Paper BB-16

Condensed and Sparse Indexes for Sorted SAS® Datasets

Mark Keintz, Wharton Research Data Services, University of Pennsylvania

ABSTRACT

The standard SAS index normally can speed up data retrieval of subsets, but can be suboptimal for datasets sorted on the index variable. For large sorted groups the normal index wastes disk space by creating a "pointer" to every record in the group. This paper demonstrates a condensed index with pointers only to the first and last record in each group. Dramatic reductions in elapsed time, input/output, and disk space are realized.

The paper will also show a "sparse" index for sorted data in which groups are relatively small but the index range is large (e.g. every millisecond over a span of days). When retrieving a subset of cases between a pair of values of the sort variable, an index that points to only a few selected records can also save disk space and elapsed time.

The paper will show the SAS programs used to create, and make use of these special indexes.

INTRODUCTION

Two types of “compressed” indexes are reviewed in this paper: “condensed” and “sparse”.

The idea for a condensed index came from our need to speed up data retrieval from large files of trades and quotes we get monthly from the New York Stock Exchange (NYSE®). What do we mean by large? For example, the April 2007 NYSE quotes file contains 2.9 billion observations covering about 8,500 stock symbols over 19 trading days. It produced a SAS data set requiring 165 gigabytes of disk storage. These data sets are rapidly getting bigger, doubling in size every 15 to 18 months. More recently, for the single day of April 16, 2010 the quotes files contained 0.7 billion observations. Users extracting relatively small subsets of data were having long waits, even with the benefit of SAS® indexes.

We also noticed that even our SAS® index files were large (71 gigabytes to record two indexes for the April 2007 dataset above), so we developed a smaller index file which we hoped would also speed up data retrieval. This paper describes a “condensed index” design that takes only a fraction of the disk space of a normal SAS index file, and yields significant savings in elapsed time and CPU time. It even noticeably reduces I/O.

More recently, we developed and tested the use of a “sparse” index. Like the condensed index, it is meant for sorted datasets. But while the condensed index is “lossless” (i.e. keeps all the information available in the normal SAS index), the sparse index records index information only for a subset of the records in the dataset. These serve as guides to the “neighborhood” of the desired record extract. As will be shown later, this can be useful when there are too many values of the index variable (say time-of-day recorded to the millisecond) for a condensed index to be effective.

These techniques are not as general as the normal SAS index. Our data sets benefit from the condensed and sparse indexes, because they have the following characteristics:

- The data sets are static. Once indexes are created, no further change will be made.
- Extractions are most frequently based on variables for which a single value defines large subsets of the data (i.e. the value is not highly “discriminant”). In our case, users most commonly specify a relatively small number of stock symbols and/or dates. Even selecting just a single symbol (or date) would yield large numbers of records (e.g. over 100,000). This benefits the condensed index.
- Some extractions want just a single contiguous span of records (benefis the sparse index).
- Our data sets are sorted by the subsetting variables: e.g. stock SYMBOL and DATE for the condensed index, or TIME for the sparse index.

1. SHORT REVIEW OF THE SAS® INDEX

As mentioned above, we encountered SAS index files that were very large. Why does the SAS index take so much space? The primary reason is that the index file tracks each observation in the data file. According to SAS® online documentation (see Understanding SAS Indexes in the **SAS 9.2 Language Reference: Concepts, Second Edition)**:

The index file consists of entries that are organized hierarchically and connected by pointers, all of which are maintained by SAS. The lowest level in the index file hierarchy consists of entries that represent each distinct value for an indexed variable, in ascending value order. Each entry contains this information:

- a distinct value
- One or more unique record identifiers (referred to as an RID) that identifies each observation containing the value. (Think of the RID as an internal observation number.)

In other words, at the lowest level, index entries have a scheme like this:

Figure 1. Index Entries (lowest level) for variable SYMBOL (in unsorted data set)

```
SYMBOL        RID
AA            1,456; 2,234; 4,567; 6,789; … 121,989
…             …
IBM           2,001; 9,945; 13,232; 14,544; … 998,567
…             …
ZZ            557; 12,891; 34,565; 44,650 … 989,456
```

You can see that there is one RID for each observation in the data set, and there is one entry for each value of the index variable. In the case of the April 2007 quotes file mentioned above, there would be about 8,500 entries (unique values of SYMBOL) at the lowest level of the index file, and 2.9 billion RID's. That's about 340,000 RID's per value of SYMBOL.

If the original data set is sorted by SYMBOL, then the index entries would look like Figure 2, in which RID's for each value of SYMBOL form consecutive lists.

Figure 2. Index Entries (lowest level) for variable SYMBOL (in sorted data set)

```
SYMBOL        RID                                       LAST RID
AA            1; 2; 3; 4; 5; …                           210,000
…             …
IBM           1,221,222,101; 1,221,222,102 …       1,222,220,668
…             …
ZZ            2,899,300,001; 2,899,300,002 …       2,900,000,000
```

2. A CONDENSED SIMPLE INDEX FOR SORTED FILES

2.1 Saving storage space with a condensed index

Both of the index files above take up the same amount of storage, but in the second case most of it is superfluous – namely all the RID's between the first and last RID for each entry. An index containing only the first and last RID's would save a lot of space, with no loss of information. Such a *condensed index* could look like the following:

Figure 3. Condensed Index Entries for variable SYMBOL (in sorted data set)

```
SYMBOL        FRID                                          LRID
AA            1                                          210,000
…             …
IBM           1,221,222,101                        1,222,220,668
…             …
ZZ            2,899,300,001                        2,900,000,000
```

Clearly this saves space. The standard SAS index file has **N_O** (number of ***observations*** in the data set) RID's, but the condensed index has **$2*N_i$** (number of ***index values***) RID's. So the design will **save space only if the observations are grouped** (i.e. multiple consecutive observations per each value of the index variable). At the lower limit (one observation for each indexed value, i.e. N_i=N_O), the condensed design would actually use more storage. In the case of the April 2007 data set (N_O=2,900,000,000 and N_I=8,500) the ratio N_i/N_o of RID's per index value will be about 340,000 to 1 (yielding 170,000 FRID/LRID pairs per index value).

