Figure 2.19 (English query repeated); the following questions would arise:

- to which copy of *works_for* does "*works_for.co_id*" refer in the first line of the WHERE clause?
- to which copy of *works_for* does "*works_for.salary*" refer in the embedded SELECT clause?
- to which copy of *works_for* does "*works_for.co_id*" refer in the embedded WHERE clause?
- to which copy of *works_for* does "*works_for.salary*" refer just after the end of the embedded query?

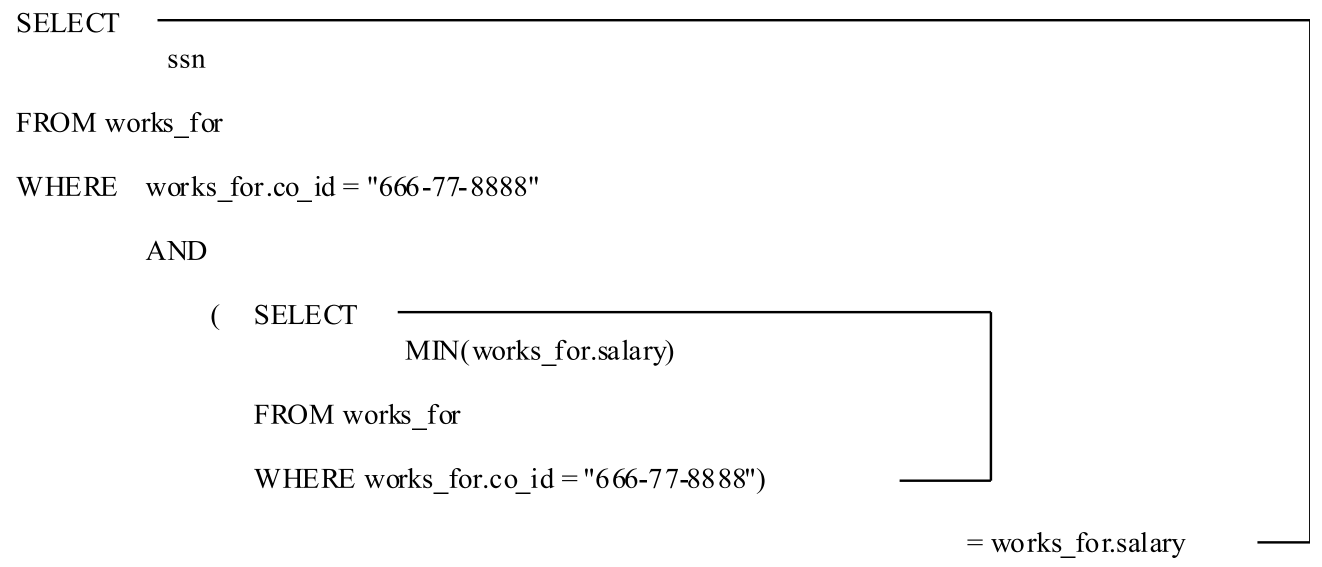
Find the person(s) who is(are) paid the lowest salary by the company whose govt_id is 666-77-8888

```
SELECT ssn
FROM works_for
WHERE works_for.co_id = "666-77-8888"
        AND ( SELECT MIN(works_for.salary)
                FROM works_for
                WHERE works_for.co_id = "666-77-8888")    = works_for.salary
```

Figure 2.19

The answers to all three question are simple once we consider the SFW statement's scope rules, viz.:

- •each SFW statement constitutes a name scope (block) which begins at the beginning of its SELECT clause and ends at the end of its last clause, which could be a FROM, WHERE, GROUP BY, HAVING, or ORDER BY clause
- •an SFW's FROM clause is the name declaration statement for the scope defined by the SFW statement
- •the scope of a declaration of a name -- in a FROM clause -- runs from the beginning to the end of the FROM clause's SFW statement, with breaks for the scopes of embedded declarations of the same name.



```
SELECT
          ssn
FROM works_for
WHERE   works_for.co_id = "666-77-8888"
        AND
            (   SELECT
                          MIN(works_for.salary)
                FROM works_for
                WHERE works_for.co_id = "666-77-8888")
                                                   = works_for.salary
```

Figure 2.20

The scopes of the two declarations of the name works_for are as shown in Figure 2.20, so that:

- •the occurrence of *works_for.co_id*" in the first line of the WHERE clause refers to the declaration of *works_for* in the outer SFW
- •the occurrence of *works_for* in the embedded SELECT clause refers to the declaration of *works_for* in the embedded SFW
- •the occurrence of *works_for* in the embedded WHERE clause refers to the declaration of *works_for* in the embedded SFW
- •the occurrence of *works_for* just after the end of the embedded query refers to the declaration of *works_for* in the outer SFW

Note that in the query of Figure 2.19 there were no references from the inner scope to names declared in the outer scope; the embedded SFW statement in Figure 2.19 is, therefore, termed an *uncorrelated* sub-query; being uncorrelated, it need be evaluated only once, rather than once per row of the table produced by the outer WHERE clause.

An example of a *correlated* sub-query, in fact examples of two correlated sub-queries, may be found in Figure 2.14b. In this case, the two sub-queries nested in the outer WHERE clause must be evaluated, and the comparison of their values done, once for each row of the *company* table, the table produced by the outer FROM clause.

