ABSTRACT

The advantage of an indexed Base SAS® engine’s dataset is that when the dataset is queried with a WHERE clause the process may be optimized and index-subsetting will work faster than a sequential read. Unfortunately the Base SAS engine is unable to use more than one index at a time to optimize a WHERE clause. Especially if a WHERE clause contains an OR condition between two different (indexed) variables the Base SAS engine won’t optimize it and will execute a sequential read.

The main idea of this paper is to offer a solution to the following question: How to handle a situation in which we have a WHERE clause with an OR condition between two different indexed variables and want to use the advantage of indexing to subset the data faster? The solution is datastep based and uses hash-tables. But foremost it is both simple to implement and efficient.

INTRODUCTION

When we are working with datasets in the SAS Base engine we often try to use some optimization techniques which allow us to improve performance. There are dozens of such techniques but some of the most basic approaches could be described with the following steps:

- narrow the data, i.e. select an observation only IF it is necessary for the process,
- even better, use a WHERE clause instead of a subsetting IF statement, as you will reduce the data before it is inserted into PDV,
- if a WHERE clause subsets only a small part of the dataset (and the dataset is big enough, i.e. spans across more than 3 pages) add an INDEX to the dataset,
- if your INDEX file is big and the dataset is quite "static" (i.e. doesn’t change to often) consider Mark Keintz’s compressed indexes (see [8]).

Unfortunately there’s a limit to index usage as SAS Base engine can’t use more than one index to optimize a WHERE clause. It is explained in the question 22: "Why can’t an index be used if there is an OR in the WHERE expression?" in Billy Clifford’s paper [6].

But are we 100% sure that we can’t do this, i.e. to use the advantage of INDEXes when a WHERE clause contains an OR condition? In fact we can. If we incorporate some small additional programming effort we are able to overcome the obstacle described above.

There are two basic steps behind the process. The first is to realize that a WHERE clause with an OR condition can be split into separate clauses which can be executed independently (and use different INDEXes). The second is to realize that if we can efficiently manage information on which observations have already been read we won’t have a problem with consolidating the data and will avoid potential duplicates.

For the sake of clarity, from now on, any further phrases such as "SAS is doing something" will be related to the Base SAS engine unless explicitly noted.

TOOLS

Before we start the main topic let’s take a quick look at two basic concepts we are going to work with, i.e. indexes and hash-tables.

**Indexes.** The concept of SAS index, from a user point of view, is very simple and intuitive. We can think of an index as a list of "key-value" pairs. "Keys" are values of the variable on which the index is built. "Values" are lists of row identifiers, a.k.a. RIDs, which are the pointers to location of observations containing a given value in a dataset. The most natural analogy would be a book and... its index. For example, if our dataset looks as follows:
A dataset with two variables

<table>
<thead>
<tr>
<th>Obs.</th>
<th>VarA</th>
<th>VarB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>10</td>
</tr>
</tbody>
</table>

we can think of an index constructed for variable VarA as:

An index for variable VarA

<table>
<thead>
<tr>
<th>Key: {RIDs}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: {1, 4, 7}</td>
</tr>
<tr>
<td>B: {2, 5}</td>
</tr>
<tr>
<td>C: {3, 6}</td>
</tr>
</tbody>
</table>

and for variable VarB as:

An index for variable VarB

<table>
<thead>
<tr>
<th>Key: {RIDs}</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: {1, 3, 5, 7}</td>
</tr>
<tr>
<td>20: {2, 4, 6}</td>
</tr>
</tbody>
</table>

We have already mentioned that indexes are used to optimize data selection in WHERE clauses. When we write the code:

WHERE VarA = "B";

SAS estimates the number of observations read by the clause from the dataset, and if it is worth it, SAS uses the index. What does it mean "uses the index"? Well, instead of a sequential read through the dataset, SAS will look at a list of record identifiers for that given value "B" of variable VarA and will read only the observations pointed by RIDs.