Given this design for a condensed index, the primary questions are (1) how can it be used?, and (2) how well does it perform?

2.2 How to use a condensed index

As an example, let's say you want to extract all the observations in data set QUOTES which have SYMBOL="IBM". In the absence of a condensed index, you might submit a program like this:

Example 1: Subset extraction without condensed index

```
DATA ibm_quotes;
  SET quotes;
  WHERE symbol='IBM';
RUN:
```

If a SAS index exists for variable SYMBOL in data set QUOTES, this data step would use it to directly read only the observations with SYMBOL="IBM". To do the equivalent using a condensed index, there are two requirements: (1) an index accessible in a DATA step, and (2) a way to read only the requested observations from QUOTES. To satisfy the first requirement, assume we have another SAS data set, call it IX_SYM, containing three variables, SYMBOL, FRID, and LRID, as in figure 3 We'll demonstrate later how to create IX_SYM.

To satisfy the second requirement, we can use the POINT= option of the SET statement. According to SAS online documentation the "POINT=variable" option:

> Specifies a temporary variable whose numeric value determines which observation is read.
> POINT- causes the SET statement to use random (direct) access to read a SAS data set.

In other words, if you want to read observation 2007 from data set QUOTES, you could simply enter

```
p=2007;
SET quotes POINT=p;
```

within a data step. Note, you can NOT enter the actual numeric value, as in

```
SET quotes point=2007;**This will fail with an error message **;
```

We now have the pieces. Here's how to put them together:

Example 2: Subset extraction with condensed index

```
DATA ibm_quotes (DROP=frid lrid);
  SET ix_sym;             ** Read the IX_SYM ¨driver¨ file **;
  WHERE symbol='IBM';     ** Applies to IX_SYM only **;
  DO p=frid to lrid;      ** For each IBM obs**;
    SET quotes  POINT=p; **directly read the record **;
    OUTPUT;              **and output it**;
  END;
RUN:
```

So what's going on here? The essence of this example is that there are two nested SET statements. The outer SET reads from the condensed index data set IX_SYM. The following WHERE statement keeps only the entry for 'IBM' (note it does NOT apply to the subsequent SET QUOTES statement). We now have FRID and LRID, specifying the range of IBM observations in QUOTES.

Then, the DO statement sets up a loop from FRID to LRID. Using the "POINT=" option, the inner SET statement reads each corresponding observation from QUOTES, and the subsequent OUTPUT statement writes it. The resulting data set, IBM_QUOTES, has the same observations as example 1. Also, it adds no variables to those already in QUOTES: FRID and LRID (from IX_SYM) are eliminated by the "DROP=" parameter, and P is not kept because it is classified as a temporary variable due to its use in the "POINT=" option.

2.3 How well does the condensed index perform?

To evaluate the condensed index we used a data set from the NYSE, representing all quotes for the month of January, 2006. The data set profile is as follows:

- N_O=1,542,953,506 (about 1.5 billion)
- Sorted by SYMBOL
- N_i = 8,371

We then ran retrievals of 10, 100, and 1,000 symbols, using the SAS index and the condensed index. Each retrieval was repeated 5 times, and the averages are shown in Table 1.

Table 1
Resource Use, by Number of Symbols and Index Type
Averages over five runs

	Number of Symbols Requested		
	10	100	1,000
Elapsed Time (mm:ss)			
SAS Simple Index	0:58	1:52	13:32
Condensed Index	0:40	1:16	9:05
Percent Change	-33%	-32%	-33%
CPU Time (in seconds)			
SAS Simple Index	0:56	1:48	12:31
Condensed Index	0:37	1:08	8:12
Percent Change	-35%%	-37%	-35%
Memory Use (Kbytes)			
SAS Simple Index	363	429	1,119
Condensed Index	430	503	1,205
Percent Change	+18%	+17%	+8%

These tests were run on the following platform:

System:	SUN V440, 4 Processors, 8GB ram
Operating System:	Solaris 9

You can see that elapsed time is reduced on the order of 33%. Most of this seems to be due to savings in CPU time (average about 36%). But there is a price to pay - memory use **increases**, although proportionately less (average around 13%) than the decrease in CPU and elapsed time. This makes sense, since there is probably more overhead in compiling and executing program statements to read the condensed index, than in the SAS binary routines used to process the normal SAS index. We also saw minor savings in input/output counts when using the condensed index. Those results are not shown here because we were not able to eliminate the effects of file caching, resulting in wide variations from run to run. Some runs even reported zero block input operations, demonstrating excellent file caching by our computing platform, but rendering input/output counts useless for index comparison.

2.4 How to create the condensed index

Creating the condensed index is a simple, though time consuming, process of reading through the sorted data set, and saving the beginning and ending observation numbers for each value of the index variable. The following program (Example 3) creates the condensed index data set from the original QUOTES data set.

