

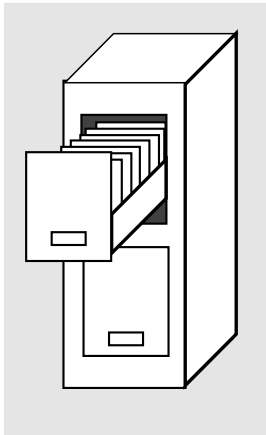
# 6

Creating a summary of dataitem(s) from a single dataset

## Query Questions

### The Single Dataset Query Question

The query question which summarizes data from a single dataset is the heart of QueryCalc. Although a variety of methods exist in QueryCalc to simultaneously extract data from multiple datasets, databases and multiple HP3000's (as explained in Chapters 10, 11, and 13), each of these methods is built around the question that gets data from a single dataset.



The single-dataset query question will always look something like this:

```
@Using invoices, sum of amount when  
category is 501 and date > 930101
```

The word "when" is the keyword in a query question. The phrase to the left of "when" specifies what statistics are to be summarized. The phrase(s) following "when" specify under what conditions retrieved records are to be added into the summarization. These restrictions are called the *qualifying phrase(s)*. The records in the dataset which pass these restrictions are commonly called the *qualifying entries*.

The statistics which may be summarized in a query question are these:

```
Sum of ....  
Avg of ....  
Max of ....  
Min of ....  
Var of ....  
Dev of ...  
Val of ....
```

The last statistic in the list should be read as: "get me the first value of...". This statistic is different than the others. The search of a dataset stops immediately once a single record has been qualified. The value returned is thus the "First Value". This form of query question will be predominantly used in dataset "rereadings" (an idea which will be explained shortly).

## The Query Question Explained

A complete query question will always look basically like this:

```
@Using qcdemo.invoices, avg of amount
when jobnum ib 8000,8100 and date>=19950601
```

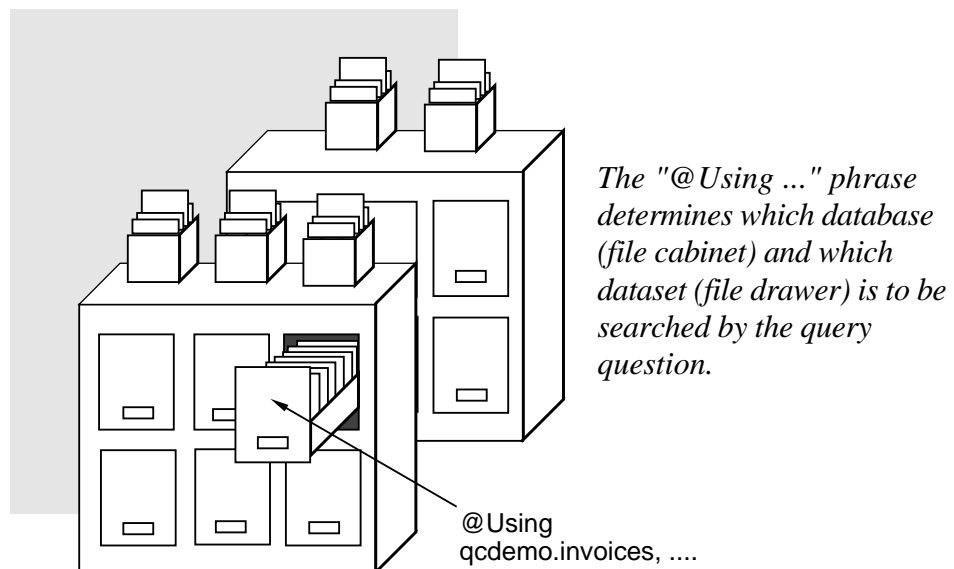
In order to explain what's possible in a query question, each phrase in the query question will be described in detail in the next few pages.

### The "Using..." Phrase

The initial "Using..." phrase at the beginning of a query question specifies which dataset in which database is to be searched. The phrase may be typed in any of four different fashions. Three of them are:

```
@Using database.dataset, ....
@Using database, ....
@Using dataset, ....
```

*Or the phrase may be left off altogether.* If the phrase is left off, QueryCalc will look only in the last-used database (called the *default* database). A query question is not complete until it is fully specified as to which database and dataset is to be searched. QueryCalc will automatically attempt to fill in whatever information you omit. If you specify only the database, as in the second example, QueryCalc will change *default* databases and look only there. If no single dataset contains all of these items, QueryCalc will announce that as an error. Should more than one dataset possess all of the items, QueryCalc will present you with a list of choices.



## The Item(s) to be Summarized may be Specified as an Equation

The item to be summarized in the query question can either be a single dataitem, as in this example:

```
@Using db.ds, sum of amount
```

or it can be a calculated combination of various dataitems:

```
@Using db.ds, sum of
amount*workcomplete/jobcost
```

Because of this feature in QueryCalc, the object of the summarization is called a *dataitem equation*. Both the item *amount* by itself and the phrase *amount\*workcomplete/jobcost* are dataitem equations. The rules for use of dataitem equations are these:

1. All of the dataitems must come from the same dataset.
2. Only the five basic mathematical operators ( + , - , \* , / , ^ ) are allowed. No functions are permitted. Function manipulation of the retrieved items can be handled using the user-defined query functions (see Chap. 11).
3. Parentheses are not allowed. (Parentheses are used for other purposes in specifying dataitems.) If you need to enter an equation such as

$$A * (B + C)$$

where A, B, and C are dataitem names, multiply the equation through and enter it as

$$A * B + A * C$$

A common practice used by many database programmers is to create dataitem names which use some or all of the mathematical characters, but which were not meant to be interpreted mathematically. Hyphens are especially commonly used.