Note, however, that, even though the number of Baltimore-headquartered divisions and the number of Saint Louis-headquartered divisions are computed for each company, these numbers cannot be passed from the inner SFW to the outer SFW as would be desired if we were interested in the results of the related query of Figure 2.21a. Rather, a sub-query occurs in a comparison, and all that can be passed up to the outer query is the Boolean result of the comparison. In the simplest SQL version of the query of Figure 2.21a tables are renamed; we therefore take a very small digression to discuss the subject of table renaming.

Find every company that has more divisions with headquarters in Baltimore than divisions with headquarters in Saint Louis, and for each such company find the number of Baltimore-headquartered divisions and the number of Saint Louis-headquartered divisions

Figure 2.21a

Renaming is done in the SFW statement by appending the desired new name, termed, appropriately, an *alias*, to the name of a table in the FROM clause. Thus, for example, the relational algebra query of Figure 1.20 (English query in Figure 1.6a) can be written in SQL as in Figure 2.21b.

```
SELECT skill1.ssn
FROM skill skill1, skill skill2
WHERE (skill1.co_id = 111-22-3333)
            AND (skill2.co_id = skill1.co_id)
            AND (skill1.ssn = skill2.ssn)
            AND (skill1.prod_id <> skill2.prod_id)
```

Figure 2.21b

If we wished to produced the results requested by the query of Figure 2.21a, we would have to write an SFW statement like that of Figure 2.21c, i.e., one in which a good deal of unnecessary additional work would be done. (For starters, a potentially vastly larger cartesian product would have to be computed.)

```
SELECT company.govt_id, COUNT(DISTINCT div1.div_name),
                    COUNT(DISTINCT div2,div_name)
FROM company, division div1, division div2
WHERE (  SELECT COUNT(division.div_name
                FROM division
                WHERE    (company.govt_id = division.co_id)
                              AND (division.div_name = div1.divname)
                              AND (div_hq = "Baltimore"))

                              >

            (  SELECT COUNT(division.div_name
                FROM division
                WHERE    (company.govt_id = division.co_id)
                              AND (division.div_name = div2.divname)
                              AND (div_hq = "Saint Louis"))
```

Figure 2.21c

We now return from the large digression, in which we have discussed the topics of scope and table renaming, and we resume our discussion of the embedding operator *EMPTY <SFW>*. Using our newly-acquired understanding of name scoping in SQL, we can write the query of Figure 2.18a as in Figure 2.22

```
SELECT division.div_hq
FROM division
WHERE    EMPTY (  SELECT site.loc
                            FROM site
                            WHERE (site.loc = "Atlanta")
                                          AND (site.co_id = division.co_id)
                                          AND (site.div_name = division.div_name))
```

Figure 2.22

The embedding expression *NOT EMPTY <SFW>*:

- •returns *false* if <SFW> returns an empty table, i.e., a table with zero rows
- •returns *true* otherwise.

Thus, the query of Figure 2.18a can be written, in yet another way, as in Figure 2.23

```
SELECT division.div_hq
FROM division
WHERE    NOT EMPTY (  SELECT site.loc
                                 FROM site
                                 WHERE (site.loc = "Atlanta")
                                            AND (site.co_id = division.co_id)
                                            AND (site.div_name = division.div_name))
```

Figure 2.23

The embedding expression *<value> <comparison op> ALL <SFW>*:

- •returns *true* if <SFW> returns a table with a single column and <value> bears the relation <comparison op> to the value in every row of that table
- •returns *false* if <SFW> returns a table with a single column and <value> does not bear the relation <comparison op> to the value in every row of that table, i.e., if there is at least one row to whose value <value> does not bear the relation <comparison op>
- •is a syntax error otherwise.

Thus, the query of Figure 2.24a can be written as in Figure 2.24b

Find every employee (of any company) who earns more than the highest-earning employee of the company with govt_id of 111-22-3333.

Figure 2.24a

```
SELECT works_for.ssn
FROM works_for
WHERE    work_for.salary >ALL (SELECT works_for.salary
                                          FROM works_for
                                          WHERE works_for.co_id = "111-22-3333)
```

Figure 2.24b

This query can, of course, also be written as in Figure 2.24c.

```
SELECT works_for.ssn
FROM works_for
WHERE    work_for.salary > (  SELECT MAX(works_for.salary)
                              FROM works_for
                              WHERE works_for.co_id = "111-22-3333)
```

Figure 2.24c

The embedding expression *<value> <comparison op> SOME <SFW>*:

- •returns *true* if <SFW> returns a table with a single column and <value> bears the relation <comparison op> to the value in at least one row of that table
- •returns *false* if <SFW> returns a table with a single column and <value> does not bear the relation <comparison op> to the value in any row of that table
- •is a syntax error otherwise.

Thus, the query of Figure 2.25aa can be written as in Figure 2.25b

Find every employee (of any company) who earns more than the at least one employee of the company with govt_id of 111-22-3333.

Figure 2.25a

```
SELECT works_for.ssn
FROM works_for
WHERE    work_for.salary >SOME (   SELECT works_for.salary
                                   FROM works_for
                                   WHERE works_for.co_id = "111-22-3333)
```

Figure 2.25b

This query can, of course, also be written as in Figure 2.25c.

```
SELECT works_for.ssn
FROM works_for
WHERE    work_for.salary > (  SELECT MIN(works_for.salary)
                              FROM works_for
                              WHERE works_for.co_id = "111-22-3333)
```

Figure 2.25c

The embedding expression *<value> <comparison op> ANY <SFW>* is identical to <value> <comparison op> SOME <SFW>.