Example 3: Creating the condensed index data set

```
DATA ix_sym;
  frid=1;               **Initialize FRID for 1st symbol**;
  DO UNTIL (lastcase);  **Stop after entire dataset is read in **;
    DO lrid=frid by 1 UNTIL (last.symbol);
      SET quotes (keep=symbol) END=lastcase;
      BY symbol;        **The data set MUST be sorted by SYMBOL**;
    END;
    OUTPUT;             **At end of each SYMBOL, output FRID & LRID**;
    frid=lrid+1;        ** Update FRID for the next SYMBOL **;
  END;
RUN:
```

3. CONDENSED COMPOSITE INDEX FOR SORTED FILES

To support subset selection such as

```
WHERE symbol='IBM' and date between '10jan06'd and '13jan06'd
```

a SAS composite index based on combinations of SYMBOL and DATE is typically used.

Given that the condensed index provided significant improvement over the simple SAS index on SYMBOL, we created and tested condensed index files meant to substitute for a composite index. Using the same data set QUOTES as above, but specifying that it is sorted by SYMBOL and DATE, we created a condensed index data set named IX_SYMDAT, shown in Figure 4.

Figure 4. Condensed Index File IX_SYMDAT, based on SYMBOL and DATE

SYMBOL	DATE	FRID	LRID
AA	03JAN06	1	11,789
AA	04JAN06	11,790	21,843
…	…	…	…
AA	31JAN06	191,001	210,000
…	…	…	…
IBM	03JAN06	1,221,222,101	1,221,274,657
IBM	04JAN06	1,221,274,658	1,221,339,558
…	…	…	…
IBM	31JAN06	1,222,004,101	1,222,220,668
…	…		
ZZ	31JAN06	2,898,300,001	2,900,000,000

IX_SYMDAT has the same structure as IX_SYM, with one exception: it has two index variables (SYMBOL and DATE) instead of one. For each SYMBOL/DATE combination, FRID and LRID indicate the first and last corresponding observations in the QUOTES data set.

We also created a second higher-level index file, IX2_SYM. But instead of containing first and last RID's for the QUOTES data set, IX2_SYM contains pointers to the IX_SYMDAT data set, as in Figure 5.

Figure 5. Condensed Index File IX2_SYM

SYMBOL	FRID1	LRID1
AA	1	20
…	…	…
IBM	112,001	112,020
…	…	…
…	…	…
ZZ	169,981	170,000

3.1 How to use the hierarchical composite index

Taken together, IX2_SYM and IX_SYMDAT form a hierarchical index of QUOTES. IX2_SYM is a condensed index of IX_SYMDAT, and IX_SYMDAT is a condensed index of QUOTES. We can use the upper-level index (IX2_SYM) to directly read only the subset in the bottom level index (IX_SYMDAT) that contains the desired SYMBOL value. From those IX_SYMDAT records we can keep only the ones which fall in the desired date range (e.g. 10jan06 through 13jan06). In turn we read only the QUOTES records that satisfy both the SYMBOL and DATE constraints, as in Example 4: Note that the example uses an IF statement, because a WHERE statement cannot be used in combination with the "POINT=" option of the SET statement.

Example 4: Using hierarchical condensed index data sets

```
DATA ibm_quotes (DROP=frid: lrid:);
  SET ix2_sym;
  WHERE symbol='IBM';
  DO p1=frid1 to lrid1;
    SET ix_symdat POINT=p1;
    IF '10Jan 06'd <= date<='13Jan 06'd THEN DO p=frid to lrid;
      SET quotes POINT=p;
      OUTPUT;
    END;
  END;
RUN:
```

3.2 Performance results for the hierarchical composite index

To see how well the hierarchical index works, we reran the tests above twice, once selecting quotes from a single day, and once from a 10-day period. The results are in Table 2:

Table 2
Resource Use, by Number of Symbols, Date Range, and Index Type
Averages over five runs

	Date Range					
	1 Day			10 Consecutive Days		
	Number of Symbols			Number of Symbols		
	10	**100**	**1,000**	**10**	**100**	**1,000**
Elapsed Time (mm:ss)						
SAS Composite Index	0:02.6	0:05.1	0:39	0:33	0:56	7:09
Condensed Index	0:01.7	0:03.3	0:28	0:22	0:40	5:18
Percent Change	-33%	-37%	-28%	-33%	-29%	-26%
CPU Time (in seconds)						
SAS Composite Index	0:02.4	0:04.7	0:35	0:32	0:54	6:28
Condensed Index	0:01.6	0:03.0	0:24	0:20	0:35	4:45
Percent Change	-35%	-37%	-30%	-36%	-35%	-27%
Memory Use (KBytes)						
SAS Composite Index	364	433	1,166	364	439	1,166
Condensed Index	521	594	1,296	521	594	1,296
Percent Change	+43%	+37%	+11%	+43%	+35%	+11%

Savings in CPU time and elapsed time for the composite index are about the same as for the simple index, averaging around 33% for elapsed time and CPU time. Again there is a trade-off with memory use, although its relative increase goes down as the number of SYMBOL values increase (from 43% for 10 symbols to 11% for 1,000 symbols). But the date range used for the extraction has no influence on memory use.

4. DIRECT ACCESS OF SUBSETS OFTEN CAN BE BEATEN BY SEQUENTIAL ACCESS

Although using the POINT= technique in a loop driven by the condensed index is faster than the SAS index, it 's really benefitting only from reduced processing of the index values. It's still not taking full advantage of the fact that large blocks of consecutive records are being read with each disk input operation. That is, if you are reading records 12,000,001 through 13,000,000, the program, at its core is doing this:

Example 5a: Basic structure for retrieving a range of records using a DO loop.

```
DATA ibm_quotes;
  frid=12000001;  lrid=13000000;
  DO p=frid to lrid;
      SET quotes POINT=p;
      OUTPUT;
  END;
  DROP frid lrid;
RUN:
```

But SAS can probably do this faster, if instead, you used this:

Example 5b: Basic structure for retrieving a range of records using FIRSTOBS and OBS

```
DATA ibm_quotes;
  SET quotes (firstobs=12000001 obs=13000000);
RUN:
```

The difference here is that in Example 5b SAS sets the reading limits at the compile phase – i.e. the SAS compiler "knows" that one million <u>consecutive</u> records will be read. In Example 5a, there is repeated implementation of the POINT= overhead. SAS only "knows" to read a particular record when the pointer identifies it. For each record, SAS must calculate the value of P and identify the corresponding record in the dataset before retrieving the data. In Example 5b, SAS simply reads the "next" record, until it reaches OBS.