## Resolving Ambiguities in Dataitem Names

How can you tell what's what in a dataitem equation such as this:

```
@avg of current-age-age-operated-on
```

Surprisingly, QueryCalc can often work these equations out by itself. QueryCalc does this by examining each part of the dataitem equation piece by piece. If the specified database contains the item *current*, the next hyphen must be a minus sign. If *current* is not a dataitem, *current-age* is tried. If this dataitem name exists, the following hyphen is interpreted as a minus sign. This step-wise process of parsing dataitem names will fail however if there are two dataitems in the database, one named *current* and the other *current-age*. The second dataitem will never be seen and equation processor will unsuccessfully attempt to subtract *age-age-operated-on* from *current*.

You can remove any ambiguities in your specified dataitem equations by placing backslashes (" \ ") around the dataitem names:

```
@avg of \current-age\-\age-operated-on\
```

Dataitem names specified in this manner are unambiguous. An added advantage of the backslashes is that the processing of the query question is slightly faster.

## The Qualifying Phrases

The qualifying phrase(s) following the word "when" in the query question determines which records are to be added into the accruing statistics during the search of the specified dataset. The word "when" is not always required. A query such as the following:

```
@Using invoices, sum of amount
```

will simply qualify every record in the dataset (but because no search items were specified to be matched, a serial search will be required). A more standard query question would look like this:

```
@Using invoices, sum of amount
when category is 501
```

This query will extract only a subset of the records in the dataset. If the dataitem *category* is a search item, the retrieval time will generally be quite quick.

A qualifying dataitem may also be a dataitem equation, as in this example:

```
@Using invoices, sum of amount when
amount-balance*1.25 > 10000
```

## The Three Parts of a Qualifying Phrase

Every query qualifying phrase consists of three parts: (1) a *dataitem* (or *dataitem equation*), (2) a *relational operator* ("relop" for short), and (3) a *pattern* to be matched in the dataset's records.

<i>dataitem</i>	<i>relop</i>	<i>"pattern"</i>
amount	>	10000

The qualifying value ("10000") is called a "pattern" to emphasize the nature of a match in a database. You don't so much match numbers or letters when you qualify records as you match *bit patterns* (*strings of 1's and 0's*). Being a simple machine, that's all the computer can look for. Text forms a certain set of bit patterns, numbers another set. QueryCalc automatically generates the proper bit patterns based on the dataitem type being matched.

## Chained or Serial Search?

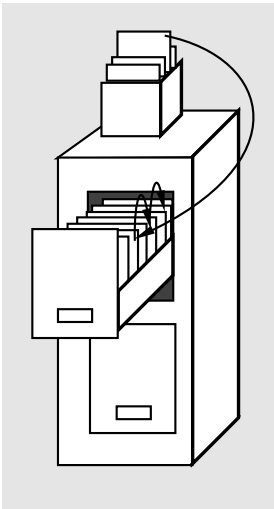
The qualifying phrases determine whether or not the search is to be *chained* or *serial*. For a chained search to be possible in IMAGE, at least one dataitem in the qualifying phrase must be (1) a search item, (2) matched with "=" or *is* relop, and (3) specified as a whole (no partial substrings).

```
@sum of amount when
category is 501 and date > 19930101
```

The dataitem *category* would meet those criteria in this example. *Date* would not qualify for a chained search in IMAGE, even if it were a search item.

The rules are slightly different for KSAM databases. KSAM is less restrictive about a precise pattern match than IMAGE is. For a keyed search to be possible in KSAM, at least one dataitem in the qualifying phrase must be (1) a search item, (2) the relop may be any relational operator other than *not equals* (" $\neq$ "), and (3), if a partial string is to be used, it must begin with the first character.

Otherwise, there is no difference between query questions for IMAGE and KSAM databases.



## The Seven Possible Relational Operators

There are only seven relational operators ("relops") possible in a query question: (1) equals, (2) greater than, (3) less than, (4) greater than or equals, (5) less than or equals, (6) not equals, and (7) "is between". The relational operators may be written in a query question in any of the following ways:

---

```

=   is  ie   eq
>   gt  igt
<   lt  ilt
>=  => ge   ige  inlt
<=  =< le   ile  ingt
<>  #   ne   ine  isnot
ib

```

---

These are the most common standard representations for the seven possible relational operators. All of the relop forms on a row are equivalent. However, there is a restriction with the use of some of the relops. Those relops which start with a letter, or the "#" symbol, must be separated by a space from the dataitem name and the pattern to be matched. The reason is alpha characters and the "#" symbol are legitimate characters in IMAGE dataitem names. Without spaces, reliable sentence parsing becomes impossible, as in this case:

```
@sum of amount when ssn##452567191
```

QueryCalc will announce its inability to properly parse such a condition. The query question must be rewritten in this fashion:

```
@sum of amount when ssn# # 452567191
```

The qualifying phrase may be run together, without spaces, if the symbolic relops (=, >, <, etc.) are used. These symbols are illegal characters in IMAGE dataitem names, thus there is no ambiguity as to what part of the text is the dataitem name and what is not. An example:

```
@sum of amount when ssn#<>45267191
```

## Specifying the Pattern to be Matched

A qualifying phrase may be written as simply as this:

```
... when state is CA
```

where the pattern to be matched is CA. Quotes are not required around the pattern if the value to be matched is composed of only alphanumeric characters, absent of any form of punctuation or spacing. Quotes *are* required to surround the pattern to be matched if non-alphanumeric characters appear as part of the pattern, as in this example:

```
... when employee is "Smith, Joe"
```

The dataitems in these two examples, *state* and *employee*, are text dataitems. If a dataitem is defined as an "X"-type text field in the database, both upper and lower case text may appear in the dataitem's values (see Chap. 2). If the text dataitem is defined to be a "U"-type field, then only upper-case values are expected to appear in the dataitem's values. QueryCalc will automatically upshift all pattern text to match a "U"-type dataitem. *If however the dataitem is an "X"-type field, QueryCalc will leave the pattern to be matched as you typed it. Therefore you must be careful to specify a text pattern correctly, letter for letter, in its correct lettercase, if a match is to be found in the database.*

```
... when idnumber is 134527
... when pressure is 1e5,7.3e6
```

Numbers require no such consideration. Regardless of how the number is stored in the database or how you represent the number in your query question, QueryCalc will automatically build the proper bit pattern to match the datatype used in the database.