2.2.3 The FROM Clause

At this point the reader, having seen only the form of the FROM clause shown in Figure 2.26, has, very possibly, come to the conclusion that it is the least complex of first three clauses of the SFW.

FROM $<table_1>$, $<table_2>$, ... , $<table_n>$

Figure 2.26

While this may actually be the case, we have not, by any means, finished our discussion of the FROM clause; for starters, exactly the same effect as is obtained by the FROM clause of Figure 2.26 is obtained by the FROM clause of Figure 2.27.

FROM $<table_1>$ CROSS JOIN $<table_2>$ ... CROSS JOIN $<table_n>$

Figure 2.27

For the reader who is puzzled as to why the cartesian product operator should have an alternative appellation as peculiar as CROSS JOIN, we note that *join*, denoted $\bowtie_{<condition>}$, or simply, $\bowtie$, is the name of an auxiliary relational algebra operator -- a combination of cartesian product and *select* (σ). Before we return to the question of the CROSS JOIN nomenclature, we consider the various the versions of the JOIN operation which are available in SQL.

In the most general case, $\bowtie_{<condition>}$, is termed *theta-join*; if <condition> is a conjunction of simple equality tests, then $\bowtie_{<condition>}$, is termed *equijoin*; without a subscripted condition, the operator $\bowtie$ is called *natural join*. The relational algebra expression of Figure 2.28a means exactly the same as that of Figure 2.28b.

$<table_1> \bowtie_{<condition>} <table_2>$

Figure 2.28a

$\sigma_{<condition>}(<table_1> \times <table_2>)$

Figure 2.28b

The expression of Figure 2.29, however, is quite something else.

$<table_1> \bowtie <table_2>$

Figure 2.29

Suppose that the manufacturing-industry database's scheme had been declared as in Figure 2.30, rather than as in Figure 1.1, i.e., with *similar* attributes always having the same name rather than sometimes having different names. (For example, with *govt_id* of *company* changed to *co_id* for uniformity with the other table schemes which have a *co_id* attribute.)

company(co_name, <u>co_id</u>, ceo_ssn, hq_loc)
division(<u>co_id,</u> <u>div_name</u>, subdiv_of, dir_ssn, div_hq)
site(<u>co_id,</u> <u>div_name, loc</u>)
product(<u>prod_id,</u> <u>co_id,</u> div_name, loc, prod_descr)
person(<u>ssn</u>, name, address)
works_for(<u>ssn,</u> <u>co_id</u>, div_name, salary, emp_id, sup_ssn)
skill(<u>ssn, prod_id,</u> <u>co_id)</u>

Figure 2.30

If a relational algebra query based upon the revised database scheme of Figure 2.30 began with, say, the three-way cartesian product of Figure 2.31a, it would likely, though not necessarily, continue with a *select* of the form shown in Figure 2.31b.

company $\times$ division $\times$ site

Figure 2.31a

```
σ
   (company.co_id = division.co_id)
      and(division.co_id = site.co_id)
      and(division.div_name = site.div_name)
```

Figure 2.31b

Natural join is a combination of *cartesian product* with a *select* whose condition specifies that all same-named attributes have equal values. The relational algebra expression which combines the cartesian product of Figure 2.31a with the select of Figure 2.31b could, thus, be written succinctly as in Figure 2.32.

$\text{company} \bowtie \text{division} \bowtie \text{site}$

Figure 2.32

Note, however, that, given the database scheme of Figure 1.1,the relational algebra expression of, for example, Figure 1.19, repeated here as Figure 2.33, would benefit little from the use of the *natural join* -- the reason that we didn't introduce the operator earlier.

```
π_company.govt_id(σ   [company × division × site × product × person × works_for × skill])
                       (company.govt_id = division.co_id)
                          and(company.hq_loc = "Atlanta")
                          and(division.div_hq = "Boston")
                          and(site.co_id = company.govt_id)
                          and(site.div_name = division.div_name)
                          and(site.loc = "Cleveland")
                          and(product.manuf_co = company.govt_id)
                          and(product.manuf_div = site.div_name)
                          and(product.prod_id = FA-S/2)
                          and(person.ssn = company.ceo_ssn)
                          and(person.name = "Joline Jones")
                          and(works_for.ssn = person.ssn)
                          and(works_for,co_id = company.govt_id)
                          and(works_for.salary > 500000)
                          and(skill.ssn = person.ssn)
                          and(skill.prod_id = product.prod_id)
                          and(skill.manuf_co = company.govt_id)
```

Figure 2.33

Thus far we have discussed only the relational algebra versions of equijoin and natural join. In SQL, *equijoin* can be written in the syntax of Figure 2.34a, in which case it has exactly the same meaning as the simpler FROM-WHERE of Figure 2.34b.

FROM <table$_1$> <alias$_1$> JOIN <table$_2$> <alias$_2$>
ON <alias$_1$>.<attribute> = <alias$_2$>.<attribute>

Figure 2.34a

FROM <table$_1$> <alias$_1$>, <table$_2$> <alias$_2$>
WHERE alias$_1$>.<attribute> = <alias$_2$>.<attribute>

Figure 2.34b

The *equijoin* operator can be also be written in the syntax of Figure 2.35a, in which case it has exactly the same meaning as the simpler FROM-WHERE of Figure 2.35b.