Of course, in **our** case, there are a couple of problems: (1) we typically have multiple ranges (i.e. multiple trading symbols) and (2) because the FIRSTOBS and OBS values are required in the compile phase, the range of needed records must be determined before the data retrieval step begins. This means some macro programming will be needed to generate the needed DATA step.

The first problem is trivial –simply include multiple references to the data set in the SET statement, as here:

Example 6. Multiple object of the SET statement, with OPEN=DEFER.

```
 SET quotes (firstobs=12000001 obs=13000000)
     quotes (firstobs=22000001 obs=23000000)
     OPEN=DEFER;
```

Note the “open=defer” tells SAS not to waste memory by building a buffer for every time a data set is listed in the SET statement. Instead, when the first data set is completely processed, the released memory buffer space is reused for the next object. Without this capability, the program could easily exhaust memory and fail.

To deal with the second problem, we essentially have to run a preliminary DATA step to read the index file and prepare the arguments of the SET statement. In a modification of Example 2, we generate a collection of sequential accesses, for IBM and DELL, in Example 7a:

Example 7a: Using Condensed index to generate sequential access parameters

```
DATA _null_;
  RETAIN r_list $32700;
  SET ix_sym  END=lastix;    ** Read the IX_SYM ¨driver¨ file **;
  WHERE symbol=’IBM’ or symbol=’DELL’;
  range=  catx(‘ ‘,cats(‘(firstobs=’,frid),cats(‘obs=’,lrid,’)’));
  r_list=catx(‘ ‘,r_list,’quotes’,range);
  IF lastix THEN CALL SYMPUT(‘set_list’,trim(range_list));
RUN:
DATA ibm_dell_quotes;
   SET &set_list  OPEN=DEFER;
RUN:
```

This technique transforms the single data step in example 2 to a pair of data steps. The first reads the condensed index with the purpose of building a list of dataset ranges. The list is put into a macro variable (SET_LIST) which is then used in the second data step. The second data step, after the macrovar SET_LIST is resolved, looks like this:

Example 7b: Resolved SAS Script Resulting from Example 7a

```
DATA ibm_dell_quotes;
   SET quotes (firstobs=1001578298  obs=1001729694)
       quotes (firstobs=1290224462  obs=1290246907)
        OPEN=DEFER;
RUN:
```

4.1 Performance results of sequential access controlled by condensed index

And it works. Table 3 shows further gains by moving from direct access to sequential access, in both cases using the condensed index. Like the initial comparison to the SAS index, both clock time (10% to 21%) and especially CPU time (about 19% to 38%) are saved, at the expense of memory. When compared to the ordinary SAS index, of course, results will be are even greater (approaching 50%).

Table 3
Resource Use, by Number of Symbols, Date Range
SET with POINT= vs. (FIRSTOBS ... OBS)
Averages over five runs

	Date Range					
	10 Consecutive Days			Entire Month		
	Number of Symbols			Number of Symbols		
	10	100	1,000	10	100	1,000
Elapsed Time (mm:ss)						
SET + POINT=	0:21.1	0:40.0	5:03	0:43.3	1:23	9:45
SET + FIRSTOBS/OBS	0:17.9	0:36.0	4:10	0:35.7	1:05	7:50
Percent Change	-15%	-10%	-18%	-18%	-21%	-20%
CPU Time (in seconds)						
SET + POINT=	0:18.9	0:35.6	4:45	0:39.4	1:14	8:36
SET + FIRSTOBS/OBS	0:15.3	0:29.7	2:56	0:29.2	0:54	6:22
Percent Change	-19%	-17%	-38%	-26%	-27%	-26%

Table 3
Resource Use, by Number of Symbols, Date Range
SET with POINT= vs. (FIRSTOBS ... OBS)
Averages over five runs

	Date Range					
	10 Consecutive Days			**Entire Month**		
	Number of Symbols			**Number of Symbols**		
	10	**100**	**1,000**	**10**	**100**	**1,000**
Memory Use (KBytes)						
SET + . POINT=	2,828	2,901	3,609	2,697	2,771	3,479
SET + FIRSTOBS/OBS	4,208	4,282	5,374	4,199	4,274	5,425
Percent Change	+49%	+48%	+49%	+56%	+54%	+56%

4.2 Accommodating very long range lists

The program in examples 7a and 7b are a simplified version of what was used to produce Table 3. In particular, the program would fail in retrieving data for 1,000 symbols, since the corresponding ranges would yield a SET statement longer than its maximum allowable length (32,767). But this can be addressed by generating multiple SET statements, ending up with a program such as below:

Example 8: Using multiple SET statements, each with multiple ranges.

```
DATA MY_QUOTES;
  DO UNTIL (last1);
    SET quotes (firstobs=... obs=...) quotes (firstobs=... obs=...)
        quotes (firstobs=... obs=...) quotes (firstobs=... obs=...)
        ...
        OPEN=DEFER END=last1;
    OUTPUT;
  END;
  DO UNTIL (last2);
    SET quotes (firstobs=... obs=...) quotes (firstobs=... obs=...)
        quotes (firstobs=... obs=...) quotes (firstobs=... obs=...)
        ...
        OPEN=DEFER END=last2;
    OUTPUT;
  END;
RUN:
```

The major new detail in example 8 is that it incorporates the SET QUOTES statements in explicit loops, each with a corresponding OUTPUT statement. If these loops were not used the data set would retrieve data out of order, and prematurely stop when the set statement with the shorter range reached its end.