Just as a reminder, this is only "a user point of view". Of course, under the hood it is more complex than the description above. Starting for example with the fact that indexes are stored in a separate file, and they are tree-shaped data structures, and the "estimation of the number of observations to be read" is a complicated process, and eventually optimization of WHERE clauses is not the only purpose of indexes existence. But the intuition we already have is good enough for the beginning. Very good references to discover indexes in details are: Billy Clifford’s paper [6] and Michael A. Raithel’s book [7], and of course SAS on-line documentation.

Hash-tables. The concept of a hash-table is very user friendly as long as we start with good intuition. Users which are not familiar with object oriented programming notation may, at the first glance, consider hash-table’s syntax a bit awkward, but do not judge a book by its cover!

From a user perspective a hash-table can be considered as a younger and smarter sibling of a classical, well known, SAS temporary array. Let’s take a quick look at arrays and declare a temporary array ARR. We can do it for example by calling the following code:

```
array ARR[6] $ temporary;
```

We can visualise ARR as a pre-allocated, in-memory, and fixed-size set of adjacent cells, with integer pointer addressing each cell, waiting for the data. Like in the figure below:

To populate ARR with the data we can run the following code:

```
k = 2; v = "B"; ARR[k] = v; ①
k = 1; v = "A"; ARR[k] = v; ②
k = 4; v = "D"; ARR[k] = v; ③
```

and after each line of code the array can be visualised as presented in the figure below:

To retrieve a cell’s value we are using a corresponding key’s value in array’s reference as for example:

```
k = 2; v = ARR[k]; put v=;
```

and as a result in the SAS log we will see: v=B printed out.

In case of a hash-table (again, from a user point of view) the process, modulo the syntax, looks similar. Let’s declare
a hash-table `HSH`. We can do it for example by calling the following code:

```plaintext
length k 8 v $ 1;
declare hash HSH(ordered:"ascending",
    hashexp:8);
HSH.DefineKey("k");
HSH.DefineData("v");
HSH.DefineDone();
```

Before we continue a note about syntax’s analogies. In the `declare hash` statement we are giving a hash-table a name (it is an array statement’s analogy). With the `ordered` option we are forcing keys to be in the ascending order (in the array keys are ordered by default since they are ascending integers). The `hashexp` option establishes maximum size of a hash-table (it could be very very loosely compared to array’s size declaration). The `.DefineKey()` method indicates variables which are used as a key (an analogy of `k` in the `v = ARR[k]` code) and the `.DefineData()` method indicates variables which are used to hold the data portion (an analogy of `v` in the `v = ARR[k]`). And the `.DefineDone()` method is the equivalent of a semicolon at the end of array’s definition.

We can visualise `HSH` as a not-pre-allocated, in-memory, and not-fixed-size set of “key-data” pairs, with (not necessary integer) key-pointers addressing each data portion. Hence after the declaration `HSH` looks like in the figure below and it is awaiting to be populated with the data.

```
<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The process of inserting data into `HSH` uses the `.add()` method and to populate `HSH` with the data we can run the following code:

```plaintext
k = 2; v = "B"; HSH.add();    // 1
k = 1; v = "A"; HSH.add();    // 2
k = 4; v = "D"; HSH.add();    // 3
```

and after each line of code the hash-table can be visualised as presented in the figure below (notice how the order of keys is changing due to the `ordered:"ascending"` tag):

```
<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>&quot;B&quot;</td>
</tr>
<tr>
<td>[1]</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>[2]</td>
<td>&quot;B&quot;</td>
</tr>
</tbody>
</table>
```

When a hash-table has been populated retrieving data is as simple as `2+2`. All we need to do is to set the key’s value and call `HSH`’s `.find()` method, as in the following example:

```plaintext
k = 2; HSH.find(); put v=;
```

and in the SAS log we will see: `v=B` printed out. We don’t even have to use any assignment statement since the hash-table will handle it itself. If the `.find()` method will be successful then variable `v` will be populated with data automatically.

As a side note, the above process of adding the data to the hash-table `HSH` can also be executed with a do loop in a similar fashion as with arrays. Code for arrays would look as follows:

```plaintext
k = 0;
do v = "A", "B", "C", "D", "E", "F";
k + 1;
    ARR[k] = v;
end;
```

while for hash-tables it would be:

```plaintext
k = 0;
do v = "A", "B", "C", "D", "E", "F";
k + 1;
    HSH.add();
end;
```

A hash-table doesn’t have to use integers as keys and, what is even more comfortable, we can have a more complex data portion than a single cell. For example we can declare a hash-table `HT` and populate it with data in the following way:

**Code:**

```plaintext
length k1 8 k2 $ 1 v1 $ 1 v2 8;
declare hash HT(ordered:"ascending");
HT.DefineKey("k1", "k2");
HT.DefineData("v1", "v2", "k1");
HT.DefineDone();

k1 = 1; k2 = "m"; v1 = "A"; v2 = 13; HT.add();
k1 = 2; k2 = "i"; v1 = "B"; v2 = 17; HT.add();
k1 = 3; k2 = "n"; v1 = "C"; v2 = 42; HT.add();
k1 = 4; k2 = "i"; v1 = "D"; v2 = 66; HT.add();
```

---

1 A method in object oriented programming terminology may be considered as a function associated with an object.
k1 = 5; k2 = "p"; v1 = "E"; v2 = 78; HT.add();
k1 = 6; k2 = "w"; v1 = "F"; v2 = 82; HT.add();

Visualisation:

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1, k2</td>
<td>v1, v2, k1</td>
</tr>
<tr>
<td>[1, &quot;m&quot;]</td>
<td>&quot;A&quot;, 13, 1</td>
</tr>
<tr>
<td>[2, &quot;i&quot;]</td>
<td>&quot;B&quot;, 17, 2</td>
</tr>
<tr>
<td>[3, &quot;n&quot;]</td>
<td>&quot;C&quot;, 42, 3</td>
</tr>
<tr>
<td>[4, &quot;i&quot;]</td>
<td>&quot;D&quot;, 66, 4</td>
</tr>
<tr>
<td>[5, &quot;p&quot;]</td>
<td>&quot;E&quot;, 78, 5</td>
</tr>
<tr>
<td>[6, &quot;w&quot;]</td>
<td>&quot;F&quot;, 82, 6</td>
</tr>
</tbody>
</table>

What’s even more useful is that hash-tables allow to load data straight from an external dataset within declare hash statement. For example, assuming that dataset work.SomeDataset contains tree numeric variables k, d1, and d2 we could load it into a hash-table with following code:

```sas
options FULLSTIMER 1,
      MSGLEVEL = I 3;

length k d1 d2 8;
declare hash HfrmDS(dataset:"work.SomeDataset");
HfrmDS.DefineKey("k");
H frmDS. DefineData("d1", "d2");
H frmDS. DefineDone();
```

So, the punch line here is that a hash-table is a flexible and dynamically allocated data structure, with a very efficient data access time and a "dictionary" kind of behaviour.

Again, as a reminder, this is only "a user point of view". As in the case of indexes, also in the case of hash-tables there is much more happening under the hood than in the description above. Starting for example with the fact that there is an internal hashing function involved, and data are kept in tree-shaped data structures (AVL-trees). But also in this case the intuition we already have is good enough. References to discover hash-tables in more details are: Paul Dorfman’s paper [5], Paul Dorfman’s and Don Henderson’s book [4], Chris Schacherer’s paper [3], Paul Dorfman’s and Koen Vyverman’s paper [2], Art Carpenter’s book [1], and of course SAS on-line documentation.

**THE PROCESS**

Now, when we covered all prerequisites, we can begin the process of modifying our code in a way that will allow us to handle a `WHERE` clause with an `OR` condition and at the same time use the advantage of `INDEXes`.

**The Dataset.** At the very beginning let’s turn-on some additional options which will give us extended logging features.

