

10

Operating on sets of data items as members of a class

Search Sets

Defining Sets of Records

It's possible to define records in QueryCalc as sets on the basis of common attributes. This is one of the more powerful features of QueryCalc, and yet quite simple to use. You will find search sets useful for a variety of reasons, but search sets are particularly valuable for (1) statistical and financial analyses, and (2) high-speed generic searches. Using search sets, employees in a database can be divided and subdivided into classes for purposes of insurance risk analyses. Or school children can be broken into economic and ethnic classes for the purposes of demographic analyses. And manufacturing processes can be subdivided into distinct classes for purposes of reliability and yield analyses.

If you think of a class—as in a class-action lawsuit—every member of the class shares some common property. In the case of a class-action lawsuit, every member of the class has been adversely impacted by some common action. It is the presence of a common attribute that defines a class. A second class drawn from the same community of members would be defined by another feature. Two such classes could potentially have (1) a few members in common (*intersecting sets*), (2) have no members in common (*mutually exclusive sets*), or (3) one class could be totally contained within the other (a *proper subset*). Membership in one class does not preclude membership in another. Any one individual could be potentially be a member of an infinite number of sets.

Imagine the set, which we'll call Set A, of all people who received at least one paycheck in 1993. In QueryCalc, the set is created through the use of a modified query question:

```
@using payrecord, store in !a emp-num  
when date ib 930101,931231
```

Set A is denoted as *!a* in QueryCalc (as always, capitalization is unimpor-

tant). The set that is created by this query question will be a list of employee-id numbers. Most of the people in this class will, of course, have received a number of paychecks during the year, but it is the nature of the @STORE query question to *eliminate all duplicate entries* once the search set has been built. The process of duplicate elimination is accomplished by sorting the retrieved search values and recording only those which are different. Thus, when completed, Set A will contain only one id-number for each employee who received at least one paycheck in 1993, and by consequence of the duplicate-elimination algorithm used, the set will be presented in a quasi-ascending order.

Twenty-six such sets may be created, labeled A to Z. The sets may be reused and redefined any number of times during the course of the execution of a file.

Using the Search Set

The use of a defined set in QueryCalc is straightforward. If we wish to determine the average number of dependents that Set A has, the query question would be:

```
@using employees, avg of numdeductions
when emp-num=!a
```

The query process proceeds in the following manner. The "pattern" to be matched (an employee number in this case) is neither "hard-coded" into the query question nor taken off of the spreadsheet. Rather, the first employee number is taken from Set A's list and a keyed search is performed on that particular value. As normally done, all of the statistics appropriate to the query question are calculated. Once the end of the search chain has been reached, the next search value is taken from the list and its chain is searched. The calculated statistics accrue until all of the items' records in Set A have been searched. In this manner, the average of the deductions in Set A will be calculated as a class.

The Rules

There are a few rules which govern the use of sets in QueryCalc. They are:

1. The search sets contain lists of search item *values*, not record numbers. Thus a search set belongs to no single dataset or database and may be used in any variety of database environments, so long as the dataitem type and length are identical. The dataitem name is unimportant.

2. A set may be formed using any dataitem (keyed or not), but when the set is used for pattern matching, the specified dataitem must be a search item in the dataset to be searched. This rule exists for purposes of speed and computational efficiency.
3. Only one dataitem may be recorded in a search item list. This rule is in force because the sets will only be used against one search item at a time.

A Quick Exercise

To demonstrate the power of search sets, please type the following:

```
:hello user.aics,qcdemo
:run qc.qcprogs.aics
```

You are now in QueryCalc. Move to cell B3, and type:

```
@opendb qcdemo/FRONT
@using employees, store in !b socsecnum
  when zip is 88047!
```

The @STORE query question creates set B, which will be a list of the people who live in a certain community. There will be five people who meet this criterion. To see the list, type:

```
@show !b
```

To use the list to determine the gross wages paid to this particular set of people in 1984, move to cell B4 and type:

```
@using payrecord, sum of gross when
  socsecnum is !b and date ib 840101,841231!
```

To calculate the average number of regular hours each of these people worked, using a different dataset, move to cell B5 and type:

```
@using labor, avg of regular when
  socsecnum=!b!
```