The program that generates the code in Example 8 is below. It uses the CALL EXECUTE statement to generate the data step above, which SAS executes immediately upon completing the DATA _NULL_ step.

Example 9: Generate sequential access parameters for multiple SET statements.

```
DATA _null_;
  array r_list {10} $32767 _temporary_;
  RETAIN R 1;
  SET ix_sym  END=lastix;        ** Read the IX_SYM ¨driver¨ file **;
  WHERE symbol in (LIST OF 1,000 SYMBOLS HERE);
  range = catx(‘ ‘,cats(‘(firstobs=’,frid),cats(‘obs=’,lrid,’)’));
  r_list{r}=catx(‘ ‘,r_list{r},’quotes’,range);
  IF length(r_list{r}) > 32500 THEN r=r+1;
  IF lastix THEN DO;
    CALL EXECUTE (‘DATA MY_QUOTES;’);
    DO S = 1 to R;
      l_text=cats(‘last’s);
      CALL EXECUTE (cat(‘  DO UNTIL(‘, l_text, ‘);’));
      CALL EXECUTE (cat(‘SET ‘,trim(r_list{s}),’OPEN=DEFER END=’,l_text,’;’);
      CALL EXECUTE (‘OUTPUT;’);
      CALL EXECUTE (‘END;’);
    END;
    CALL EXECUTE (‘RUN:’);
  END;
RUN:
```

5. DEVELOPMENT OF THE “SPARSE” INDEX

The primary reason that the condensed index is so effective is the fact the data are sorted and have relatively few (about 8,500 SYMBOL and 20 dates) values of the index variable(s), relative to the number of records in the data set. I.e. each index level accounts for a large number of records. This advantage is reduced when the file is sorted and indexed a variable such as time-of-day to the nearest millisecond (the examples below are restricted to data for a single day, so the DATE variable is not relevant). Even during the normal trading hours (from 09:30AM to 4:00PM) there are over 23 million possible values for TIME measured to the millisecond. As a result even the busiest trading day (with 700 million quotes) will yield an average of only 30 records per time value.

One approach to this problem might be to record and save a condensed index record for only a selected, standardized subset of values, i.e. a “sparse” set of values. In the case of the TIME variable, we could generate an FRID and LRID for, say, say every half hour (e.g. 00:00:00.000, 00:30:00.000, 01:00:00.000, …. 24:00:00.000). Then if a user wanted all the quotes between two standardized value (e.g. 09:30:00 to 10:00:00 - the first half hour of a trading day), we could develop a DATA step with a set statement such as:

```
SET quotes (firstobs=FRID1 obs=LRID2)
```

where FRID1 is the FRID for 09:30:00 and LRID2 is the LRID for 10:00:00 (we assume requested boundaries on the exact second, so no millisecond values are written here). This would work fine *if there were records in the original file for the exact time value 09:30:00 and 10:00:00*, which can not be guaranteed. So in the case of the sparse index we revise the concept of FRID to mean the “*first record id on or after* the given time value (call it FRIDOA). Similarly we create a variable LRIDOB (*last record id on or before* the given time value), so that

```
SET quotes (firstobs=FRIDOA1 obs=LRIDOB2)
```

where FRIDOA1 is the FRIDOA for 09:30:00 and LRIDOB2 is the LRIDOB for 10:00:00, and come from a constructed dataset ISX_TIME, which looks like figure 6:

Figure 6. Sparse Index File ISX_TIME

TIME	FRIDOA	LRIDOB
00:00:00.000	1	.
00:30:00.000	1	.
01:00:00.000	1	.
…	…	…
04:00:00.000	1	.
…	…	…
09:30:00.000	4,639,918	4,706,899
10:00:00.000	60,194,750	60,239,539
10:30:00.000	101,695,799	101,755,626
…	…	…
16:00:00.000	513,721,715	513,723,159
…	…	…
23:30:00.000	.	514,800,605
24:30:00.000	.	514,800,605

There are a number of differences in the structure of the spare index file above from the condensed index file (say figure 5):

1. There are missing LRIDOB values for the early time points. This is due to the revised definition used for LRID, namely it now identifies the last record ***on or before*** the time value, not strictly on the time value. On this particular date the first record occurred at 04:00:05AM, so there were NO record on or prior to midnight, 12:30AM,… through 4AM. Similarly the last record on this date occurred at 20:00:01, so for 10:30PM and later there would be no "on or after" records, yielding LRIDOA=missing.
2. The FRIDOA value does not change over the first few time points. This is also due to the definition revision. Since the first record was at 04:00:05, it is the first record on or after every prior time point (from 00:00:00 through the 04:00:00 point). The same principle applies to the repeated value of LRIDOB for the later time points. This could also happen in the middle of the day if trading were temporarily suspended.
3. The FRIDOA value for a given time is not equal to 1+LRIDOB for the prior time point. This is, of course, the main result of creating a **sparse** index. Had every index value for TIME been kept, there would be no "holes" in the RID values, nor would the revised FRIDOA and LRIDOB variables be needed.