## Matching a Pattern Off of the Spreadsheet

Much of QueryCalc's power is derived from its ability to extract pattern values off of the spreadsheet. Brackets ([...]) are used in a query question to form a "window" back into the spreadsheet's equation interpreters so that you can synthesize a pattern to be matched.

Examples of common numeric equation patterns are:

```
.... when productno is [ab12]
.... when idnumber is [1900+g13+g17/10]
... when startmonth is [190000+{az5}]
```

When a query question containing equations is replicated, the cell addresses used in the equations will be automatically adjusted by the amount of displacement traveled on the spreadsheet. The exception are cells surrounded by braces, which indicate absolute cell references (the third example on the previous page).

Text equations may be similarly created:

```

.... when name is [$g34]
.... when date ib [{"19"+{az5}], [{"19"+{az6}]}
... when product=[$if$(ab7>ab8, "MRJ1501", g34)]

```

Any numeric or text equation that can be placed in a cell on the spreadsheet may appear with the brackets of a query question. In the third example, the pattern to be matched is made conditional on the relationship of two cells, Ab7 and Ab8.

## Subitems & Substrings<sup>†</sup>

Patterns may similarly be matched to subitems in a dataitem array. The subitem is referenced by a *single* index:

```

... month-profit(4) > 12000
... store-number(16) = [$r45]

```

The dataitem may be either text or numeric. It is of course necessary that the pattern to be matched be the same data type as the dataitem.

Substrings within a text dataitem may also be matched. Substrings are indicated by *two* indexes:

```

... date(3,5) = 051

```

The first index indicates the *start* position of the first character in the dataitem's text string; the second indicates the *stop* position. The example shown

---

<sup>†</sup>A *string* is simply a string of *text* characters. This sentence is called a string in the common parlance of programming. A *substring* is a specified subset of the whole string, with specific start and stop character positions. *Subitems* are different. A subitem is part of a *dataitem array*. Simple (non-arrayed) dataitems are written as I1, R2, Z10, X20, etc. Arrays are written as 30X6, 20R2, 10Z6, etc. The simplest way to imagine an array is think of post office boxes. All of the boxes are given the same (dataitem) name. It's the *index* number that is attached to the dataitem name that allows you to select the proper box. In QueryCalc, if you leave the subitem index off, you select data from the first subitem.

would declare a match whenever *date* equaled any of the following values: 840513, 20051012, MY05167.

Matches searching for substrings within a subitem are indicated by *three* indices:

```
... acctcode(4,5,8) is GR7E
```

The first index indicates the *subitem*. The second and third indexes indicate the *start* and *stop* positions within the character string. This form of subitem matching can be used only with text dataitems.

All of the indexing values for a substring or subitem may be taken off of the spreadsheet, and thus be made variable rather than remain fixed values. The previous example could be rewritten as:

```
... when acctcode(c5,c6,c7) is GR7E
```

where cells C5, C6, and C7 contain the desired indexes. *The index values may only be cell references, not equations.* If the index value is to be calculated, it must be calculated in the referenced cell. An example which would violate the proper use of indices is:

```
... acctcode(c1*4+2,1,4) (illegal)
```

Otherwise, fixed (constant) values, relative cell addresses, and absolute cell addresses may be freely intermixed:

```
... acctcode(c1,1,{d5})
```

## Text Matches Using Wildcards "@"

For text dataitems, wild cards may also be used:

```
... name is @ote  
... name is ote@  
... name is @ote@
```

The first example would find all name values which end in "ote". The second would find those that began with "ote". The third would find those that contained the string "ote" anywhere in their entry. The match is *case sensitive* for X-type text dataitems. It is *case insensitive* for U-type text dataitems.

Table 6.1. Match Patterns Reviewed

1. <i>state is CA</i>	The pattern to be matched is CA. No quotes are required if there are no spaces or punctuation marks in the pattern value.
2. <i>employee is "Smith, Joe"</i>	Quotes are required to surround the pattern if non-alphanumeric symbols (anything other 0-9,a-z) are used. If the IMAGE dataitem to be matched is a "U-type" text dataitem, the pattern will be upshifted to match the database entries.
3. <i>productno is [ab12] name is [\$g34]</i>	Brackets indicate that the pattern to be matched will come off of the spreadsheet. In the first example, productno is matched with the numeric value taken from cell Ab12). A text dataitem must be matched with a text equation. In the second example, name is matched with the text value from cell G34.
4. <i>acctcode(5) is P</i>	A dataitem which contains subitems is referenced with <i>one</i> index. If no subitem is specified for a subitemed array, the first subitem is assumed.
5. <i>productnum(5,8) is [\$CG7]</i>	A substring of a text dataitem (either X or U type) can be matched using <i>two</i> indexes. The indexes indicate that the substring to be matched begins with the fifth character and ends with the eighth.
6. <i>@val of acctcode(9,3,5) when productnum(5,8) is 3415</i>	A substring of a subitem may be matched with <i>three</i> indexes. The substring of characters residing in character positions 3 to 5 of the 9th subitem of <i>acctcode</i> will be the value returned when the indicated substring of <i>productnum</i> equals 3415.
7. <i>@val of month-profit(d5) when productnum(1,{d6}) is [\$g34]</i>	Cell references (but not equations) may be used in place of fixed values for any of the indexing values. The cell references may be either <i>relative</i> or <i>absolute</i> .
8. <i>name is Hol@ name is @ght name is @erin@</i>	"Wild cards" (@) may be used in text string matches. In the first example, any text value beginning with "Hol" will qualify. In the second, any text ending with "ght" will qualify. In the third example, the string of characters "erin" anywhere in the text entry will qualify.