FROM <table$_1$> JOIN <table$_2$> USING <attribute>

Figure 2.35a

FROM <table$_1$> <alias$_1$>, <table$_2$> <alias$_2$>
WHERE alias$_1$>.<attribute> = <alias$_2$>.<attribute>

Figure 2.35b

Natural join can be written in the syntax of Figure 2.36a, in which case it has exactly the same meaning as the simpler FROM-WHERE of Figure 2.36b, except that the resulting table contains only one column for <attribute> rather than two identical columns -- on the assumption that <attribute$_1$>, <attribute$_2$>, ... , <attribute$_n$> are precisely those attributes which <table$_1$> and <table$_2$> have in common..

FROM <table$_1$> NATURAL JOIN <table$_2$>

Figure 2.36a

```
FROM <table_1> <alias_1>, <table_2> <alias_2>
WHERE   <(alias_1>.<attribute_1> = <alias_2>.<attribute_1>)
            AND (<alias_1>.<attribute_2> = <alias_2>.<attribute_2>
                        :
            AND (alias_1>.<attribute_n> = <alias_2>.<attribute_n>
```

Figure 2.36b

As the reader may have guessed by this point, the CROSS JOIN nomenclature was invented to distinuguish a simple cartesian product (CROSS JOIN) from JOIN ... ON, JOIN ... USING, and NATURAL JOIN.

The variants of *cartesian product* cum *select* which we have considered thus far aren't variants on the operations themselves, but, rather, variant notations for expressing what is already expressible using simple FROM and WHERE clauses. There are, however, three variant operations: LEFT JOIN, RIGHT JOIN, and FULL JOIN.

If the *cartesian product* of two tables is specified in a FROM clause, and those rows are *selected*, in the WHERE clause, for equality of an attribute(s) in the first table with an attribute(s) in the second table, then all information contained in rows of the first table which don't *match up with* any row of the second table is lost, as is all information contained in rows of the second table which don't *match up with* any row of the first table. When that information must be preserved, we use: the LEFT JOIN to preserve information from first-table rows that don't *match up*; RIGHT JOIN to preserve information from second-table rows that don't *match up*; FULL JOIN to preserve information from both first-table rows that don't *match up*, and second table rows that don't *match up*.

Consider the shortened and changed *company* and *division* tables of Figures 2.37a-b

co_name	**govt_id**	**ceo_ssn**	**hq_loc**
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	77-00-0007	Trenton
Southern Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	77-00-0007	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia

Figure 2.37a

co_id	div_name	subdiv_of	dir_ssn	div_hq
333-44-5555	axle	manufacturing	888-00-0006	Cambridge
333-44-5555	front axle	axle	777-00-0007	Mattapan
333-44-5555	rear axle	axle	777-00-0007	Roxbury
444-55-6666	communications	null	888-00-0010	West Chester
444-55-6666	purchasing	null	888-00-0011	Bensalem

Figure 2.37b

If we perform an *equijoin* of the two tables, on *ceo_ssn* of *company*, and *dir_ssn* of *division*, the the result is the table of Figure 2.38; all information is lost about the the company whose ceo has social security number 999-00-0001, and the company whose ceo has social security number 999-00-0004; all information is also lost about the the divisions whose directors have social security numbers 888-00-0006, 888-00-0010, and 888-00-0011.

co_name	govt_id	ceo_ssn	hq_loc	co_id	div_name	subdiv_of	dir_ssn	div_hq
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	front axle	axle	777-00-0007	Mattapan
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	rear axle	axle	777-00-0007	Roxbury
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	front axle	axle	777-00-0007	Mattapan
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	rear axle	axle	777-00-0007	Roxbury

Figure 2.38

The *left outer join* of the two tables, in the order *company*, *division*, contains all the information in, i.e., all the rows of, the result of the *equijoin* plus all company information for companies whose *ceo_ssn* weren't equal to the *dir_ssn* of any division -- with result-table rows for these companies *right-padded with* NULLs. The SFW for the *left outer join* is shown in Figure 2.39a, with its result in Figure 2.39b.

```
SELECT *
FROM company LEFT JOIN division ON company.ceo_ssn = division.dir_ssn
```

Figure 2.39a

co_name	govt_id	ceo_ssn	hq_loc	co_id	div_name	subdiv_of	dir_ssn	div_hq
New Jersey Tire	222-33 -4444	77-00-0007	Trenton	333-44-5555	front axle	axle	777-00-0007	Mattapan
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	rear axle	axle	777-00-0007	Roxbury
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	front axle	axle	777-00-0007	Mattapan
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	rear axle	axle	777-00-0007	Roxbury
New York Tire	111-22-3333	999-00-0001	New York	NULL	NULL	NULL	NULL	NULL
Southern Axle	333-44-5555	999-00-0001	Atlanta	NULL	NULL	NULL	NULL	NULL
Maryland Brake	555-66-7777	999-00-0004	Columbia	NULL	NULL	NULL	NULL	NULL

Figure 2.39b

The *right outer join* of the two tables, in the order *company*, *division*, contains all the information in, i.e., all the rows of, the result of the *equijoin* plus all division information for companies whose *ceo_ssn* weren't equal to the *dir_ssn* of any division -- with result-table rows for these companies *left-padded with* NULLs. The SFW for the *right outer join* is shown in Figure 2.40a, with its result in Figure 2.40b.

```
SELECT *
FROM company RIGHT JOIN division ON company.ceo_ssn = division.dir_ssn
```