```
libname mysets BASE "...";
data mysets.countries;	infile cards dlm = '0A'x;
  input country $ :50. ;
call streaminit(2222);
  sort = rand("uniform");
cards;
  Afghanistan [AFG]
  Aland Islands [ALA]
  Albania [ALB]
  ... 
  Virgin Islands, US [VIR]
  Wallis and Futuna Islands [WLF]
  Western Sahara [ESH]
  Yemen [YEM]
  Zambia [ZMB]
  Zimbabwe [ZWE]
  ;
run;
```

As the next step we are going to create a bigger dataset which we will use in the process.

```
data mysets.INDEXX_OR(
  INDEX = (country date);
  out = mysets.countries(drop = sort);
  by sort;
runcs sort
```

In [1] we specify whether to write extended system performance statistics to the SAS log (e.g. memory usage, real time, cpu time, etc.) In [2] we specify the level of details in messages that are written to the SAS log and value "I" indicates to print additional notes pertaining to index usage, merge processing, sort utilities, etc.

Using Wikipedia as a data base we are going to prepare a randomly ordered (3) dataset of countries names (248 observations) which will be used as a base for data preparation in the main code.
... set mysets.countries;

format date yymmdds10.;
do date = '1jan1960'd to '28apr2019'd;
y = year(date);
m = month(date);
d = day(date);
call streaminit(123);
measurement = 456+round(rand("Normal")*78);
output;
if rand("Uniform") > 0.9 then output;
end;
run /*cancel*/;

The INDEXX_OR dataset has two simple indexes: country (-country) and date (date), created in lines 271 to 274. The dataset is built in the following way: for each country (repeated a dozen times) a bunch of records, with dates and random measurements, the sets seed for rand() function and with about 10% of "natural" duplicates, is generated. Two following procedures will give us the dataset’s shape and metadata.

The output shows:

<table>
<thead>
<tr>
<th>Variables</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexes</td>
<td>2</td>
</tr>
<tr>
<td>Observation Length</td>
<td>96</td>
</tr>
<tr>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Compressed</td>
<td>NO</td>
</tr>
<tr>
<td>Sorted</td>
<td>NO</td>
</tr>
</tbody>
</table>

The CONTENTS Procedure

Data Set Name MYSETS.INDEXX_OR
Member Type DATA
Engine V9
Created 04/28/2019 09:00:00
Last Modified 04/28/2019 09:00:00
Data Representation WINDOWS_64
Encoding utf-8
Observations 70935765

The PRINT Procedure

Indexes usage test. We are going to summarize measurements in variable SoM (country) and count them in variable i (date) for observations selected with a WHERE clause. We will do it for both SQL and datastep processing. To prove that proc SQL really uses an index to work the WHERE clause out we will run the following code:
proc sql;
select
  sum(measurement) as SoM format best32.
, count(1) as i
from
  mysets.INDEXX_OR
where
  country = 'Poland [POL]';
quit;

When we look into the SAS log, thanks to the MSGLEVEL = I option, we can see the following information:
INFO: Index country selected for WHERE clause optimization

Just for completeness, if we use a WHERE clause to subset data in the datastep:

data _NULL_;
set mysets.INDEXX_OR END = eof;
where
  date between '01may2015'd and '30may2015'd
  OR
  country = 'Poland [POL]';
  SoM + measurement;
i + 1;
if eof then
  do;
    put SoM= best32. i=;
  end;
run;

we will receive an equivalent INFO notification relating to index date.

In the next test we will see that when the WHERE clause contains the OR condition on two different variables SAS won't use any index to optimize subsetting, regardless we use the WHERE clause in proc SQL (1) or in a datastep (2).

proc sql;
select
  sum(measurement) as SoM format best32.
, count(1) as i
from
  mysets.INDEXX_OR
where
  date between '01may2015'd and '30may2015'd
  OR
  country = 'Poland [POL]';
quit;

data _NULL_;  
set mysets.INDEXX_OR END = eof;
where
  date between '01may2015'd and '30may2015'd
  OR
  country = 'Poland [POL]';
  SoM + measurement;
i + 1;
if eof then
  do;
    put SoM= best32. i=;
  end;
run;

After running the above code the SAS log contains the following notes for proc SQL:
NOTE: PROCEDURE SQL used (Total process time):
  real time 1:32.40
  user cpu time 8.79 seconds
  system cpu time 5.45 seconds
  memory 5478.53k
  OS Memory 20724.00k