6. USE OF THE SPARSE INDEX

Using this index can be relatively trivial if a user requests all the quotes between any two of the standard time values (i.e. values in the sparse index file). For a sparse index file ISX_TIME and a single-day QUOTES file (sorted by TIME), the program to retrieve all cases from the first 09:30:00.000 record through the last 10:00:00.000 record is as follows:

Example 10: Using Sparse Index to drive retrieval of Quotes Between Two "Standard" Times.

```
DATA _null_;
  RETAIN fobs .;
  IF time='09:30:00't THEN fobs=fridoa;
  IF time='10:00:00't THEN DO;
    IF fobs=. or lridob=. THEN DO;
      lobs=0;  fobs=1;
    END;
    CALL SYMPUT('fobs',cats(fridoa));
    CALL SYMPUT('lobs',cats(lridob));
    STOP;
  END;
RUN:
DATA final;
  SET quote (firstobs=&fobs  obs=&lobs);
RUN:
```

Note that the program checks to see if there actually are valid end point values, and if not, makes FIRSTOBS and OBS parameters that will generate an empty dataset FINAL with no error messages.

But example 10 obviously does not address the task of retrieving the needed records if one wants data between two non-standard times (i.e. not in the sparse index. In this case, think of the data retrieval as having three parts:

1. A middle component constructed by the FRIDOA and LRIDOB of a set of indexed time values. This group needs no further test on the TIME value. It is "closed" at both ends (includes all records from start to finish).
2. A "leading" component which reads the range of records immediately preceding the middle. This needs a test to delete any records preceding the earliest desired time. It is "open" at the start (might not include the earlier observations) and closed at the finish.
3. A "trailing" component which read the range of records immediately following the middle. This also needs a test to stop adding records as soon as it reaches a TIME later than the desired ending time value. It is "closed" only at the start.

For instance, if you want all the records from 09:45:00 through 10:45, the middle component would be based on the 10:00 and 10:30 index values, the leading component would start with the 09:00 records (ending just before the 10:00 records), the trailing component would start just after the 10:30 records and end just before the 11:00 records:

Example 11: Retrieving Data in Three Components Constructed from Sparse Index

```
 DATA final;
   ** Read the leading component **;
   DO UNTIL (end_of_lead);
     ** MacroVar FOBS_LEAD is the FRIDOA for 09:30,        **;
     ** MacroVar OBS_LEAD is 1 less than FRIDOA for 10:00  **;
     SET quotes (firstobs=&fobs_lead obs=&obs_lead) END=end_of_lead;
     IF time < "09:45:00"t THEN continue;
     OUTPUT;
   END;

  ** Read the middle component **;
  DO UNTIL (end_of_mid):
    ** MacroVar FOBS_MID is the FRIDOA for 10:00           **;
    ** MacroVar OBS_MID is the LRIDOB for 10:30            **;
    SET quotes (firstobs=&fobs_mid obs=&obs_mid) END=end_of_mid;
    OUTPUT;
  END;

  ** Read the trailing component **;
  DO UNTIL (end_of_trl);
    ** MacroVar FOBS_TRL is 1 + LRIDOB for 10:30           **;
    ** MacroVar OBS_TRL is 1 less than FRIDOA for 11:00    **;
    SET quotes (firstobs=&fobs_trl obs=&obs_trl) END=end_of_trl;
    IF time > "10:45:00"t THEN STOP;
    OUTPUT;
  END;
RUN:
```

Unlike the condensed index, the sparse index-based program requires at most 3 dataset buffers, because instead of accessing non.contiguous blocks of records (i.e. various trading SYMBOLs), it requires no more than thee record blocks, eliminating the need to cofntemplate the "OPEN=DEFER" option.

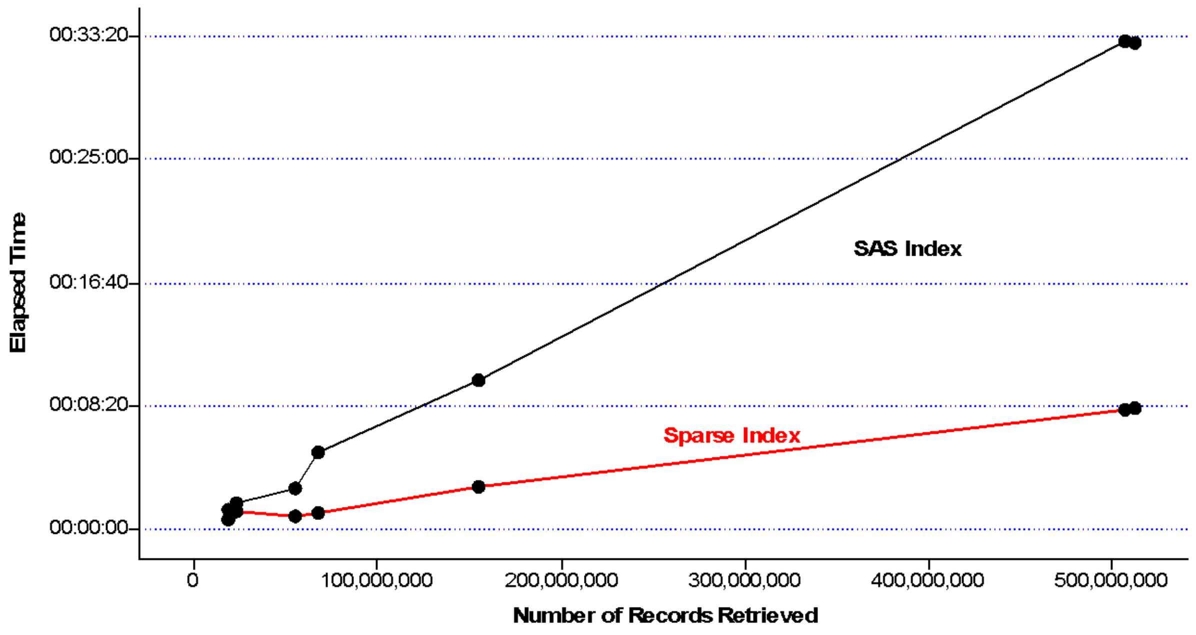
The program used to generate the macrovars FOBS_LEAD (starting record for the lead group), OBS_LEAD (finishing record), FOBS_MID, OBS_MID, FOBS_TRL, and OBS_TRL (plus one other not shown in this example) is in the appendix. It effectively determine the record intervals between consecutive sparse index entries, and then determines which intervals are to be used for the leading, middle, and trailing components of the DATA step.