Defining a Set Using Multiple Datasets, Databases, or Even Multiple HP3000's

The formation of a set of search item values may often require extracting information from several different datasets, perhaps located in different databases or on different HP3000s. It's quite possible to do this, but it's done one dataset at a time, sequentially. The method used is that a set of search values defined from one dataset are applied to a second dataset, where the set is redefined and further qualified. In the example shown here, three databases are used. The first database is an MPE flat file, the second a KSAM file, and the third an IMAGE database. Set A will not be completely defined until the third query question has been executed.

```
@using mpedb.employees, store in !a idnumber
      when startyear ib 1980,1984
```

```
@using ksamdb.insurance, store in !a idnumber
      when idnumber=!a and disability=Y
```

```
@using imagedb.accidents, store in !a emp-number
      when emp-number=!a and accident-type>3
      and accident-date ib 19900101,19901231
```

The Set A created by this series of query questions is a list of id-numbers for those people who started with the company in the years 1980 to 1984 (information found only in the first dataset), who have disability insurance (information found only in the second dataset) and who suffered an accident greater than type 3 in a particular range of dates (information found only in the third dataset).

Notice that in the third query question, the employee id-number has a different name, yet the search list can still be used. That's because the search list contains only *values*. If the search item is of the same data type and length, you can use the search list, without regard to the item's name.

When qualifying a search list from multiple datasets and databases, you will find it most efficient to execute your @STORE query questions in that order which will eliminate as many search entries as possible in the first query, and then in the second, and so on. The only exception to this general rule is when the dataitem to be @STOREd is not a search item in one of the datasets. That dataset will always have to be the first dataset to be searched.

Nested Subsets

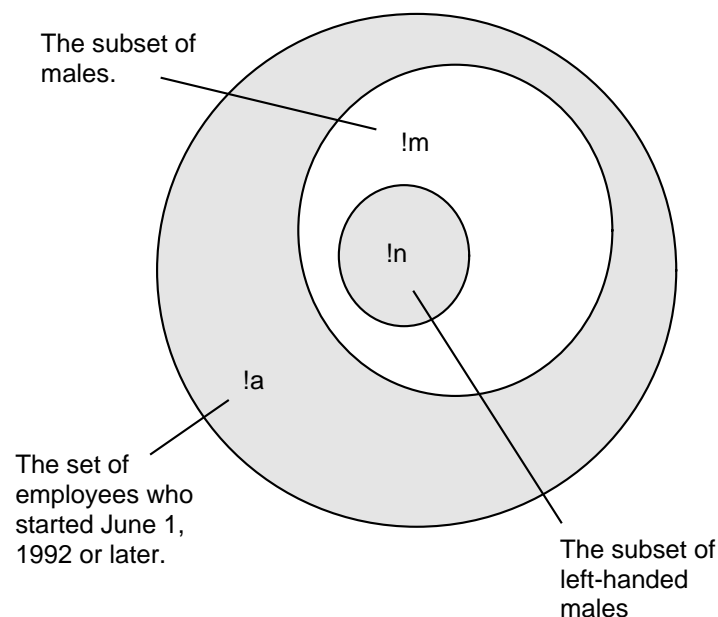
Nested subsets are created in an identical manner. It is quite easy to break a population of entries into a series of hierarchically-organized subsets for further statistical analyses. For example:

```
@using db1.employees, store in !a idnumber
    termdate<999999 and startdate>=920601
```

```
@using db1.employees, store in !m idnumber
    when sex is M and idnumber=!a
```

```
@using db1.employees, store in !n idnumber
    when idnumber is !m and right-handed is N
```

These three @STORE query questions create three sets. Set M is a subset of the male employees, drawn from the list of people who began work on or after June 1, 1992 and continue to work now (Set A). The second @STORE creates set N, a subset of the set M, which is composed of only those males who are left-handed and meet the criteria of set A. By breaking an original population into these and similar smaller groups, you are offered the possibility of performing high-speed, repeated statistical or financial analyses on selected groups. An example of such an analysis appears on the following page.



An Example of a Nested Subset Analysis

The following analysis is an example drawn from real data and a program in actual use. The analysis represents a general estimate of the surgical risk associated with coronary arterial bypass-graft surgeries (CABGRISK) when broken into several categories of risk factors and intra- and post-operative complications.

The analysis begins on Page A with the creation of nine sets, labeled A through I. Because the data must be extracted from a variety of datasets, a series of query questions must be used. Set A isolates all of the patient numbers who had operations in the specified range of dates (cell Ad7). Set B is formed by using Set A's list of patient id-keys to determine those patients who had the proper form of operation while eliminating any patients which had complicating factors such as vascular disease (vsd) or additional valve surgery (cell Ad8). As occasionally happens, the length of the query question necessary to form Set B exceeds the 187-character limit of the cell. Thus, a second set, C, is formed as a subset of Set B, further refining the list of qualifying patient numbers (cell Ad9). This is not a particularly efficient procedure, but it is sometimes necessary.