Figure 2.40a

co_name	govt_id	ceo_ssn	hq_loc	co_id	div_name	subdiv_of	dir_ssn	div_hq
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	front axle	axle	777-00-0007	Mattapan
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	rear axle	axle	777-00-0007	Roxbury
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	front axle	axle	777-00-0007	Mattapan
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	rear axle	axle	777-00-0007	Roxbury
NULL	NULL	NULL	NULL	333-44-5555	axle	manufacturing	888-00-0006	Cambridge
NULL	NULL	NULL	NULL	444-55-6666	communications	null	888-00-0010	West Chester
NULL	NULL	NULL	NULL	444-55-6666	purchasing	null	888-00-0011	Bensalem

Figure 2.40b

The *full outer join* of the two tables, in the order *company*, *division*, contains all the information in, i.e., all the rows of, the result of the *equijoin* plus all company information for companies whose *ceo_ssn* weren't equal to the *dir_ssn* of any division, plus all division information for companies whose *ceo_ssn* weren't equal to the *dir_ssn* of any division -- with result-table rows for the additional rows padded with nulls to either the right or left as appropriate. The SFW for the *full outer join* is shown in Figure 2.41a, with its result in Figure 2.41b.

```
SELECT *
FROM company FULL JOIN division ON company.ceo_ssn = division.dir_ssn
```

Figure 2.41a

co_name	govt_id	ceo_ssn	hq_loc	co_id	div_name	subdiv_of	dir_ssn	div_hq
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	front axle	axle	777-00-0007	Mattapan
New Jersey Tire	222-33-4444	77-00-0007	Trenton	333-44-5555	rear axle	axle	777-00-0007	Roxbury
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	front axle	axle	777-00-0007	Mattapan
Atlanta Axle	444-55-6666	77-00-0007	Atlanta	333-44-5555	rear axle	axle	777-00-0007	Roxbury
New York Tire	111-22-3333	999-00-0001	New York	NULL	NULL	NULL	NULL	NULL
Southern Axle	333-44-5555	999-00-0001	Atlanta	NULL	NULL	NULL	NULL	NULL
Maryland Brake	555-66-7777	999-00-0004	Columbia	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	333-44-5555	axle	manufacturing	888-00-0006	Cambridge
NULL	NULL	NULL	NULL	444-55-6666	communications	null	888-00-0010	West Chester
NULL	NULL	NULL	NULL	444-55-6666	purchasing	null	888-00-0011	Bensalem

Figure 2.41b

2.2.4 The GROUP BY Clause

Up to this point we have had the luxury of relating SFW queries to queries in our extended version of relational algebra; since we didn't extend relational algebra with analogs to SQL's GROUP BY and HAVING, this is no longer the case. We therefore begin their discussion by considering their intended use: The GROUP BY clause, and, optionally, the HAVING clause, are used when the values of aggregate functions are required, but their values on multiple, non-overlapping, sub-tables of the table produced by the FROM clause rather than their values on the entire table produced by the FROM clause.

The GROUP BY clause partitions the table produced by the FROM and WHERE clauses; its presence in an SFW statement also changes the meaning of the SELECT clause. Whereas the SELECT clause of an SFW statement that doesn't have a GROUP BY clause produces exactly one line (row) of output, the SELECT clause of an SFW statement with a GROUP BY clause produces one line (row) per sub-table in the partition; in an SFW statement that doesn't have a GROUP BY clause, every aggregate function in the SELECT clause is applied to the entire table produced by the FROM and WHERE clauses; when a GROUP BY is present, every aggregate function in the SELECT clause is applied to each of the sub-tables in the partition of the table produced by the FROM and WHERE clauses.

The type of partitioning that can be effected by the GROUP BY clause, on the table produced by FROM and WHERE clauses, is partitioning of that table into sub-tables all of whose rows have the same values for some specified subset of the table's attributes. As an example, the SFW statement of Figure 2.42 would partition the company table of Figure 1.2a into the six sub-tables of Figures 2.43a-f -- assuming that the only rows of the tables are those rows explicitly shown in Figure 1.2a. These six tables would not, however, be the final result of the SFW statement; the final result would be the two-column table of Figure 2.44, i.e., one row for each of 2.43a-f.

```
SELECT hq_loc, COUNT(hq_loc)
FROM company
GROUP BY hq_loc
```

Figure 2.42

co_name	govt_id	ceo_ssn	hq_loc
New York Tire	111-22-3333	999-00-0001	New York

Figure 2.43a

co_name	govt_id	ceo_ssn	hq_loc
New Jersey Tire	222-33-4444	999-00-0002	Trenton

Figure 2.43b

co_name	govt_id	ceo_ssn	hq_loc
Southern Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Georgia Brake	666-77-8888	999-00-0005	Atlanta

Figure 2.43c

co_name	govt_id	ceo_ssn	hq_loc
Maryland Brake	555-66-7777	999-00-0004	Columbia

Figure 2.43d

co_name	govt_id	ceo_ssn	hq_loc
Florida Brake	777-88-9999	999-00-0006	Pensacola

Figure 2.43e

co_name	govt_id	ceo_ssn	hq_loc
Ohio Windshield	888-99-0000	999-00-0007	Toledo

Figure 2.43f

hq_loc	
New York	1
Trenton	1
Atlanta	3
Columbia	1
Pensacola	1
Toledo	1

Figure 2.44

In the SELECT clause of the SFW statement of Figure 2.42, the aggregate function COUNT is applied to attribute *hq_loc*, whose name happens the be the only unaggregated attribute name in the SELECT clause. In general, aggregate functions can be applied to one set of attributes while another set occurs unaggregated in the SELECT clause. Thus, for example, the query of Figure 2.45a can be written in SQL as shown in Figure 2.45b.