and for the datastep:
SoM=174856439 i=383492
NOTE: There were 383492 observations read from the data set MYSETS.INDEXX_OR.
WHERE (date>='01MAY2015'D and date<='30MAY2015'D)
  or
  (country='Poland [POL]');
NOTE: DATA statement used (Total process time):
  real time 48.16 seconds
  user cpu time 5.39 seconds
  system cpu time 4.29 seconds
  memory 724.56k
  OS Memory 15856.00k

There is no information about index usage during code execution. In both cases a sequential read took place.

General Overview. Ok, so how to solve the "OR issue"? The solution is datastep based and uses hash-tables, but we will go through it step-by-step starting with arrays approach and than jumping into "hashes".

Before we dive into details, first and the most important thing is to realise that we can split the WHERE clause around the OR condition into two separate clauses. After that we can execute both WHERE clauses independently - what ensures
that indexes will be used since we are having only simple conditions. Eventually, as the final step, we have to somehow bring the results (i.e. subsetted data) together. But we have to do it in such a way that the combined dataset will not contain duplicated observations coming from both WHERE clauses. The idea to prevent duplicates is to keep the record of already read observations.

Example. Let’s consider the following WHERE clause
WHERE VarL="B" OR VarN=20;
and the following dataset:

<table>
<thead>
<tr>
<th>Obs.</th>
<th>VarL</th>
<th>VarN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
<td>40</td>
</tr>
</tbody>
</table>

with both variables indexed. If we split the WHERE clause into two independent clauses: WHERE VarL="B" and WHERE VarN=20, as in the figure above, and execute them under two separate set statements in one datastep we will get duplicated records in the produced dataset. In the figure below they are marked with \( \times \) symbol. The “Current Obs.” column keeps track of the observation’s number that was read-in.

Dataset with observations read by WHERE VarL="B" and WHERE VarN=20

<table>
<thead>
<tr>
<th>Current Obs.</th>
<th>VarL</th>
<th>VarN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>20</td>
</tr>
</tbody>
</table>

As we wrote the idea is to keep a record of observations read during execution of the first part of the WHERE clause. Hence, in the example we are considering, after executing WHERE VarL="B" clause we have the following observations in the output dataset:

Observations read with WHERE VarL="B"

\[
\begin{array}{|c|c|c|}
\hline
\text{Current Obs.} & \text{VarL} & \text{VarN} \\
\hline
4 & B & 20 \checkmark \\
5 & B & 20 \checkmark \\
6 & B & 30 \checkmark \\
3 & A & 20 \checkmark \\
4 & B & 20 \checkmark \\
5 & B & 20 \checkmark \\
\hline
\end{array}
\]

and a list of observations numbers read from the input dataset:

A list of read observations

<table>
<thead>
<tr>
<th>Observation number</th>
<th>“4”</th>
<th>“5”</th>
<th>“6”</th>
</tr>
</thead>
</table>

When we execute the second clause each time before sending an observation to the output dataset we are checking if its number is on the list. In the example we are using the WHERE VarN=20 clause which fetches observations 3, 4, and 5. Since observations number 4 and 5 were already on the list hence they are not outputted and only the observation number 3 is. The final dataset contains only four observations marked with tick-mark \( \checkmark \) for observations from the first WHERE clause part and with \( \times \) for observations from the second and duplicated observations omitted (marked with \( \times \)).

Observations added by WHERE VarN=20

\[
\begin{array}{|c|c|c|}
\hline
\text{Current Obs.} & \text{VarL} & \text{VarN} \\
\hline
4 & B & 20 \checkmark \\
5 & B & 20 \checkmark \\
6 & B & 30 \checkmark \\
3 & A & 20 \checkmark \\
4 & B & 20 \times \\
5 & B & 20 \times \\
\hline
\end{array}
\]

Now when we have a general overview of the process and we tested it with an example we can jump right into the code.

Programming. The first attempt considers using a temporary ARRAY. To be able to do that we have to do some "pre-processing". We have to get the number of observations to set the ARRAY’s size and in consequence allocate memory.