7. RESULTS FOR SPARSE INDEX VS. SAS INDEX

The sparse index was tested against the SAS index on a Window7 64-bit machine. East test was preceded by a system reboot to eliminate any effect of disk caching. The dataset used was a single day (03/20/2009) of quotes (NOBS=514,800,604). The elapsed time results appear in the graph below.

Figure 7

Comparision of SAS Index vs. Sparse Index



These results show that the sparse index typically takes about 25% of the elapsed time as the SAS index. Memory use never varied more than 10%, and I/O was not reported in this test. In the "SAS Index" test runs (using a simple "WHERE time BETWEEN … and …" statement) the use of the index was optional. Interestingly, SAS reported applying the index in every case, even when retrieving 512 million records (99% of the dataset). This appears contrary to the SAS documentation which suggests that SAS typically declines to use the index when it estimates that more than about one-third of the data will be retrieved (page 518 of the SAS 9.2 Language Reference: Concepts, 2nd. Ed.). It may be that, because SAS recognizes the data set is sorted, it does not use the "one-third" rule.

8. WEAKNESSES AND NOTES ON THESE COMPRESSED INDEXES

The fact that the condensed and sparse indexes are faster in these tests does not indicate any deficiency in the normal SAS index, which was designed to work in far more general environments. There are several conditions supported by the SAS index that would make either compressed index impossible or useless. Some of the more important ones are below:

1. The data set is dynamic. The SAS index is automatically updated when observations are added or removed from an indexed dataset.
2. SAS procedures use SAS indexes. Just putting a WHERE statement in any procedure will potentially use a SAS index, often eliminating the need for a separate subset extraction. This problem could be mitigated by applying the SAS procedure to a data VIEW constructed via a compressed index.
3. The number of values of the index (N_I) approaches the number of observations in the data set (N_O). As the ratio of N_I/N_O grows, at some point the size of the condensed index will no longer provide any significant advantages over the SAS index. Of course, the sparse index is more resistant to this problem.

However, there are some potential benefits to the condensed index and sparse indexes that have not been explored in this paper, such as:

1. Given the small amount of space taken by the condensed index, it would be "cheap" (in terms of disk space) to add condensed indexes sorted by the DATE variable. Even though the original data set is

sorted by SYMBOL/DATE, these indexes could be used to create an extract sorted by DATE/SYMBOL in a single DATA step, without calling a SORT procedure.

2. Use of the condensed index could support removal of the sort variable(s) from the data set, providing further disk space savings with no loss of information or (perhaps) performance.
3. Advance estimating of extractied data set size. Because the condensed index essentially contains a frequency table of the index variables, precise advanced estimation of extract size is easy. This can help when deciding whether an extract is so large that sequential access is more efficient than utilizing an index. Even the sparse index can be used to estimate a minimum and maximum size for the extracted data set.
4. For some datasetd, it may be reasonable to contruct a combination condensed/sparse index.

CONCLUSIONS

If you have a large, static, sorted (on the index variables) data set that has few index values relative to the number of observations, you should consider using a condensed index for extracting subsets. The programming is relatively simple, the savings in disk space can be striking, and elapsed time and CPU time can drop significantly. The price for this is increased memory use, which appears to be relatively small for larger extracts.

Even in the case where there are a relatively large number of index values, using a sparse dataset can improve data retrieval performance by a factor of 4.

ACKNOWLEDGMENTS

Thanks are due to my former colleague Shuguang Zhang, with whom I coauthored the initial paper, of which this is an extension.

CONTACT INFORMATION

This is a work in progress. Your comments and questions are valued and encouraged. Please contact the author at:

Author:	Mark Keintz
Address:	Wharton Research Data Services
	305 St. Leonard's Court
	3819 Chestnut St
	Philadelphia, PA 19104
Work Phone:	215.898.2160
Fax	215.573.6073
Email	mkeintz@wharton.upenn.edu