The @STORE equations for Page A are shown on the opposite page. The actual number of items each equation generates are shown below.

Risk factors in females having CABG

From:1981

To:1983

Date select List A:	1,407.00
CABG-I List B:	846.00
CABG-II List C:	837.00
Female List D:	164.00
Age > 60 List E:	105.00
Obesity List F:	43.00
Diabetes List G:	12.00
Hypertension List H:	10.00
Smoker List I:	0.00

The output that results from the construction of the nested subsets on Page A of the CABGRISK report.

A		_____a_____		_____b_____		_____c_____		_____d_____		_____e_____		_____f_____
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												

Equations for Page A

- (Ad4): 1981
- (Ad5): 1983
- (Ad7): @Using cardio.name, store in !a id-key when surg-date
ib [d4*10000],[d5*10000]
- (Ad8): @Using cardio.op, store in !b id-key when id-key=!a and
\coro-art-surg\=Y and \resection-va\<>Y and \repair-vsd\<>Y and
\valve-surg\<>Y and \cong-surg\<>Y and \thoracic-surg\<>Y
- (Ad9): @Using cardio.op, store in !c id-key when id-key=!b and
\peri-vasc-surg\<>Y and \graft-rep\<>Y
- (Ad10): @Using cardio.name, store in !d id-key when id-key=!c
and sex=F
- (Ad11): @Using cardio.name, store in !e id-key when id-key=!d
and age-operated-on>=6000
- (Ad12): @Using cardio.preop, store in !f id-key when id-key=!e
and obesity=Y
- (Ad13): @Using cardio.preop, store in !g id-key when id-key=!f
and diabetes=Y
- (Ad14): @Using cardio.preop, store in !h id-key when id-key=!g
and hypertension=Y
- (Ad15): @Using cardio.preop, store in !i id-key when id-key=!h
and smoker=Y

The equations of Page A necessary to create the nine hierarchically nested subsets used in the CABGRISK report.

Surgical Risk Analysis by Class (Continued)

The use of the defined sets in the actual cardiovascular surgical risk analysis appears on Page B of the report. Five of the eight data columns of the report are shown below. A portion of the equations which comprise Page B are shown on the opposite page.

The first column (CABG ALL) refers to the patients of Set C. The next column (Females) references those in Set D. The following columns use the data in Sets E, F, and G respectively. The columns using Sets H and I, as well as some additional summary statistics, occur off-page.

	CABG ALL	Females	Females over 60	Females >60, obese	Females >60, obese diabetic
Count and Percentage	-----	-----	-----	-----	-----
	837	164	105	43	12
	100.0%	19.6%	12.5%	5.1%	1.4%
Intra-operative myocardial infarction	19 2.3% 2.3%	5 3.0% 0.6%	4 3.8% 0.5%	1 2.3% 0.1%	0 0.0% 0.0%
Peri-operative myocardial infarction	28 3.3% 3.3%	7 4.3% 0.8%	5 4.8% 0.6%	0 0.0% 0.0%	0 0.0% 0.0%
Stroke File III	11 1.3% 1.3%	1 0.6% 0.1%	1 1.0% 0.1%	0 0.0% 0.0%	0 0.0% 0.0%
Operative mortality	12 1.4% 1.4%	3 1.8% 0.4%	3 2.9% 0.4%	1 2.3% 0.1%	0 0.0% 0.0%
Myocardial infarction File IV	0 0.0% 0.0%	0 0.0% 0.0%	0 0.0% 0.0%	0 0.0% 0.0%	0 0.0% 0.0%

The data as extracted by class of patient. 837 of the 1407 surgeries performed during the two year period were coronary bypass grafts, without additional complicating factors. 164 of these operations were on females.