For each company, find the average salary paid by the company to its employees.

Figure 2.45a

```
SELECT co_id, AVG(works_for.salary)
FROM works_for
GROUP BY co_id
```

Figure 2.45b

While we can put an unaggregated attribute into the SELECT list of an SFW with a GROUP BY, it makes no sense for that unaggregated attribute not to be one of those on the GROUP BY's list, i.e., one of the partitioning attributes. The reason is that the SELECT is intended to produce one line of output for each sub-table in the partition, but there does not necessarily exist a single value, within any sub-table, for any attribute which is not on the GROUP BY's list of partitioning attributes. Consider, as an example, the (illegal) query of Figure 2.46. Exactly what value should be output as *ceo_ssn* for the sub-table whose *hq_loc* is "Atlanta?"

```
SELECT hq_loc, COUNT(hq_loc), ceo_ssn
FROM company
GROUP BY hq_loc
```

Figure 2.46

Notice that what we've said is that an unaggregated non-partitioning attribute may not appear on the SELECT list of an SFW that has a GROUP BY clause. We have not said that every partitioning attribute *must* appear on the SELECT list -- and there would be no good reason for such a constraint, though the beginning student of

SQL often mistakenly feel there is; thus, for example, the query of Figure 2.47 is perfectly legal -- though its output would probably be less than maximally useful as a division is identified by its *co_id* together with its *div_name*.

```
SELECT div_name, COUNT(subdiv_of)
FROM division
GROUP BY co_id, div_name
```

Figure 2.47

The query of Figure 2.47 would, by the way, partition the division table of Figure 1.2b into the sub-tables of Figures 2.48a-j.

co_id	div_name	subdiv_of	dir_ssn	div_hq
666-77-8888	innovation	null	888-00-0001	Valdosta

Figure 2.48a

co_id	div_name	subdiv_of	dir_ssn	div_hq
666-77-8888	research	innovation	888-00-0001	Boston

Figure 2.48b

co_id	div_name	subdiv_of	dir_ssn	div_hq
666-77-8888	development	innovation	888-00-0002	Nashville

Figure 2.48c

co_id	div_name	subdiv_of	dir_ssn	div_hq
111-22-3333	personnel	null	888-00-0003	New York

Figure 2.48d

co_id	div_name	subdiv_of	dir_ssn	div_hq
333-44-5555	manufacturing	null	888-00-0005	Boston

Figure 2.48e

co_id	div_name	subdiv_of	dir_ssn	div_hq
333-44-5555	axle	manufacturing	888-00-0006	Cambridge

Figure 2.48f

co_id	div_name	subdiv_of	dir_ssn	div_hq
333-44-5555	front axle	axle	888-00-0007	Mattapan

Figure 2.48g

co_id	div_name	subdiv_of	dir_ssn	div_hq
333-44-5555	rear axle	axle	888-00-0008	Roxbury

Figure 2.48h

co_id	div_name	subdiv_of	dir_ssn	div_hq
444-55-6666	communications	null	888-00-0010	West Chester

Figure 2.48i

co_id	div_name	subdiv_of	dir_ssn	div_hq
444-55-6666	purchasing	null	888-00-0011	Bensalem

Figure 2.48j

2.2.5 The HAVING Clause

When the values of aggregate functions are required for some, but not all, sub-tables produced by a GROUP BY, the desired sub-tables can be specified through the use of a HAVING clause. The reader will recall that the WHERE clause in an SFW statement contains a condition which specifies which rows of a table are desired; In similar fashion, the HAVING clause contains a condition which specifies which sub-tables of a partition are desired. Just as the condition in a WHERE clause must be appropriate for application to a row, the condition in a HAVING clause must be appropriate for application to a (sub-)table. It thus makes sense for a HAVING clause condition to involve the application of aggregate functions to a sub-table; it doesn't, on the other hand, make sense for such a condition to refer to the unaggregated value of an attribute unless that attribute has the same value for all rows of the sub-table, i.e., unless it is one of the partitioning attributes in the GROUP BY clause.

As a very simple example of the use of the HAVING clause, consider the query of Figure 2.49a, and the corresponding SFW of Figure 2.49b which, applied to the database instance of Figures 1.2a-f, produces the result shown in Figure 2.49c.

Find every city in which three or more companies have their headquarters.

Figure 2.49a

```
SELECT company.hq_loc
FROM company
GROUP BY company.hq_loc
HAVING COUNT(company.hq_loc) >= 2
```

Figure 2.49b

hq_loc
Atlanta

Figure 2.49c

A more complex example of the use of the HAVING clause may be found in Figure 2.50a-b, with its result in Figure 2.50c.

Find the name and social security number of the ceo of every company that has two or more first-level subdivisions, i.e., two or more subdivisions with *subdiv_of* values of *null*.