```plaintext
data _null_; 366
if 0 then set mysets.INDEXX_OR nobs = nobs; 367
call symputx("_NOBS_", nobs, "G"); 368
stop;
run; 369
```

Having the metadata (i.e. the number of observations, _NOBS_ macrovariable) collected we are:
1. declaring a temporary ARRAY to be a list to mark visited observations,
2. executing a DoW-loop in which we are reading-in data for the first part of our WHERE clause (lines 374 to 384),
3. using the CUROBS option to create a variable that contains the observation number that was just read from the dataset,
4. marking the ARRAY’s cell which key equals to the current value of curobs variable,
5. starting to aggregate the data (lines 382 and 383),
6. executing a second DoW-loop in which we are reading-in data for the second part of the WHERE clause (lines 387 to 399),
7. verifying if a visited observation was already read and if that’s the case, going to the next iteration and skipping the aggregation,
8. and eventually if the visited observation wasn’t already reading in the temporary array (line 395, it allows to add third, fourth and further WHERE conditions in a very simple way just by copying the second DoW-loop and changing the condition) and updating aggregated data (lines 396 and 397).

```
data _NULL_;  
ARRAY _obs_[&_NOBS_] _temporary_;  
do until(eof);  
set  
   mysets.INDEXX_OR END=eof CUROBS=curobs  
;  
   where date between '01MAY2015'D and '30MAY2015'D;  
   _obs_[curobs] = 1;  
   SoM + measurement;  
i + 1;  
end;  
eof = 0;  
do until(eof);  
set  
   myssets.INDEXX_OR END=eof CUROBS=curobs  
;  
   where country = 'Poland [POL]';  
   if _obs_[curobs] NE 1 then  
      do;  
         _obs_[curobs] = 1;  
      end;  
   end;  
put SoM= best32. i=;  
stop;  
run;  
```

The result is the same as in the case of the previous, index-less, ones but now the SAS log shows totally different notes and infos.

```
INFO: Index date selected for WHERE clause optimization.  
INFO: Index country selected for WHERE clause optimization.  
SoM=174856439 i=383492  
NOTE: There were 98206 observations read from the data set MYSETS.INDEXX.OR.  
WHERE (date>='01MAY2015'D and date<='30MAY2015'D);  
NOTE: There were 285681 observations read from the data set MYSETS.INDEXX.OR.  
WHERE country='Poland [POL]';  
NOTE: DATA statement used (Total process time):  
   real time 4.78 seconds  
   user cpu time 0.68 seconds  
   system cpu time 1.78 seconds  
   memory 555102.21k  
   OS Memory 570044.00k  
```

We can see that indexes were used, which decreased the execution time. Unfortunately the ARRAY approach has one drawback. The consequence of using a temporary array is that we have to preallocate the memory to handle markers for all observations even though only a small part of ARRAY’s cells will be used, which is inefficient.

A solution for ARRAY’s memory issue would be a data structure which can dynamically modify its size. And in such a case a hash-table appears to be the perfect candidate. A hash-table allows us to add elements without previous memory allocation and in terms of searching works very efficiently (not as fast as array’s direct access but fast enough).

To use a hash-table our previous code needs only slight changes:

```
array _obs_[&_NOBS_] _temporary_;  
do until(eof);  
set  
   myssets.INDEXX_OR END=eof CUROBS=curobs  
;  
   where date between '01MAY2015'D and '30MAY2015'D;  
   _obs_[curobs] = 1;  
   SoM + measurement;  
i + 1;  
end;  
eof = 0;  
do until(eof);  
set  
   myssets.INDEXX_OR END=eof CUROBS=curobs  
;  
   where country = 'Poland [POL]';  
   if _obs_[curobs] NE 1 then  
      do;  
         _obs_[curobs] = 1;  
      end;  
```

3See Paul Dorfman’s paper to learn more details about this wonderful programming technique.
NOTE: There were 98206 observations read from the data set MYSETS.INDEXX_OR.
WHERE (date='01MAY2015'D and date='30MAY2015'D);

NOTE: There were 285681 observations read from the data set MYSETS.INDEXX_OR.
WHERE country='Poland [POL]';

NOTE: DATA statement used (Total process time):
real time 1.99 seconds
user cpu time 0.68 seconds
system cpu time 1.31 seconds
memory 26409.81k
OS Memory 41148.00k

Again we can see that indexes were used, which significantly decreased the execution time. In case of the hashtable approach memory footprint is much smaller than in the array case (~26MB vs. ~555MB). To be clear, the memory footprint is still bigger than the one from index-less processing but now it looks like fair trade-off between time and RAM.

There is another approach which uses hash-tables, simplifies the code and makes it execute faster. Just to differentiate it from the previous approach let’s name it "hash approach 2". Changes in the code are:

1. two DoW-loops are replaced with a single set statement with input dataset used twice,
2. and 3 the WHERE clauses are moved into dataset options (lines 450 and 454),
3. the .find() method is replaced with the .check() method which doesn’t retrieve the data but only checks if the key’s value exists in the hash-table,
4. the goto statement is used to skip aggregation if we encounter an already read observation.

The result is the same as in the previous cases. The log shows following notes and infos.

INFO: Index date selected for WHERE clause optimization.
INFO: Index country selected for WHERE clause optimization.

SoM=174856439 i=383492
The code from the previous section was executed on a laptop with following characteristics:

- **Lenovo Y700**,  
  - Intel(R) Core(TM) i7-6700HQ CPU @2.60GHZ,  
  - 16GB RAM, SSD + HDD disk drive,  
  - Windows 10 Pro N,  
  - Base SAS 9.4M4 with memsize 8GB.

To compare execution times and efficiency the code was also executed on two different data setups and machines: one on a desktop and the other on a server.

**Desktop machine characteristics were:**

- **HP EliteDesk 800 G1 SFF,**  
  - Intel(R) Core(TM) i5-4590 CPU @3.30GHz,  
  - 8GB RAM, HDD disk drive,  
  - Windows 7 Enterprise - ServicePack 1,  
  - Base SAS 9.4M4 with memsize 6GB.

**Datasets were:**

- **Small:**  
  - Observations: 3,668,464  
  - File Size (bytes): 353,173,504 ~ 337MB

- **Medium:**  
  - Observations: 70,304,151  
  - File Size (bytes): 6,765,871,104 ~ 6GB

- **Big:**  
  - Observations: 378,833,440  
  - File Size (bytes): 36,457,152,512 ~ 34GB

**Common attributes:**

- Variables: 6  
- Indexes: 2  
- Observation Length: 96

**The results (in terms of time) were as follows:**

<table>
<thead>
<tr>
<th></th>
<th>sql no index</th>
<th>datastep no index</th>
<th>hash appr. 1</th>
<th>hash appr. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>real time</strong></td>
<td>0:00.32 (0:00.02)</td>
<td>0:00.27 (0:00.03)</td>
<td>0:00.30 (0:00.08)</td>
<td>0:00.30 (0:00.03)</td>
</tr>
<tr>
<td>user cpu</td>
<td>0:00.11 (0:00.01)</td>
<td>0:00.10 (0:00.03)</td>
<td>0:00.04 (0:00.02)</td>