Equations for Page B

(Bb7): AD9
 (Bb8): 1
 (Bb10): @Using cardio.post, num when id-key=!c and \kpl-d(3)\=Y
 (Bb11): B10/B7
 (Bb12): B10/{B7}
 (Bb14): @Using cardio.post, num when id-key=!c and \kpl-d(4)\=Y
 (Bb15): B14/B7
 (Bb16): B14/{B7}
 (Bb18): @Using cardio.post, num when id-key=!c and \kpl-d(12)\=Y
 (Bb19): B18/B7
 (Bb20): B18/{B7}
 (Bb22): @Using cardio.post, num when id-key=!c and alive-or-dead=D
 (Bb23): B22/B7
 (Bb24): B22/{B7}
 (Bb26): @Using cardio.postpost, num when id-key=!c and \kpl-d(4)\=Y
 (Bb27): B26/B7
 (Bb28): B26/{B7}
 (Bc7): AD10
 (Bc8): C7/B7
 (Bc10): @Using cardio.post, num when id-key=!d and \kpl-d(3)\=Y
 (Bc11): C10/C7
 (Bc12): C10/{B7}
 (Bc14): @Using cardio.post, num when id-key=!d and \kpl-d(4)\=Y
 (Bc15): C14/C7
 (Bc16): C14/{B7}
 (Bc18): @Using cardio.post, num when id-key=!d and \kpl-d(12)\=Y
 (Bc19): C18/C7
 (Bc20): C18/{B7}
 (Bc22): @Using cardio.post, num when id-key=!d and alive-or-dead=D
 (Bc23): C22/C7
 (Bc24): C22/{B7}
 (Bc26): @Using cardio.postpost, num when id-key=!d and \kpl-d(4)\=Y
 (Bc27): C26/C7
 (Bc28): C26/{B7}

The equations for two of the columns on the opposite page. The first column (spreadsheet column B) analyzes the risk of complications for the entire set of patients, Set C. The second column (spreadsheet column C) analyzes exactly the same risks for female patients, Set D. Because the columns are basically identical, column C is a /REPlicated duplicate of column B. Column C was modified using the search-and-replace command (/S&R), replacing "!c" with "!d" everywhere in the column. Each of the remaining columns was built in same manner. The time necessary, therefore, to construct the entire page was only 10-15 minutes.

High-Speed Generic Searches in IMAGE

An unanticipated benefit of @STORE sets in QueryCalc is that they provide a mechanism for high-speed generic searches of IMAGE datasets under specific conditions. Other database structures, such as ISAM, KSAM, and SQL[†], often have advantages over IMAGE when a query question's relational operator (*relop*) is "greater than" or "is between". These relops do not imply a search for a single value, but rather a range of values. High-speed range searches, unfortunately, are normally impossible in IMAGE.

The mechanism by which a generic search is implemented in ISAM, KSAM and SQL is called a *b-tree*, short for *binary-tree*. IMAGE, in contrast, uses *hashed* keys. In a hashing algorithm, the search item value, even though it may be text, is considered to be a number. All data stored in a database is stored as 1's and 0's. Whether those bits are to represent text or numeric information depends only on the person defining the database, not the storage mechanism itself. It's therefore quite possible to take any value and view it as a number. In a hashing key scheme, this number will represent an address in a look-up table. The value held at the table address will be the record number of the first record containing the search item value. Being only a two-step process, hashing is the most efficient search algorithm known, but it suffers from a severe drawback in that the search item value must be known in its entirety before it can be used.

B-trees are different. A b-tree operates by asking a series of yes-no questions (hence the term, *binary*). The questions are of the form: is the search value is greater or less than a particular value? By navigating a b-tree, records may either be found (1) which possess exactly the search value specified, or (2) lie within a specified range of values. This latter form of search is called a *generic* search. The capacity for generic searches underlies the appeal of b-tree search indexes. However, this enhanced search capability is not without its cost. B-tree indexes are usually quite a bit slower than hashed searches.

Generic searches may be simulated in IMAGE databases using @STORE sets. But what you may find to be surprising is that generic searches in IMAGE using QueryCalc's @STORE sets are often significantly faster than KSAM b-tree searches. The trick is quite simple. Any search item in an IMAGE detail dataset must have a master dataset attached to it. A serial-

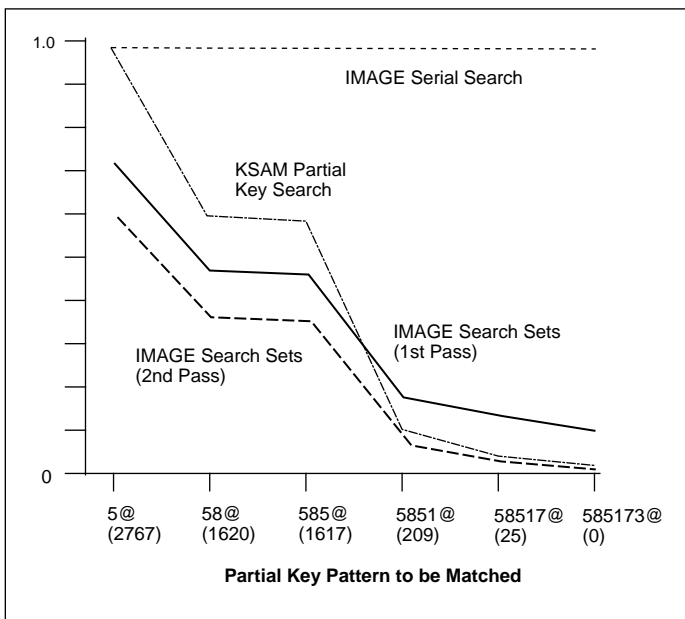
[†]ISAM stands for *indexed sequential access method*. ISAM is an IBM product. KSAM is an HP product and is very similar to ISAM. KSAM stands for *keyed sequential access method*. SQL was originally an IBM sublanguage, part of the DB2 database, but the use of the term has now become generic. SQL stands for *structured query language*.

