Paper 024-2009

A Faster Index for Sorted SAS® Datasets

Mark Keintz
Wharton Research Data Services, University of Pennsylvania
Philadelphia PA

ABSTRACT

In a NESUG 2007 paper with Shuguang Zhang, I demonstrated a compressed index (termed "condensed index" in that paper) which provided significant performance gains (about 33% elapsed time and cpu time) in retrieving subsets from sorted datasets in which each level of the sort variables(s) has many observations, and the sort variable was used as the selection criterion. This paper extends the compressed index in two ways: (1) replication of a SAS composite index, and (2) further performance gains (up to 50%) by selectively replacing direct access (i.e. POINT=) techniques with sequential access (FIRSTOBS and LASTOBS).

This paper demonstrates, with sample DATA steps, how to create and use a compressed index, and compares its performance to SAS indices, and to our previous results. Other advantages such as precise prediction of the number of retrieved records will be discussed.

This paper assumes the reader has an intermediate level of familiarity with the SAS DATA step. The use of the "point=" operand of the SET statement will be briefly introduced and will be central to the paper. Some macro coding will be used as well.

INTRODUCTION

The idea for a compressed 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. 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 dataset above), so we developed a smaller index file which we hoped would also speed up data retrieval... This paper describes a "compressed 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.

This technique is not as general as the normal SAS index. Our data sets benefit from the compressed index, 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).
- Our data sets are sorted by the subsetting variables: stock symbol and date.

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 http://support.sas.com/onlinedoc/913/getDoc/en/lrcon.hlp/a000440261.htm):

> 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 COMPRESSED SIMPLE INDEX FOR SORTED FILES

2.1 Saving storage space with a compressed 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 *compressed index* could look like the following:

Figure 3. Compressed 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 $\mathbf{N_O}$ (number of ***observations*** in the data set) RID's, but the compressed index has $\mathbf{2*N_i}$ (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 compressed 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 compressed index, the primary questions are (1) how can it be used?, and (2) how well does it perform?

2.2 How to use a compressed 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 compressed index, you might submit a program like this:

Example 1: Subset extraction without compressed 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 compressed 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 compressed 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 compressed 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 compressed index perform?

To evaluate the compressed 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 compressed 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
Compressed Index	0:40	1:16	9:05
Percent Change	-33%	-32%	-33%

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

	Number of Symbols Requested		
	10	100	1,000
CPU Time (in seconds)			
SAS Simple Index	0:56	1:48	12:31
Compressed Index	0:37	1:08	8:12
Percent Change	-35%%	-37%	-35%
Memory Use (Kbytes)			
SAS Simple Index	363	429	1,119
Compressed 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 compressed 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 compressed 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 compressed index

Creating the compressed 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 compressed index data set from the original QUOTES data set.

Example 3: Creating the compressed 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. COMPRESSED 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 compressed index provided significant improvement over the simple SAS index on SYMBOL, we created and tested compressed 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 compressed index data set named IX_SYMDAT, shown in Figure 4.

Figure 4. Compressed 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. Compressed 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 compressed index of IX2_SYMDAT, and IX2_SYMDAT is a compressed 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 compressed 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
Compressed 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
Compressed 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
Compressed 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 compressed 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. SAS "knows" that one million consecutive 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.

Of course, in **our** case, there are a couple of problems: (1) **we** typically have multiple ranges; 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 data set listed in the SET statement. Instead, when the first data set is complete 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 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 Exemple 7a:

Example 7a: Using Compressed 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 compressed 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 compressed index

And it works. Table 3 shows further gains by moving from direct access to sequential access, in both cases using the compressed 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 Number of Symbols			Entire Month 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%
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 generating the corresponding ranges would result in a SET statement far longer than its maximum permissible 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 set 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. WEAKNESSES AND NOTES ON THE COMPRESSED INDEX

The fact that the compressed index is faster in these tests does not indicate any deficiencies 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 the 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.

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 compressed index will no longer provide any significant advantages over the SAS index.

However, there are some potential benefits to the compressed index that we have not explored in this paper, such as:

1. Given the small amount of space taken by the compressed index, it would be "cheap" (in terms of disk space) to add compressed 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 compressed index would support removal of the sort variable(s) from the data set, providing further disk space storage with no impact on performance.

3. Because the compressed index essentially contains a frequency table of the index variables, precise 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.

4. Unlike the SAS index, a variant of the compressed index could be formed that does not have an entry for every value of the index variable. Instead an entry could represent a range of values. For instance, in the case of our NYSE data sets, the observations are actually sorted by SYMBOL, DATE, and TIME (recorded to the nearest second). Making a compressed index of all the TIME values would yield an index file with too few records per SYMBOL/DATE/TIME combination to provide any performance benefits. But by making index entries that cover time ranges (e.g. containing the first and last RID of each hour), retrievals that have a time constraint could be supported. But unlike a compress index, this "condensed index" cannot recreate all the information in an ordinary SAS 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 compressed 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.

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
	216 Vance Hall
	3733 Spruce St
	Philadelphia, PA 19104-6301
Work Phone:	215.898.2160
Fax	215.573.6073
Email	mkeintz@wharton.upenn.edu