<td>0:00.05 (0:00.04)</td>
</tr>
<tr>
<td>system cpu</td>
<td>0:00.16 (0:00.03)</td>
<td>0:00.17 (0:00.03)</td>
<td>0:00.20 (0:00.00)</td>
<td>0:00.18 (0:00.02)</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>26406.78k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Medium:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>real time</strong></td>
<td>1:07.23 (0:07.56)</td>
<td>1:27.32 (0:49.28)</td>
<td>0:23.76 (0:01.79)</td>
<td>0:02.55 (0:00.48)</td>
</tr>
<tr>
<td>user cpu</td>
<td>0:06.35 (0:00.89)</td>
<td>0:07.32 (0:00.56)</td>
<td>0:00.80 (0:00.07)</td>
<td>0:00.63 (0:00.04)</td>
</tr>
<tr>
<td>system cpu</td>
<td>0:07.77 (0:01.08)</td>
<td>0:08.84 (0:02.90)</td>
<td>0:02.53 (0:00.14)</td>
<td>0:01.73 (0:00.02)</td>
</tr>
<tr>
<td><strong>Big:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>real time</strong></td>
<td>8:22.38 (0:47.58)</td>
<td>10:09.25 (3:24.73)</td>
<td>1:15.04 (0:09.41)</td>
<td>0:07.33 (0:02.22)</td>
</tr>
<tr>
<td>user cpu</td>
<td>0:44.23 (0:03.56)</td>
<td>0:36.22 (0:04.46)</td>
<td>0:03.16 (0:00.23)</td>
<td>0:02.37 (0:00.01)</td>
</tr>
<tr>
<td>system cpu</td>
<td>0:42.12 (0:03.54)</td>
<td>1:17.15 (0:10.16)</td>
<td>0:05.71 (0:00.27)</td>
<td>0:03.88 (0:00.29)</td>
</tr>
</tbody>
</table>

An additional advantage of that last approach is that it allows us to extend the **WHERE** clause with multiple **OR**s in the easiest way by just adding a dataset name with a new **WHERE** part in the **SET** statement. Which brings the idea of wrapping it into a convenient macro (see the last section for a pointer to details).

**BENCHMARKING**
Server machine characteristics were:
ProLiant DL380 Gen9 HP,
Intel(R) Xeon(R) CPU E5-2667 v3 @3.20GHz,
256GB RAM,
Red Hat Linux,
EG session on SAS 9.4M3 with memsize 8GB.

Datasets were:

<table>
<thead>
<tr>
<th>Type</th>
<th>Observations</th>
<th>File Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>4'4019'606</td>
<td>6709182464 ~ 6GB</td>
</tr>
<tr>
<td>Medium</td>
<td>246'134'809</td>
<td>37513396224 ~ 35GB</td>
</tr>
<tr>
<td>Big</td>
<td>1'917'837'577</td>
<td>292296458240 ~ 272GB</td>
</tr>
</tbody>
</table>

Common attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Variables</th>
<th>Indexes</th>
<th>Observation Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>13</td>
<td>2</td>
<td>152</td>
</tr>
</tbody>
</table>

In both cases index usage in hash-table approaches improved performance time. But to be non-judgemental we have to admit that in the case of “small” sets differences in times weren’t as impressive as in the case of “big” ones.

**THE CODE**

If you are interested in testing approaches presented above yourself and want to play a bit with the code and data you can download SAS codes which were the motivation for this paper under the following "world wild web" address:
http://www.mini.pw.edu.pl/~bjablons/SASpublic/

you can find code with data: Countries.sas and a bunch of OR-condition-in-WHERE-clause-with-INDEX-[...].sas codes (the "[...]" extends the discussion).

**REFERENCES**


**ACKNOWLEDGMENTS**

Author would like to acknowledge Filip Kulon, Krzysztof Socki and Allan Bowe for their contribution and effort to make this paper looks and feel as it should. Thanks Filip! Thanks Krzysztof! Thanks Allan!

Author would like to acknowledge the Citibank Europe PLC Poland for the support in the performance benchmarking execution.

**CONTACT INFORMATION**

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail address:
yabwon@gmail.com
bartosz1.jablonski@citi.com
or via the following LinkedIn profile:
www.linkedin.com/in/yabwon
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.