search of the master dataset is almost always much less expensive in time and CPU resources than a serial search of the detail dataset. All of the search values which qualify in a partial-key serial search of the master dataset are @STORED in a QueryCalc search set. That set is then applied in a search of the corresponding detail dataset. The query questions will be of this nature:

```
@using employee-id, store in !m socsecnum
when socsecnum=58@
```

```
@using payrecord, sum of gross
when socsecnum is !m
```

In this example, EMPLOYEE-ID is an IMAGE master dataset. PAYRECORD is a detail dataset. The acceleration advantage that search sets offer accrues because there are often a large number of entries attached to each search chain in an IMAGE detail dataset. Although serially searching the master dataset and building the search Set M requires some resources, the technique is actually faster than either serially searching the detail dataset or using a b-tree search against an identical KSAM dataset. Indeed, as shown by the figure below, the speed advantage can often be quite significant.



The ratio of measured search times for various generic search techniques. The partial key pattern to be matched is shown along the x-axis. The "@" represents a wild card symbol. The number of records that qualified in each search is shown in parentheses.

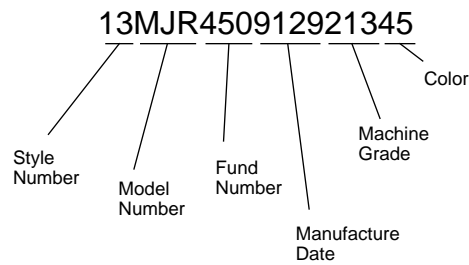
A serial search of a detail dataset will generally be the slowest method. Its time-of-search will however be independent of the number of records that are qualified. When the number of records qualified becomes large enough (10-30%) of the dataset, serial searches often become faster than b-tree searches.

Two IMAGE search set curves are shown. The "1st Pass" curve includes both query questions shown in the text above (a serial search of the master dataset followed by a keyed search of the detail dataset). But once a search set has been built, it may be reused again and again without further overhead. The reuse times are shown in "2nd Pass".

High-Speed Partial Key Searches of Concatenated Keys

The capacity to search for only a portion of a key item's value is called a *partial key search*. The b-trees keys of KSAM and SQL allow you to do this quite easily, but only if you are searching for the first few characters of the key. Quite often however the need arises to seek out records on the basis of values lying somewhere in the middle of the key value. In this case, unfortunately, a b-tree will be of no use. However, IMAGE inquiries using QueryCalc's @STORE sets will continue to work as efficiently and offer distinct advantages over b-tree generic searches.

Intermediate partial key searches become an important topic if your database uses concatenated keys. A *concatenated key* is one where a number of separate characteristics are put together in a single key. An example of a concatenated key is shown here:



Concatenated keys can often be quite useful. They allow you to locate specific records quite quickly. But just often, they represent a significant barrier to efficient report writing. If you must search for an item in the middle of the key, such as *fund number* in this example, generally the only recourse in KSAM, IMAGE, and SQL is to use a serial search.

The search-accelerating method discussed on the previous two pages works just as well for intermediate partial key searches, but only when IMAGE databases are used. The technique will be exactly the same: (1) first perform a serial search of the master dataset, storing the qualified key values, and then (2) apply the formed set to the detail dataset, as shown:

```
@using part-master, store in !a part-key
  when part-key(6,8) = 450
```

```
@using part-detail, sum of qty-on-hand
  when part-key is !a and stock-region is AZ
```

This technique will not work with MPE flat files, KSAM, or SQL datasets

because they do not have directly accessible master datasets. The presence of such accessible key files (master datasets) in IMAGE is the reason the method works.

If the key item is text, there are a number of ways to search for pieces of the key item value. The search may proceed either by absolute position or through the use of "wild cards". Please see Chap. 6, "Query Questions" for a complete list of partial search capabilities.

