1 Queries and the SELECT statement

When statisticians analyze data they often look for differences between groups. For example, quality control experts might compare the yield of a manufacturing process under different operating constraints; clinical trial statisticians examine the effect on patient health of a new drug in comparison to a standard; and market researchers might study inventory and sales at different locations in a large retail chain. These data-analysis activities require reduction of the data, either by subsetting, grouping, or aggregation. A query language allows a user to interactively interrogate the database to reduce the data in these ways and retrieve the results for further analysis.

We focus on one particular query language, the Structured Query Language (SQL), an ANSI (American National Standards Institute) standard. SQL works with many database management systems, including Oracle, MySQL, and Postgress. Each database program tends to have its own version of SQL, possibly with proprietary extensions, but to be in compliance with the ANSI standard, they all support the basic SQL statements.

The SQL statement for retrieving data is the SELECT statement. With the SELECT statement, the user specifies the table she wants to retrieve. That is, a query to the database returns a table. The simplest possible query is

```
SELECT * FROM Chips;
```

This SELECT statement, gives us back the entire table, Chips (Figure **??**), found in the database, all rows and all columns. Note that we display SQL commands in all capitals, and names of tables and variables are shown with an initial capital and remaining letters in lower case. As SQL is not case sensitive, we use capitalization only for ease in distinguishing SQL keywords from application specific names. The * refers to all columns in the table.

The table returned from a query may be a subset of tuples, a reduction of attributes, or a more complex reduction of a table in the database. It

	Date	Transistors	Microns	ClockSpeed	DataWidth	Mips
8080	1974	6000	6.00	2.0	8	0.64
8088	1979	29000	3.00	5.0	16	0.33
80286	1982	134000	1.50	6.0	16	1.00
80386	1985	275000	1.50	16.0	32	5.00
80486	1989	1200000	1.00	25.0	32	20.00
Pentium	1993	3100000	0.80	60.0	32	100.00
PentiumII	1997	7500000	0.35	233.0	32	300.00
PentiumIII	1999	9500000	0.25	450.0	32	510.00
Pentium4	2000	42000000	0.18	1500	32	1700.00

Figure 1: The data frame called Chips gives data on the CPU development of PCs over time. The processor names serve as the data frame row names. The variables are Date, Transistors, Microns, ClockSpeed, Datawidth, and Mips. Data from .....

may even be formed by a combination of tables in the database. In this section, we examine how to form queries that act on one table. Section **??** addresses queries based on multiple tables.

The direct analogy of the data frame to the database table made in the previous section, helps us understand the subsetting capabilities in the query language. The S language has very powerful subsetting capabilities in part because it is an important aspect of data analysis. Just as a subset of a data frame returns a data frame, a query to subset a table in a database returns a table. The square brackets [ ] form the fundamental subsetting operator in the S language. (These are covered in detail in Chapter **??**.) We focus here on those aspects that are closest to the SQL queries. Recall that we can select particular columns or variables by name. For example, in the *Chips* data frame, to grab the two variables Microns and Mips we use a vector containing these column names,

```
Chips[ c("Mips", "Microns") ]
```

Notice that the order of the variable names in the vector determines the order that they will be returned in the resulting data frame. If *Chips* were

a table in a database then the SQL query to obtain the above subset would be:

```
SELECT Mips, Microns FROM Chips;
```

To form a subset containing particular cases from a data frame, we may provide their row names. The following example, retrieves a data frame of Microns and Mips for the Pentium processors:

```
Chips[c("Pentium", "PentiumII", "PentiumIII", "Pentium4"),
        c("Mips", "Microns") ]
```

The resulting data frame is:

	Mips	Microns
Pentium	100.00	3100000
PentiumII	300.00	7500000
PentiumIII	510.00	9500000
Pentium4	1700.00	42000000

The equivalent SQL query to obtain the above subset would be:

```
SELECT Microns, Mips FROM Chips
       WHERE Processor = 'Pentium' OR Processor = 'PentiumII'
        OR Processor = 'PentiumIII' OR Processor = 'Pentium4';
```

A clearer way to express this query is with the IN keyword:

```
SELECT Microns, Mips FROM Chips
       WHERE Processor IN
        ('Pentium', 'PentiumII', 'PentiumIII', 'Pentium4');
```

Now that we have introduced a couple of examples, we present the general syntax of a SELECT statement:

```
SELECT column(s) FROM relation(s) [WHERE constraints];
```

The column(s) parameter in the SELECT statement above may be a comma-separated list of attribute names, an * to indicate all columns, or aggregate function such as MIN(Microns). We discuss aggregate functions in Section **??**.