APPENDIX

SAS program for retrieving data via the sparse index:

```
%let ST=09:15:00;   %* Start Time  **;
%let FT=11:15:00;   %* Finish Time **;

** ************************************************ **
** From the Sparse Index, make dataset of intervals  **
** using consecutive sparse index records            **
** ************************************************ **;
data _intervals;
  set isx_time;
  set isx_time (firstobs=2
    rename=(time=nxt_time fridoa=nxt_fridoa lridob=nxt_lridob));
run;

** Now generate these macrovars:  **************************
** FOBS_LEAD, OBS_LEAD :firstobs, obs for leading segment. **
** FOBS_MID,  OBS_MID  :firstobs, obs for middle segment.  **
** FOBS_TRL,  OBS_TRL  :firstobs, obs for trailing segment.**
** TYPE_LEAD : Indicate OPEN-CLOSE lead group vs OPEN-OPEN.**
**                                                         **
** Initialize the "firstobs" to 1, and the "obs" to 0      **
** ****************************************************** **;
%let fobs_lead=1;  %let obs_lead=0;  %let type_lead=xx;
%let fobs_mid=1;   %let obs_mid=0;
%let fobs_trl=1;   %let obs_trl=0;

data _null_;
  ** See if a single TIME..NXT_TIME interval is enough    **;
  do until (end1);
    set _intervals end=end1;

    ** Ignore non-qualifying time intervals               **;
    if time > "&ST"t or nxt_time < "&FT"t then continue;
    single_timerange_adequate='Y';

    ** Ignore intervals with no internal records          **;
    if fridoa=. or nxt_lridob=. then continue;

    ** Make a "TYPE" for this single interval solution:   **
    **   "OC"<=>Open on Left, Closed on right             **
    **   "CC"<=>Closed on Left, Closed on right           **
    **   "CO"<=>Closed on Left, Open on right             **
    **   "OO"<=>Open on Left, Open on right               ** ;

    if time < "&ST"t then do;      ** "Open" at start **;
      if      "&FT"t = nxt_time then INTTYPE='OC';
      else if "&FT"t < nxt_time then INTTYPE='OO';
    end;
    else if time="&ST"t then do;  ** "Closed" at start **;
      if      "&FT"t < nxt_time then INTTYPE='CO';
      else if "&FT"t = nxt_time then INTTYPE='CC';
    end;

    ** Depending on INTTYPE write out the endpoints for  **
    ** the appropriate segment                           **;

     select (inttype);
      when ('CC') do;
      ** If closed on both sides, it is a MIDDLE segment **;
      call symput('fobs_mid',cats(fridoa));
      call symput('obs_mid',cats(nxt_lridob));
    end;
    when ('CO') do;
    ** If closed only at start, a TRAILING segment     **;
      call symput('fobs_trl',cats(fridoa));
      call symput('obs_trl',cats(nxt_lridob));
    end;
    otherwise do;
    ** Otherwise it is a LEADING segment               **;
      call symput('type_lead',trim(inttype));
      call symput('fobs_lead',cats(fridoa));
      call symput('obs_lead',cats(nxt_lridob));
    end;
  end;
end;

if single_timerange_adequate='Y' then stop;

** If single interval not adequate, condense all    **
** contiguous ranges within request, build a        **
** "middle" segment, generating FOBS_MID & OBS_MID  **;
do until (end2);
  set _intervals end=end2;
  if ("&ST"t <= time and nxt_time <= "&FT"t) then do;
    fobs_mid=min(fobs_mid,fridoa);
    obs_mid=max(obs_mid,nxt_lridob);
    stime_mid=min(stime_mid,time);
    ftime_mid=max(ftime_mid,nxt_time);
  end;
end;
** Now write out endpoints for the middle segment   **;
if nmiss(fobs_mid,obs_mid)=0 then do;
  call symput('fobs_mid',cats(fobs_mid));
  call symput('obs_mid',cats(obs_mid));
end;

** If middle segment exactly satisfies request, stop**;
if stime_mid="&ST"t and ftime_mid="&FT"t then stop;

** If needed, build lead group, from the interval   **
** containing ST. Do this if either no middle group **
** (fobs_mid=.) or middle group begins after ST     **;

if (fobs_mid=. or "&ST"t < stime_mid) then do until (end3) ;
  set _intervals end=end3 ;
  if (time<= "&ST"t <= nxt_time) then do ;
    if fridoa=. or lridob=. then continue;
    fobs_lead=fridoa;
    if n(fobs_mid)=1 then obs_lead=fobs_mid-1;
    else obs_lead = nxt_lridob;
    INTTYPE='OC';
    call symput('fobs_lead',cats(fobs_lead));
    call symput('obs_lead',cats(obs_lead));
    call symput('type_lead',trim(INTTYPE));
  end;
end;

** If needed, make trailing group, from the         **
** interval containing FT.  Do this is either no    **
** middle grup (obs_mid=.) or middle group stops    **
** before FT.                                       **;

if obs_mid=. or "&FT"t > ftime_mid then do until (end4);
  set _intervals end=end4;
```

```
      if (time <= "&FT"t <= nxt_time) then do;
      ** Ignore intervals with no internal records      **;
      if fridoa=. or lridob=. then continue;

      obs_trl=nxt_lridob;
      if      n(obs_mid)=1  then fobs_trl=obs_mid+1;
      else if n(obs_lead)=1 then fobs_trl=obs_lead+1;
      else                     fobs_trl=fridoa;
      call symput('fobs_trl',cats(fobs_trl));
      call symput('obs_trl',cats(obs_trl));
    end;
  end;

  stop;
run;

  DATA final;
    IF &obs_lead > 0 THEN SELECT ("&type_lead");
      WHEN ("OC") DO;
        DO UNTIL (end_of_lead);
          SET taq.cq_30_ts (obs=&obs_lead firstobs=&fobs_lead) END=end_of_lead;
          IF time < "&st"t THEN continue;
          OUTPUT;
        END;
      END;
      WHEN ("OO") DO;
        DO UNTIL (end_of_lead);
          SET taq.cq_30_ts (obs=&obs_lead firstobs=&fobs_lead) END=end_of_lead;
          IF time < "&st"t THEN continue;
          IF time > "&ft"t THEN leave;
          OUTPUT;
        END;
      END;
    END;

    IF &obs_mid > 0 THEN DO;
      DO UNTIL (end_of_mid);
        SET taq.cq_30_ts (obs=&obs_mid firstobs=&fobs_mid) END=end_of_mid;
        OUTPUT;
      END;
    END;
    IF &obs_trl > 0 THEN DO;
      DO UNTIL (end_of_trl);
        SET taq.cq_30_ts (obs=&obs_trl firstobs=&fobs_trl) END=end_of_trl;
        IF time > "&ft"t THEN STOP;
        OUTPUT;
      END;
    END;
    STOP;
  RUN;
```