Figure 2.50a

```
SELECT DISTINCT person.name, company.ceo_ssn
FROM company, division, person
WHERE (company.govt_id = division.co_id)
        AND (company.ceo_ssn = person.ssn)
        AND (division.subdiv_of IS NULL)
GROUP BY company.govt_id, company.ceo_ssn, person.name
HAVING COUNT(division.subdiv_of) >= 2
```

Figure 2.50b

person.name	company.ceo_ssn
Joseph Jones	999-00-0001
Donald Smith	999-00-0003

Figure 2.50c

Before we complete our discussion of the HAVING clause with an explanation of the type of condition which is inappropriate there, we note that the HAVING clause is not, strictly, speaking, necessary to maintain the expressive power of the SFW; rather, for any SFW with a HAVING clause, there is an equivalent SFW without one. For example, the SFW of Figure 2.50d is equivalent to that of Figure 2.50b.

```
SELECT DISTINCT person.name, company.ceo_ssn
FROM company, division div1, person
WHERE (company.govt_id = div1.co_id)
            AND (company.ceo_ssn = person.ssn)
            AND (div1.subdiv_of IS NULL)
            AND ( SELECT COUNT(div2.subdiv_of)
                      FROM division div2
                      WHERE (div2.co_id = div1.co_id)
                                    AND (div2.subdiv_of IS NULL))   = 2
GROUP BY company.govt_id, company.ceo_ssn, person.name
```

Figure 2.50d

As an example of an SFW statement whose HAVING clause makes no sense, consider that of Figure 2.51.

```
SELECT company.hq_loc
FROM company
GROUP BY company.hq_loc
HAVING  COUNT(division.subdiv_of) >= 2
              AND (ceo_ssn = "999-00-0006")
```

Figure 2.51

The problem is, of course, that, in general, a sub-table produced by the GROUP BY clause will have rows with different values for *ceo_ssn*, so that *ceo_ssn = "999-00-0006* is not a condition of the appropriate *type* to apply to an entire sub-table.

Recall that, as indicated above, a HAVING clause condition may refer to an unaggregated attribute so long as that attribute is one of the partitioning attributes listed in the GROUP BY clause. Consider, as an example, the SFW statement of Figure 2.52a, which arises from the query of Figure 2.52b.

```
SELECT DISTINCT person.name, company.ceo_ssn
FROM company, division, person
WHERE (company.govt_id = division.co_id)
            AND (company.ceo_ssn = person.ssn)
            AND (division.subdiv_of = NULL)
GROUP BY company.govtid, company.ceo_ssn, person.name
HAVING  COUNT(division.subdiv_of) >= 2
              AND (person.name LIKE 'K%')
```

Figure 2.52a

Find the name and social security number of the ceo of every company that has two or more first-level subdivisions, i.e., two or more subdivisions with *subdiv_of* values of *null* -- and whose name starts with a *K*.

Figure 2.52b

Finally, a HAVING clause may, in fact, contain references to *only* unaggregated attributes, i.e., it may contain no applications of aggregate functions whatsoever, as in Figure 2.53a, which arises from the query of Figure 2.53b. If it does, however, it can always be written as a cheaper-to-evaluate SFW statement without GROUP By and HAVING clauses, as in Figure 2.53c.

```
SELECT person.name, company.ceo_ssn
FROM company, person
WHERE company.ceo_ssn = person.ssn
GROUP BY company.hq_loc, company.ceo_ssn, person.name
HAVING company.hq_loc = "Atlanta"
```

Figure 2.53a

Find the name and social security number of the ceo of every company whose headquarters are located in Atlanta.

Figure 2.53b

```
SELECT person.name, company.ceo_ssn
FROM company, person
WHERE    (company.ceo_ssn = person.ssn)
              AND (company.hq_loc = "Atlanta")
```

Figure 2.53c

2.2.6 The ORDER BY Clause

The final clause in the SELECT-FROM-WHERE statement, the ORDER BY clause performs the function of ordering, i.e., sorting, the rows of the table produced by the FROM, WHERE (if present), GROUP BY (if present), HAVING (if present), and SELECT clauses as desired. Ordering is, of course, by the values of one or more attributes, and may be by ASCending or DESCending value, ASCending being the default.

Consider, as an example, the *company* table of Figure 1.2a, repeated here as Figure 2.54. If we wanted a version of the table with the values of *co_name* in ASCending (alphabetical) order, as shown in Figure 2.55, we could do so by executing the SFW statement of Figure 2.56; the same table, but with values of *co_name* in descending order, as shown in Figure 2.57, could be achieved by executing the SFW statement of Figure 2.58.

co_name	govt_id	ceo_ssn	hq_loc
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	999-00-0002	Trenton
Southern Axle	333-44-5555	999-00-0001	Atlanta
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Ohio Windshield	888-99-0000	999-00-0007	Toledo

Figure 2.54

co_name	govt_id	ceo_ssn	hq_loc
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia
New Jersey Tire	222-33-4444	999-00-0002	Trenton
New York Tire	111-22-3333	999-00-0001	New York
Ohio Windshield	888-99-0000	999-00-0007	Toledo
Southern Axle	333-44-5555	999-00-0001	Atlanta

Figure 2.55

```
SELECT *
FROM company
ORDER BY co_name
```

Figure 2.56

co_name	govt_id	ceo_ssn	hq_loc
Southern Axle	333-44-5555	999-00-0001	Atlanta
Ohio Windshield	888-99-0000	999-00-0007	Toledo
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	999-00-0002	Trenton
Maryland Brake	555-66-7777	999-00-0004	Columbia
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Atlanta Axle	444-55-6666	999-00-0003	Atlanta

Figure 2.57

```
SELECT *
FROM company
ORDER BY co_name DESC
```

Figure 2.58

Finally, if the original *company* table had entries for different companies with the same name, as does the modified *company* table of Figure 2.59, then we might want a version ordered by *co_name* and secondarily ordered by *hq_loc*, i.e., the version pictured in Figure 2.60. This is readily accomplished through the execution of the SFW statement of Figure 2.61.