## Summing Subitems with the ":" Operator

For numeric items in a subitemed array, QueryCalc offers an easy way to sum a range of subitems using the ":" operator:

```
@sum of month-profit(1:6) when ....
```

Inserting a ":" between two subitem indexes is equivalent to summing all of the *month-profit* subitem values from the first month to the sixth, as shown:

```
@sum of month-profit(1)+month-profit(2)
      +month-profit(3)+month-profit(4)
+month-profit(5)+month-profit(6) when ....
```

Either or both of the *start* and *stop* subitems in the summation may be made cell references:

```
@sum of month-profit(1:d5) when ....
```

Using cell references for the indexes provides an easy mechanism for *rolling calculations*. For example, if the cell D5 contained an equation which automatically calculated the month number relative to the fiscal year, the year-to-date sum of an array could be calculated automatically.

## Using And's & Or's in the Qualifying Phrase

Up to 20 dataitem qualifying phrases may be appended together in one query question using "and's" and "or's".

```
@find when jobnum=8404 and date>850000
      or jobnum=8405 and date>850601
or jobnum ib 8406,8499 and contractamount>100000
      and date>850000
      or jobnum=8501 or jobnum=8502
```

The "and" takes logical precedence over the "or". That is, all of the "and'ed" items before the first "or" are evaluated as a unit. If the first "and" phrase is found to be true, the record is accepted. If any one of the "and" matches is not true in the first phrase, the second series of "and's" beyond the "or" is tried. The process is repeated until one complete "and" phrase is found to be true or the end of the sentence is reached. If none of the "and'ed" phrases are found to be true, the record is rejected.

## Implicit OR Lists

Because of their use elsewhere in query questions, *parentheses* cannot be used to group "and" and "or" phrases. The alternative is to use QueryCalc's *implicit or* lists. Virtually any query question can be formulated through the use of implicit or's.

```
@sum of hours*1.31+overtime
when facility is 701,711,734,707,749
and unit-number is 3409,3511,3613,4303
and job-number is 673,812,556
```

Up to 10 items may be specified in a list. The match patterns may be fully specified, incompletely specified (through the use of wild cards), taken off of the spreadsheet, or synthesized with an equation:

```
@find when last-name is
SMITH, ROBERT@[ $a3 ], [ $a4+"GER" ], @TON
```

## Dependent Query Questions

All of the query question forms discussed to this point have been *independent* query questions. That is, regardless of where the query questions are placed on the spreadsheet, the answers they extract will be independent of all prior queries.

QueryCalc also contains two related query question forms which are *dependent* on the last-asked standard query question. The forms are:

```
@ucs (using current statistics)
@rereading, sum of ....
```

## The @UCS Form

When a standard query question calculates a requested statistic (sum, avg, max, etc.), all of the statistics are actually calculated. Thus it is a no-cost query to ask for the other statistical values. The calculated statistics may be requested by using any of these @UCS (*using current statistics*) forms:

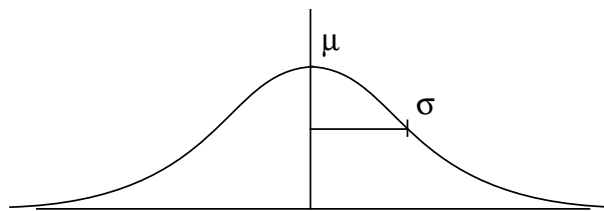
```
@ucs, sum           @ucs, avg
@ucs, dev           @ucs, var
@ucs, max           @ucs, min
@ucs, num           @ucs, pct
```

The statistic "Pct" returns the percentage of those records searched which were qualified. If the search were *serial*, the percentage would be of those

records which qualified out of the entire dataset. If the search were *chained*, the statistic would be the percentage that qualified on the chain.

Frequently, information is required to describe not merely a single statistic such as the sum or average, but instead the how the *population* of entries is distributed. Three statistics describe a Gaussian (Normal) distribution:

$n$ , the number of entries qualified,  
 $\mu$ , the mean (or average),  
 and  
 $\sigma$ , the standard deviation.



A standard "bell"-shaped Gaussian (Normal) distribution.

These three statistics may be retrieved in QueryCalc by placing the following equations in cells immediately after a standard query question:

```
@ucs , num      (n)
@ucs , avg      (μ)
@ucs , dev†     (σ)
```

## The Recalculation Order is Important

If the direction of recalculation on the spreadsheet is *row-wise*, then the subsequent dependent query questions should lie somewhere to the right of the last-asked standard query question. If the direction is *column-wise*, then the dependent query questions should appear in the column below and close to the standard query question they reference.

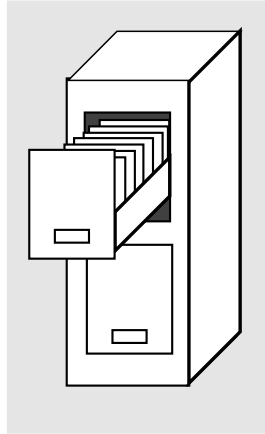
The calculated statistics (called the *current statistics*) remain in effect until either: (1) a new standalone query question, (2) text or (3) numeric equation is executed. Each of these cell types either clears the current statistics or resets them to new values. Intervening blank cells and text labels however have no effect on the current statistics.