When to Use the Method

The generic-search technique of using search sets is often quite advantageous, but like most things, it is no universal panacea. There are specific instances when it should and shouldn't be used. The rules on when to use the technique are these:

1. *Use the method if you are likely to qualify less than 20-30% of the records in the dataset.* Qualifying more records than this percentage and a single serial search of the dataset becomes faster than multiple chained searches. In these circumstances, write the query question in the normal fashion, as shown here:

```
@using part-detail, sum of qty-on-hand
when part-key(6,8) is 450 and stock-region is AZ
```

2. *Don't use the method if there is a more direct way to retrieve the records.* If you know some special information about your records, such as that the Arizona stocking region represents only a very small proportion of records in your database, and STOCK-REGION is a search item, then again write the query question as shown above. Searching down just one short search chain will be much faster than searching through multiple chains.
3. *Use the method when there are many records in the detail dataset for each master dataset key-item value.* The technique works best when a one-to-many ratio exists. The greater the number of records per chain in the detail set, the greater the speed advantage of the technique. But if only one record exists per chain, no advantage exists. Indeed, then there will be a small "cost" to the technique.

Boolean Set Algebraic Operations

QueryCalc allows the extensive manipulation of search sets using standard Boolean algebra set operations. The only requirement is that the sets must be of matching data types (e.g., I2, R4, Z8, etc.). Normally, you do not need to be aware of the data type of the search values you are manipulating. It becomes a matter of concern here simply because there is no easy way to combine text and numbers into a single set.

The syntax for the set manipulation algebra is:

```
@Using sets, !y=!a+!b
```

Set Y will be defined as the *union* of Sets A and B. Four set algebraic operators are possible and are represented in the facing figure, along with the resulting set. They are: *union*, *subtraction*, *intersection*, and *exclusion*. In all set algebra operations, duplicate entries are eliminated. No value will appear more than once.

If set operations are to be performed on multiple sets, the operations must be placed in a series of cells:

```
@using sets, !y=!a+!b
@using sets, !y=!y+!c
@using sets, !y=!y-!d
```

This sequence of set operations is equivalent to

$$!y = (!a + !b + !c) - !d$$

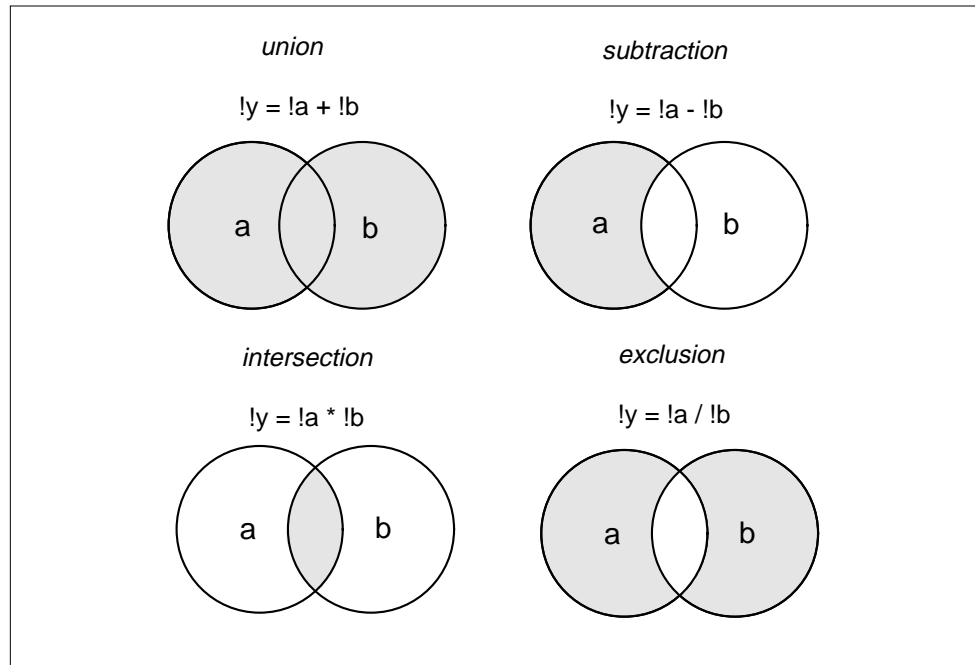
Using a series of cells, any Boolean algebraic manipulation is possible.

Order Dependence

All of the algebraic operations [other than set subtraction (-)] are *commutative*, that is, the order is unimportant. For the commutative operations, each of the equations shown below are equivalent:

```
@using sets, !y=!a+!b      @using sets, !y=!b+!a
@using sets, !y=!a/!b      @using sets, !y=!b/!a
@using sets, !y=!a*!b      @using sets, !y=!b*!a
```

Set subtraction is the exception. Order is very important, as you can visualize by looking at the set diagram.