The relation(s) parameter provides the name of a single relation (table) or a comma separated list of tables (see Section **??**). The WHERE clause is optional; it allows you to identify a subset of tuples to be included in the resulting relation. That is, the WHERE clause specifies the condition that the tuples must satisfy to be included in the results. For example, to pull all 32-bit processors that execute fewer than 250 million instructions per second, we select the tuples as follows,

```
SELECT * FROM Chips
              WHERE Mips < 250 AND DataWidth = 32;
```

The [ ] operator in S can similarly use logical vectors to subset the data frame,

```
Chips[ Chips["Mips"] < 250 & Chips["DataWidth"] == 32, ]
```

1.1 Functions

SQL is not a computational language nor is it a statistical language. It offers limited features for summarizing data. Basically, SQL provides a few aggregate functions that operate over the rows of a table, and some mathematical functions that operate on individual values in a tuple. Aside from the basic arithmetic functions of + - * and /, all other mathemtical functions are product specific. MySQL provides a couple dozen functions including ABS, CEILING, COS, EXP, LOG, POWER, and SIGN. The aggregate functions available are:

- COUNT - the number of tuples
- SUM - the total of all values for an attribute

- AVG - the average value for an attribute
- MIN - the minimum value for an attribute
- MAX - the maximum value for an attribute

With the exception of COUNT, these aggregate functions first discard NULLs, then compute on the remaining known values. Finding other statistical summaries, especially rankings, is no simple task to accomplish in SQL. We visit this problem in Section **??**.

1.2 Additional clauses

The GROUP BY clause makes the aggregate functions in SQL more useful. It enables the aggregates to be applied to subsets of the tuples in a table. That is, grouping allows you to gather rows with a similar value into a single row and to operate on them together. For example, in the inventory exercise, if we wanted to find the total sales for each region, we would group the tuples by region as follows,

```
SELECT Region, SUM(Amount) FROM Sales GROUP BY Region;
```

This functionality parallels the **tapply** function in S. Unfortunately, the WHERE clause can not contain an aggregate function, but the HAVING clause can be used to refer to the groups to be selected. The syntax for the HAVING function is:

```
SELECT Region, SUM(Amount) FROM Sales GROUP BY Region
      HAVING SUM(Amount) > 100000;
```

A few other predicates and clauses that may prove helpful are DISTINCT, NOT, and LIMIT. Briefly, the LIMIT clause, limits the number of tuples returned from the query. The NOT predicate negates the conditions in the WHERE or HAVING clause, and the DISTINCT keyword forces the

values of an attribute in the results table to have unique values. The following SELECT statement demonstrates all three. Ignoring the LIMIT clause at first, the results table consists of one for each state that has a store not in the eastern or western regions. The LIMIT clause provides a subset of size 10 from this results table.

```
SELECT DISTINCT State FROM Sales
       WHERE NOT Region IN ('East','WEST')
       LIMIT 10;
```

Another useful command is ORDER BY. According to Celko [**?**], it is commonly believed that ORDER BY is a clause in the SELECT statement. However, it belongs to the host language, meaning that the SQL query, without the ORDER BY clause, is executed, and the host language then orders the results. This may lead to misleading results. For example, in the query below it appears that the seven locations with the highest sales amounts will form the results table. However, the ORDER BY is applied after the results table is formed, meaning that it will simply order the first seven tuples in the results table.

```
SELECT Location, Amount FROM Sales
   ORDER BY Amount DESC LIMIT 7;
```

Note that the default ordering is ascending, and results can be ordered by the values in more than one attribute by providing a comm separated list of attributes. The DESC keyword reverses the ordering, it needs to be provided for each attribute that is to be put in descending order.

1.3 Summary

Briefly the order of execution of the clauses in a SELECT statement is as follows:

1. FROM: The working table is constructed.

2. WHERE: The WHERE clause is applied to each tuple of the table, and only those rows that test TRUE are retained.

3. GROUP BY: The results are broken into groups of tuples all with the same value of the GROUP BY clause, and each group is reduced to a single tuple.

4. HAVING: The HAVING clause is applied to each group and only those that test TRUE are retained.

5. SELECT: The attributes not in the list are dropped and options such as DISTINCT are applied.