---

†Standard deviation is the square root of the variance, which is also calculated (@ucs, var). The calculated variance is corrected by the factor  $n/(n-1)$ , which is commonly called the *unbiased estimator*.

## When a Record is Qualified, It's Marked

When a record is accepted into the accruing statistics of a standard, independent query question, its record number is recorded in a temporary file. *Marking* the records in this manner is a little like dog-earing all of the qualified file folders in the file drawer so that you can find them again.



### Qualified Record Numbers

12	213	363
25	215	373
39	217	374
67	218	375
68	223	382
73	245	423
76	247	456
79	267	512
80	289	523
103	301	578
104	303	580
156	356	•
185	357	•

## The @REREAD Form

The marked records allow the use of the second form of dependent query question: the *reread* of data from a qualified sublist. Rereading data from a qualified sublist often represents a substantial performance improvement over retrieving the data again. The form of the query question is:

```
@rereading, sum of balance
  @rereading, avg of vacation when regular >= 40.0
```

The first example will reread all of the last-qualified records and return the sum of the dataitem *balance*. The second example will reread all of the last-qualified records and return the average of the *vacation* days for only those records which had a *regular* hours greater than or equal to 40.

Rereading a previously qualified list using the @REREAD does not alter the list of qualified records. The current statistics are changed, however, to the new values calculated.

The @REREAD is also valuable in retrieving multiple dataitem values when one record has been isolated, such as obtaining the name, address, city and state of a single person. Dependent @REREAD's would be placed directly after a qualifying query question, as shown in the example at the top of the facing page:

```
@using employees, val of lname
      when socsecnum is [$g34]
@rereading, val of fname
@rereading, val of address
@rereading, val of city
@rereading, val of state
@rereading, val of zip
```

The record of interest is loaded into the HP3000's main memory with the execution of the first query question. Each @REREADING is a low-cost method of retrieving additional information from the same record. Finding the correct record on the HP3000's disc drives is the most "expensive" part of retrieving data. Once the record is in memory, it should be used to the greatest extent possible.

## Multilevel Rereads (Rereading Subsets)

In contrast to the single-record rereads just described, multilevel rereads allow you to progressively isolate records as subsets, thereby minimizing the need for repetitive serial searches. The technique is especially valuable for filling in tables of calculated values which are derived from a single dataset, and is discussed in detail in Chapter 7, "Summary Reports."

The syntax of a multilevel reread is:

```
@level 3 rereading, sum of amount
      when jobnum < 8500
```

where the level may range from 1 to 9.

Each @REREADING rereads the records previously found at the specified level and creates a new list of records one level down, based on the qualifying criteria of the rereading query. @LEVEL 1 REREADING is identical to the standard @REREADING.

Multilevel rereads are a simple but powerful search acceleration mechanism. The speed advantage occurs because each progressively lower-level reread must now read only a fraction of the qualifying records. But just as important, the original source pool of records is not discarded and does not need to be rebuilt by another serial search. A new subset of records can be redefined at any level by returning to that level and asking a new @LEVEL x REREADING query question.

## Query Forms Which Merely Mark Records

All of the query questions to this point in the chapter have summarized values taken from records in a dataset. There are three query question forms which do not create summaries of values. Rather, they merely mark the records so that they may be reread. The form of these query questions is:

```
@find when state is CA
@num when jobnum is 8404 and date>950000
@pct when keyname is G,M
```

The first two forms, @FIND and @NUM, are completely synonymous. You should read the query questions as either "Find me all of the records whose state is California" or "Get me the number of records when the job number is 8404 and the date is in the year 1995 or greater." The value that is returned to the cell using either of these forms is the *number* of records which were found to qualify.

The third form, @PCT, similarly marks all of the records that meet the qualifying conditions, but it returns the *percentage* of records which qualified. If the search was a serial search, the percentage will be the percentage of qualifying records in the entire dataset. If the search was a chained search, the number returned will be the percentage of qualifying records on the chain.

While the utility of only marking records and not returning a summary value may not be obvious at first, the @FIND query question form will prove to be invaluable in user-defined query functions, "UDQF"s (Chap.11), and in detail list reports (Chap.13).

## Viewing the Qualifying Records

All query question forms (other than "using current stats") mark the records which pass the qualifying conditions. These individual records can be viewed after executing a query question by typing @SHOW. The @SHOW command reads down the list of marked records and displays the individual record values, one-by-one. To advance the display to the next record, press the RETURN key. You may stop the list prematurely by typing either Control-Y, " // ", or pressing the [F8] key.

Showing the qualifying records in this way allows you to quickly determine that the records you're finding are the records you want. Various command forms for the @SHOW are shown on the facing page:

## The @SHOW Command

---

To show records found during the last query:

**@SHOW** The show command shows all of the dataitems for all of the qualifying records found during the last database query.

**@SHOW PRODUCTNO,CITY,STATE**

A subset of the dataitems found during the last database query can also be specified to be shown.

To show randomly selected records in a specified dataset:

**@SHOW INVOICES**

**@SHOW INVOICES: AMOUNT, DATE**

Either all or a selected few of the dataitems will be shown for the dataset invoices, record-by-record, in the currently defined (default) database.

**@SHOW QCDEMO.INVOICES**

**@SHOW QCDEMO.INVOICES: AMOUNT, DATE**

Randomly chosen records in a dataset in any open database may also be shown. The specified database becomes the default database.