The Boolean algebraic set operations of QueryCalc.

Manually Defining a Search Set
 Sets may be manually defined rather than be assembled from data in the database. Examples of the syntax are:

```
@using sets, define !a as
  holland, "smith, john",
  yankees, valley-plum; type=U20
```

```
@using sets, define !b as
  576, 711, 2303; type=I2
```

The rules for set assembly are simple. If the items to be defined to comprise the list are text, they must be surrounded by quotes if the item contains a comma or a semicolon; otherwise, they may simply be listed, as shown. Text will be upshifted if the dataitem type is specified to be "U" and left as is if the type is "X". If the dataitem type is specified to be a numeric data type, all of the items in the list must be numeric values. Duplicate entries will be eliminated and the list will be sorted, as is normal for all sets.

Using External Files to Define Search Sets

Search sets may also be defined using external flat ASCII files. The syntax is:

```
@using sets, define !c from
filename.groupname.acctname;type=R4
```

Please note that the difference between these this set-definitional form and the previous one is the use of the "as" and "from" prepositions.

The italicized names specify the group and account of the file and are optional. The file must be a flat ASCII file containing only printable characters. If you can text in and read the file using EDIT/3000, then it meets these specifications. You may create this file using EDIT/3000, *but the file must be kept unnumbered*. The flat ASCII file containing the search item values must look like this:

```
abercrombie
wallace
mustafa
chin
hruska
miller
lucky
syvertson
.
.
```

Only one search item value should appear in each row. The item should be left justified. If leading blanks must appear as part of the key item value, type them in that way. They will be retained. All trailing blanks will be de-blanked, however. The defined list will be sorted on entry and duplicate entries will be eliminated.

Concepts
Introduced in
Chapter 10

SEARCH SET	A list of search item values that belongs to no dataset or database.
NESTED SUBSET	A set of items which is derived from a larger set. All of the items in the subset appear in the larger set.
GENERIC SEARCH	A search which proceeds by looking for all of the records which lie between specified limits. The exact values of the search items are not known in advance, as they are in a specific search.
PARTIAL-KEY SEARCH	A search which proceeds by looking for a part of search item value.
SET ALGEBRA	The algebra of logical sets. The common operations are set union, intersection, subtraction. Union is equivalent to logical ORs. Intersection is equivalent to logical ANDs.

Technical Appendix: Comparing SQL to QueryCalc

Introduction to SQL

SQL (*structured query language*) is a database/query language that was defined in the early 1970's by IBM. SQL was built around the idea of *sets*, and thus it is appropriate to compare SQL to QueryCalc at the end of this chapter. QueryCalc, in many ways, is surprisingly similar to SQL, especially in its query questions. But there are also significant differences. These differences are not only the result of different design purposes but also very different views on how the data should be presented and how much power should be given to the user.

SQL, in contrast to QueryCalc, contains its own database structure. QueryCalc accesses pre-existing databases, generally IMAGE and KSAM. SQL, on the other hand, is not a report writer *per se*. Although SQL has a well-designed interface which may be used to interactively test query questions or be used as an *ad hoc*, "quick-and-dirty" report generator, any complicated output that would normally be associated with a production-level report must be written in another language, such as PASCAL.

One of the design criteria of SQL was to hide as much of the database structure as possible from the user. Ideally, the user would not need to learn or know anything of keys, datatypes, or the like. In actual practice, as you might expect, that becomes impossible. The skilled SQL user must eventually become at least as knowledgeable about his database structures as a user of IMAGE.

SQL datasets are constructed as tables, as shown at the right. The dataitem names are listed at the top of the columns. The presentation order is rotated 90° from the way data is presented in QueryCalc. Each record entry is a row in the table. Two attributes differentiate SQL datasets from IMAGE: (1) No row (record entry) may be repeated in a table (this is a major difference) and (2), key item values in SQL may be the result of the concatenation of several adjacent dataitems, as shown in the SPJ table, a suppliers-parts-jobs table. The other tables in the example are the suppliers, parts, and jobs tables. Key items are marked where a value is encircled by a light box.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Punch	Rome
J3	Reader	Athens
J4	Console	Athens
J5	Collator	London
J6	Terminal	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J7	400
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	300
S4	P6	J3	500
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P6	J3	100
S5	P6	J4	900
S5	P6	J7	100

A simple SQL-like database for suppliers, parts, and projects.