co_name	govt_id	ceo_ssn	hq_loc
New Jersey Tire	222-77-7777	999-88-7777	Newark
Southern Axle	555-98-7654	999-33-0004	Apalachicola
Southern Axle	333-12-3456	999-11-0007	Nashville
Southern Axle	333-44-5555	999-00-0001	Atlanta
Ohio Windshield	888-99-0000	999-00-0007	Toledo
New York Tire	111-22-3333	999-00-0001	New York
New Jersey Tire	222-33-4444	999-00-0002	Trenton
Maryland Brake	555-66-7777	999-00-0004	Columbia
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Atlanta Axle	444-55-6666	999-00-0003	Atlanta

Figure 2.59

co_name	govt_id	ceo_ssn	hq_loc
Atlanta Axle	444-55-6666	999-00-0003	Atlanta
Florida Brake	777-88-9999	999-00-0006	Pensacola
Georgia Brake	666-77-8888	999-00-0005	Atlanta
Maryland Brake	555-66-7777	999-00-0004	Columbia
New Jersey Tire	222-77-7777	999-88-7777	Newark
New Jersey Tire	222-33-4444	999-00-0002	Trenton
New York Tire	111-22-3333	999-00-0001	New York
Ohio Windshield	888-99-0000	999-00-0007	Toledo
Southern Axle	555-98-7654	999-33-0004	Apalachicola
Southern Axle	333-44-5555	999-00-0001	Atlanta
Southern Axle	333-12-3456	999-11-0007	Nashville

Figure 2.60

```
SELECT *
FROM company
ORDER BY co_name, hq_loc
```

Figure 2.61

2.3 The CREATE VIEW Statement

As was indicated above, the **CREATE VIEW** statement, which allows the user to store and name the value of an SQL expression, with attributes renamed, if desired, is probably SQL's closest construct to an assignment statement. As an example of its use, suppose that we wanted a table of people's skills together with their names and addresses and with the sites at which companies they work for manufacture the products in whose manufacture they are skilled (trained). The **CREATE VIEW** statement of Figure 2.62 produces just such a table, a table named *skills_and_locations*. The new table can, of course, be queried, through the use of SFW statements, via the table name *skills_and_locations*. The question of updating a view, i.e., a table created through the use of a **CREATE VIEW** statement is a complex one; The only comment appropriate here, as we have not paid much attention to the updating of base tables, i.e., tables created through the use of **CREATE TABLE** statements, is that permissible updates to views are extremely tightly restricted, as one might expect, because of their potentially ambiguous effects on the base tables from which they arise.

```
CREATE VIEW skills_and-locations
AS SELECT person.ssn, person.name, person.address, works_for.co_name,
                product.prod_id, product.loc
FROM skill, person, works_for, product
WHERE (skill.ssn = person.ssn)
              AND (person.ssn = works_for.ssn)
              AND (skill.manuf_co = product.manuf_co)
```

Figure 2.62

References

•Standards Documents

••International Standards Organization, *Database Language SQL*, (ISO 9075:1987(E)), 1987
••International Standards Organization, *Database Language SQL*, (ISO 9075:1989(E)), 1989
••International Standards Organization, *Database Language SQL*, (ISO 9075:1992(E)), 1992 (also available as ANSI X3.135-1992)

•Textbooks (on relational database systems)

••Elmasri, Ramez, and Shamkant B. Navathe, *Fundamentals of Database Systems, Second Edition*, The Benjamin/Cummings Publishing Company, Inc., 1994. This text, written for the average student of computer science, has a short, but adequate, introductory chapter on SQL.
••Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan, *Database System Concepts, Third Edition*, The McGraw-Hill Companies, Inc., 1997. This text, written for the average student of computer science, has a short, but adequate, introductory chapter on SQL.
••Ullman, Jeffrey D., *Principles of Database and Knowledge-Base System: Volume I: Classical Database Systems; Volume II: The New Technologies*, Computer Science Press, 1988. This text (Volume I), written for more formally-oriented students of computer science, has a short, but adequate, introductory chapter on SQL.
••Ullman, Jeffrey D., and Jennifer Widom, *A First Course in Database Systems*, Prentice Hall, 1997. This text, written for the average student of computer science, but also readable by students in management or business programs, has an extensive introduction to SQL.
••Vossen, Gottfried, *Database Models, Database Languages, and Database Management Systems*, Addison Wesley, 1991. This text, written for more formally-oriented students of computer science, has a better than average chapter on SQL.
••Connolly, Thomas, Carolyn Begg, and Anne Strachan, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Addison Wesley, 1995. This text, written for the more practically-

oriented student of computer science, information science, management, or business, has an extensive introduction to SQL.

••SQL-Specific Books for Practitioners

•Date, C.J., and H. Darwen, *A Guide to the SQL Standard, third edition*, Addison Wesley, 1993
•Cannan, Stephen, and Gerard Otten, *SQL - the Standard Handbook*.,McGraw-Hill,1992

•Celko, Joe, *SQL for Smarties: Advanced SQL Programming*, Morgan-Kauffman Publishing

•Gruber, Martin, and Joe Celko, SQL instant reference, SYBEX Inc., 1993

•Melton, Jim, and Alan R. Simon, *Understanding the New SQL: A Complete Guide*., Morgan-Kauffman, 1992