To show the contents of a search list:

**@SHOW !A** The dataitem values currently held in the specified search list will be displayed.

To show the list of currently open databases:

**@SHOWDB** The currently open databases are displayed. The default database is shown with an arrow ("**<---**<").

---

## Statistical Sampling

Two clauses may be appended to any query question which modify the sampling nature of the query question. The reasons for sampling less than every qualifying record are:

1. The statistical sampling of recorded data for scientific and engineering purposes.
2. To simply limit the number of records drawn during the initial phases of putting a report together.

The modifying clauses are called LIMIT and SAMPLE. They are used in this fashion:

```
@Using database.dataset, avg of thickness
      when jobnum is 8404 and
      machinenum is 15061;limit=300;sample=.4
```

The two clauses are independent. They may be appended to a query question individually or in concert. The order of specification is not important. The LIMIT clause limits the number of records found. The SAMPLE clause specifies the sample frequency. A sample frequency of 0.4 indicates that approximately 40% of the qualifying records will be selected.

If both clauses are added to a query question, the order of execution is as follows: the record is first qualified for acceptance on the basis of whether or not it meets the standard conditions specified in the qualifying phrase. If the record is acceptable, and would normally be added in, a coin is tossed. If the result of the coin flip says that the record should be added in, the limit value is checked. If the number of records is less than or equal to the limit, the record is included in the accruing statistics. If this newest record hits the limit, the query search quits.

If only the SAMPLE clause is used, the entire set of records is searched. (Which records constitute the searched set is determined by the type of search employed, as it is in all query questions. The searched set may be either the entire dataset if the search is a serial search, or it may be only the set of records linked together by a common key value.) The selection of records is random. If you were to search a set of 100 records with a sampling frequency of 0.4, you will not necessarily get exactly 40 records. You may, on any one pass, see 38, 39, 40, 41, or 42 records. This random-

ness is part of the nature of statistical sampling. If you desire equal sample sizes, the LIMIT clause allows you to specify the sample size. There is an important caveat to be considered, however, when both clauses are used simultaneously. This caveat is especially important when pseudoreplicating your data into distinct statistical "trials".

QueryCalc always searches a set in forward-read order. Normally, this order is indicative of the chronological order that the data has been entered (but not always). The first records are generally the oldest; the last records the newest. If the number of records in the set to be searched is 1000, and you specify a limit of 100 and a sample frequency of 0.3, the query will be satisfied by about the 300th record. Repeating the same query question again will randomly select a slightly different 100 records. But you are only reading the first third of the data. It is important to understand that you may be introducing some bias into your samples when you restrict the sample size. Earlier records may be somehow different from later ones.

## Pseudo-Replicating Data

H.G. Wells said that there are only three kinds of liars: (1) liars, (2) damn liars, and (3) statisticians. It's very easy to mislead someone else, and more importantly, yourself with statistics. Virtually everyone who has ever drowned in a river, has drowned in a river whose average depth was about six inches. A reported average without an accompanying variance is meaningless. Processes which generate identical averages may not be representative of the same physical process. One process may generate a mean with very little statistical variance. Such a process is highly predictable. Another process with exactly the same measured mean may have great variance, and thus be basically unpredictable.

*Pseudoreplication* is a simple statistical technique to determine how uniform your data truly is. Calculating the average and deviation of an entire set of records may be misleading if the process which produced the data is not particularly uniform. In the case of measuring the depth of a river, most measurements may be relatively shallow, but a few may indicate great depth. These exceptions are called *outliers*. A single average and deviation may not provide a true picture of the nature of the data. By dividing the sample set up into 10 approx. equal-sized subsets (`sample=0.1`), and recalculating the average and deviation 10 times, you can tell just by looking at the results if your data is uniform or not. Outlier data will randomly appear in some data subsets, but not others. Because of the smaller sample sizes, these outliers will tend to scatter the resulting 10 averages.

## Search Lists

The final query question form is the @STORE list. This query question form is important enough to be explained in detail (Chap. 10, "Search Sets"), so I'll only briefly mention its capabilities here. The @STORE form allows the creation of a set of search item values, which share a common membership in a class. The basic form of the @STORE query question is:

```
@using employees, store in !m socsecnum
    when state is VT
```

This query question will create a search list named M which will contain all of the social security numbers for those people whose home state of residence is Vermont. Twenty-six such search lists, labeled A to Z, may be created. *The search list is simply a list of search item values. It belongs to no dataset or database.* This independence is what gives the search list its value. You may view the contents of a search list by typing:

```
@show !m
```

You will notice that when you display a list, the search item values will be alphabetized. The alphabetization of the search list results from QueryCalc first sorting the list of retrieved values, and then eliminating all duplicate entries. *Search lists will always be composed of unique entries.*

Search lists are used in standard query questions in place of a single search item value. Instead of asking:

```
1. @using payrecord, sum of amount
    when socsecnum is 526687191
    or socsecnum is 585178564 ...
```

you would type:

```
2. @using payrecord, sum of amount
    when socsecnum is !m
```

What occurs in this second form, rather than search for a series of single dataitem values as in the first example, is that dataitem values are now taken one-by-one from the search list. The qualifying records are found, summarized, and added into the accruing statistics for each value in the list.

Please note that even in this simple example, two datasets were used. The item which qualified these social security numbers into a defined class was the state of home residence. The question that is being answered is a request for the sum of the total wages paid to the employees from Vermont, a request requiring information that must come from two different datasets.