Query Questions in SQL & QueryCalc

Query questions in the two languages are quite similar. For example, to find the records which meet a specified set of selection criteria, the syntaxes are:

```
@using parts, find when status=30 (QC)
```

```
Select * from parts where status=30; (SQL)
```

The "*" in SQL means retrieve and display all columns (dataitems) in the table. SQL organizes the records it finds into a new table, called a *result table*. QueryCalc marks its qualified records in basically the same way. A @FIND is followed by a @SHOW command. In SQL, you may specify that the retrieved record table only contain certain dataitems:

```
Select pname,color from parts
where status=30; (SQL)
```

In QueryCalc, the command following an @FIND would be:

```
@show pname,color (QC)
```

In both query languages, statistical calculations may be performed on the data in a dataset as a search proceeds. Examples of the two syntaxes are:

```
@using parts, sum of weight*454
when status=30 (QC)
```

```
Select sum(weight*454) from parts
where status=30; (SQL)
```

SQL is a Relational Database

SQL was built around the "Relational Model" of Dr. Edgar Codd. The word "relational" in a relational database has nothing to do with the common use of the English word, where you might suspect that items in different datasets can somehow be easily related to one another. Rather, the word is derived from the mathematical definition of a *relation* in set theory. A relation R is a n -tuple defined as

$$R = \langle d_1, d_2, d_3, \dots, d_n \rangle$$

where $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$, where D_1, D_2, \dots, D_n are the *domains* for each of the $d \in D$. A domain specifies the allowed set of values

which each element in the n -tuple may assume. A relation R is simply one table in an SQL database. The various d_i are the column entries (color, pname, city, etc.). Entry values in an SQL table are "drawn" either from the set of allowed values (the domains) or classified as either one of two *missing* categories, important or not important. In contrast, IMAGE and KSAM do not intrinsically impose limits on what data may be entered into a dataitem's field. Nor do KSAM or IMAGE allow marks for missing data.

Structured Query Questions

The word *structured* in Structured Query Language comes from the capacity in SQL to nest a sequence of query questions. Each query subpart establishes a qualifying set of records which may then be used to qualify another set of records in another table (dataset). In SQL, if you wished to list all of the suppliers who manufacture red parts, three tables (suppliers, parts, supp-parts) might be used. If so, the syntax would be:

```
Select suppliername
from suppliers
where suppliernum in
    ( Select suppliernum
      from supp-part
      where partnum in
        ( Select partnum
          from parts
          where color='Red' ) );
```

The syntax in QueryCalc is different, but the result is precisely the same:

```
@Using parts, store in !a partnum
    when color=Red
@Using supp-parts, store in !b suppliernum
    when partnum=!a
@Using suppliers, find
    when suppliernum=!b
@show suppliername
```

In SQL, the nested query sequence is read from bottom-to-top. In QueryCalc, the order is not only reversed, but the individual query parts must be placed in separate cells. Which is better? Beyond personal preferences, the differences between the two approaches speak to the core of the differences underlying the design philosophy of the two languages.

The Concept of Joins

SQL allows you to combine columns from various tables into a new result table. This process is called a *join*. In most commercial SQL implementations, only a few types of joins are possible. In HP's ALLBASE/SQL, only *natural joins* and *outer joins* are supported. An example of a natural join is the SQL nested query on the previous page. A natural join only returns records for which there are exact matches in the joined tables. Outer joins allow the union of records where matches don't exist. Codd (1990, *The Relational Model for Database Management: Version 2*) has defined 43 different forms of joins, giving them names such as *recursive join*, *semi-theta-join*, and *symmetric equi-join*, while suggesting that the list is by no means complete. QueryCalc is completely differently designed at this level of organization than is SQL. QueryCalc uses no joins at all. Rather, it was designed (1) to read from only one dataset at a time, and then (2) relate that data to any other dataset in any way that the user might wish.

These philosophical differences do make substantial practical differences. Dr. Paula Hawthorne, director of applications and technologies at Hewlett-Packard, recently said, "I was at one point in charge of the quality assurance group... SQL as a language is very confusing to the users. At least half of our bug calls were because people didn't understand an SQL query and what the right answer to that query was. In fact, often reading the literature, we didn't understand it either. Part of the problem with quality in relational database system is a problem with SQL" (*Interexpress*, June 1991).

SQL was designed with the best possible intentions: to make the data presentation as simple as possible. No presentation can be simpler than a row-column table. But we have always felt that motives directed towards the isolation of the user from the data storage were a fundamental mistake. Such an approach not only confuses even the most astute user, because he does not know precisely what the program is doing for him, it places an immense burden on the query language designer. The designer must generate large-scale data operations capable of every possible manipulation the user might desire. The far preferable tack would seem to be to generate a few, very powerful primitives that can connected together indefinitely.