## Using Search Lists to Accelerate Serial Reads

An important use of the @STORE form allows you to minimize the use of serial searches. Presume that *date* is a search item in a very large invoices dataset containing 10 years worth of invoices (say 100,000 invoices). Further presume that you need to report on only the invoices for a particular month (31 possible dates). Because *date* is a search item, there will be a master dataset for date, called perhaps DATE-ID. A master dataset, by its nature, contains only one entry for each search item value. Ten years of dates can total no more than 3,653 entries. Thus, it will be much quicker to gather a range of dates into a search class by serially searching a master dataset than it would be by serially reading the large detail dataset.

The two query questions used in such a search would be of this form:

```
@using date-id, store in !a date  
when date ib 19950101,19950131
```

```
@using invoices, sum of amount  
when date is !a
```

The first query question will necessarily be a serial search because of the "is between" relational operator. Although QueryCalc uses the fastest possible method to accomplish its serial reads, the second search will be faster yet. The second query question is a chained read, thus this search will access only those records which contain the desired dates in the search set and will generally be quite quick. The combination of these two techniques provides QueryCalc with a generic search technique that is comparable in speed to the very best third-party indexing techniques, with these added advantages: (1) no additional disk space is required, which is often quite extensive otherwise, (2) no interception of TurboIMAGE intrinsics is required, and perhaps most importantly, (3) the method is free and built into QueryCalc.

The further use of search lists is explained in Chapter 10, "Search Sets".

## USER EXERCISES

---

- |                             |   |
|-----------------------------|---|
| Signing On                  | <ol style="list-style-type: none"> <li>1. Sign onto the practice account by typing <b>HELLO USER.AICS</b> at the colon prompt. Supply any necessary passwords. Passwords, if they are present, were put there by your system manager and he or she will know them if you do not.</li> <li>2. Type <b>QC</b> to run QueryCalc. If this does not work, type <b>RUN QC.QCPROGS</b>. Press <b>RETURN</b> to bypass the instructions screen. You are now in QueryCalc proper.</li> <li>3. Type <b>@OPENDB QCDEMO</b>. Type <b>FRONT</b> (in all caps) in response to the password question. IMAGE databases are the only place in the HP3000 where passwords are case sensitive. You now have the database open.</li> </ol>  |
| Viewing your Open Databases | <ol style="list-style-type: none"> <li>4. Type <b>@SHOWDB</b> to show your open databases. You should see QCDEMO as the only open database. Press <b>RETURN</b> to return to the spreadsheet.</li> <li>5. Type <b>@FORM</b> to see the basic form of QCDEMO. Master datasets (3x5 card-like sets) are always arranged on top of detail datasets (file drawer-like sets). Type <b>LABOR</b> to see the labor dataset. Type <b>JOB</b> to see the JOB dataset. As you'll see, there is no JOB dataset in QCDEMO. QueryCalc will instead show you all of the dataitem and dataset names which are close to JOB.</li> <li>6. Press the <b>[f6]</b> function key to direct your output to your system printer. The system printer is connected to the HP3000 and is generally located in or near the computer room. Press <b>[f1]</b> to print the sets of QCDEMO on your system printer. Press <b>[f4]</b> to print the paths of QCDEMO. Press <b>[f3]</b> to print the entire form of QCDEMO. Press <b>[f8]</b> or " // " to return to the spreadsheet. Retrieve your output from the printer before we go any further.</li> </ol> |
| First Query Question        | <ol style="list-style-type: none"> <li>7. Move the cursor to cell B2. You can do this either by using the function keys or by typing <b>/J B2</b>. Type <b>@sum of amount</b>. Because the dataset isn't specified and because AMOUNT appears in a number of datasets, QueryCalc will ask you which dataset you wish to use. Type either the name <b>INVOICES</b> or the number <b>4</b> for the</li> </ol>   |

## USER EXERCISES

---

INVOICES dataset and press **RETURN**. QueryCalc will announce that the search must be serial, that it must read 9962 records and ask whether or not you wish to proceed. Respond by typing **Y** (yes) or **U** (use). What will appear in the cell will be the phrase, "<ok>". The dashes indicate that your input has passed syntax check, but the cell has not yet been recalculated (see Chapter 4, page 4-8).

### Immediate Execution of the First Query Question

8. Retype **@sum of amount!**, but this time with an **!** at the end. The **!** means execute the query question immediately. Answer the questions in the same manner as Step 7. This time, QueryCalc will not come back to you immediately but will instead display a message saying "Calculating query". Because this is a serial search of nearly 10,000 entries, the serial read may take a minute or two. When the answer does appear, it will be a cell full of asterisks, "\*\*\*\*\*". The asterisks mean is that the answer is too big for the current cell width. Type **/CWID 20**. This will widen the cell from 12 characters to 20. The answer, -3,616,241.26, is negative because both invoices and monies received are entered into the same dataset as positive and negative invoices. If the sum is negative, it means you took in more money than you spent.
9. Press **RETURN** to see the equation that has been entered into the cell. Because QueryCalc operates off of terminals, and because not as much information per second can be transmitted over a terminal's connection as in a personal computer, the cell equations aren't automatically displayed. You must ask for them. What you should see is "@Using qcdemo.invoices, sum of amount". Notice that QueryCalc automatically entered the "Using..." phrase for you and supplied the name of the database.

### Viewing the Individual Records You Found

10. Type **@show** to see the invoices you found with the query question of Step 8. The display will show you all of the dataitem names in the INVOICES dataset, the values for the first record found, and which items are search and sort items. Press **RETURN** to advance the listing. Press either **//**, **Control-Y**, or **[f8]** to return to the spreadsheet.

## USER EXERCISES

---

- |                              |  |
|------------------------------|--|
| A Chained Search             | 11. Now for a chained search. Move the cursor to cell B3 by pressing [f6]. Type <b>@using invoices, sum of amount when category is 501!</b> . The answer that is returned is 23,984.97. Notice the difference in speed from the previous serial search. CATEGORY is a search item in INVOICES. The purpose of a database is to find the data you need as quickly and directly as possible. Using search items in your qualifying phrases (to the extent possible) greatly accelerates data retrieval. Which dataitems are search items are marked on the database @FORM listing we generated in Step 6   |
| Synthesizing Search Patterns | 12. Move the cursor to cell A4. Type <b>"501"</b> . This enters a text value of "501" into the cell. Now move the cursor to cell B4. Type <b>@using invoices, sum of amount when category is [a4]!</b> . The "[...]" construct is a "window" back onto the spreadsheet which allows you to manufacture the search value. Any numeric or text equation may be placed within the brackets. Simply remember that a numeric equation uses no prefix (as in this example) and that a text equation always begins with a "\$". The answer that you get should be the same as in Step 11. The only difference is that the search value came off of the spreadsheet and wasn't hard-coded into the query question itself.. |
|                              | 13. Move the cursor to cell A5. Type <b>"459445550"</b> to enter a social security number into the cell. Move the cursor to cell B5. Type <b>@using labor, sum of regular when socsecnum is [\$a5]!</b> . The answer that should be returned is 1530 hours. This example is similar to Step 12, except that SOCSECNUM is a text dataitem. The pattern synthesized in the brackets must therefore be a text equation. Numbers are commonly stored in text dataitem fields. They may look like numbers, but they're not. They're text characters and they must be treated that way.  |
| Dependent Query Statistics   | 14. Move the cursor to B6. Type <b>@ucs avg!</b> . Although the query question in Step 13 asked only for the sum of regular hours, the average, variance, standard deviation, maximum, and minimum were calculated as well. Retrieving this information by using the current statistics (@UCS) is free. Virtually no system overhead is incurred. Move the cursor to B7 and type <b>@ucs max!</b> .  |

## USER EXERCISES

---

- |   |   |
|---|---|
| Rereading<br>Previously<br>Found<br>Records | 15. Every record that qualified in Step 13 was marked and may be repetitively reread. Doing this is generally much more efficient than finding the records again. Move the cursor to cell B8 and type <b>@rereading, sum of overtime!</b> . The returned answer will be 202 hours of overtime. The set of records you are rereading is the same set of records that is shown if you type @SHOW. Notice that the @UCS query questions did not disturb the list of records found in Step 13. The list of records remains in effect until another standard (non-@REREADING) query question is executed.  |
| Creating and<br>Using Search<br>Lists       | <p>16. Move the cursor to cell B9. Type <b>@using employees, store in m socsecnum when numdeductions is 4,6!</b>. We have just created a search list named M filled with employee social security numbers for those people who claim 4 to 6 deductions in their payroll taxes. As you can see, there are 57 such people. To see the values in this list, type <b>@show !m</b>. @STORE lists will always be composed of unique entries. Duplicate values are eliminated. The people represented by these social security numbers represent a <i>class</i> whose membership is defined by their number of deductions. Press <b>RETURN</b> to return to the spreadsheet.</p> <p>17. To use the list, move the cursor to cell B10. Type <b>@using labor, sum of regular+overtime*1.5 when socsecnum is !m!</b>. This question will sum all of the effective labor hours charged by this group of people (48,555.21 hours). Because SOCSENUM is a search item, this question will search down 57 different chains and add all of the hours together. @STORE lists allow you to collect data and statistics on classes exactly as you would on individuals.</p> |
| More Examples<br>& Using Your<br>Databases  | <p>18. Additional query question exercises exist in Chapter 7, Chapter 10, Chapter 11, Chapter 12, and Chapter 13.</p> <p>To ask these same kinds of question using your own data, you need to learn your own databases. The first step is to repeat Steps 3-6 with names relevant to your databases.</p>   |

## Concepts Introduced in Chapter 6

---

---

STANDARD QUERY	The query question which summarizes data from a single dataset.
"USING..." PHRASE	The first phrase in a query question. The phrase which specifies which database and which dataset is to be used.
RELOPS	The relational operators which match a dataitem value to a data "pattern".
"REREADING, ..."	A dependent query question which rereads the list of records previously qualified by a standard query question.
CURRENT STATS	The various statistics (sum, average, max, min, etc.) summarized by a query question. Although only one statistic may be asked for in a query question, all are calculated.
SUBITEMS	Dataitem values held in a dataitem array. Dataitem arrays are denoted by 30X6, 10R2, etc.
SUBSTRINGS	Specified subsections of a text dataitem's string of characters.

---

---

# InterChapter Reminder

---

All numeric data items are converted to high-resolution real numbers when they are brought into QueryCalc, regardless of how they are stored in your databases. The I, J, K, P and Z datatypes cannot however store "real" numbers (numbers with a decimal point). In order to maintain accuracy to the penny when these datatypes are used, a programming "trick" quite often employed, especially by COBOL application developers, is to "offset" certain data fields by multiplying the number by 100 (or more) before it is entered into the database.

Some report writers, such as HP's QUERY, use an edit mask to correct this offset. The numbers are used in the report writer in their offset fashion. It's only on printing that a decimal point is reinserted to make the number look correct. QueryCalc doesn't do that. A numeric value is presumed to mean what it says. Because of this, data can be extracted and mixed from a great diversity of sources. If you must extract data from an offsetted field, the solution in QueryCalc is straightforward. Divide the value by its offset during extraction. Examples of the syntax for such division is:

```
@using parts, sum of amount/100 when ....  
@using payrecord, sum of \gross-billing\1000 when ....
```

For more information about mathematical operations available in a query question, please refer to pages 6-3, 6-4, 6-5.

---