Paper SAS1680-2018

# Supercharging Data Subsets in SAS® Viya® 3.3

Brian Bowman, SAS Institute Inc., Cary, NC

## ABSTRACT

SAS® Viya® 3.3 provides a powerful new table indexing capability that has the potential to significantly improve performance for data handling and analytics actions submitted to SAS® Cloud Analytic Services (CAS). Effective use of table indexes can simplify big data scenarios, improve usability, and reduce computing resource requirements. Programming examples using the CAS procedure are presented using SAS Viya capabilities integrated into the latest release of SAS®9. This information is beneficial to SAS Programmers, data scientists, data engineers, and others involved in configuring and deploying big data applications within SAS®9 and SAS Viya.

The following examples are included:

- understanding CAS table indexes
- determining which table variables to index
- indexing while loading data into CAS
- indexing variables in CAS action output tables
- using the new CAS Index action
- data subset operations using indexes
- table size and computing resource considerations
- performance comparisons

## INTRODUCTION

SAS® Cloud Analytic Services (CAS) is the high-performance, massively parallel, multi-user compute engine powering the SAS® Viya® architecture. The CAS data model uses scalable, distributed in-memory tables supporting a rich set of analytics and data management capabilities. CAS table indexes integrate seamlessly within CAS tables and can significantly improve data subsetting performance and scalability, while reducing CPU consumption in a variety of computing scenarios.

Indexes are common in traditional databases as well as advanced data management platforms such as Base SAS®. Using various data structures and algorithmic techniques to optimize computational space-time tradeoffs, CAS indexes can significantly reduce the CPU and elapsed time required to subset data. Effective indexing requires understanding your data and subsetting use cases to determine which table columns to index.

The following major topics are discussed:

- CAS data distribution
- CAS table index design
- determining columns to index
- creating CAS tables with indexes
- measuring indexing performance

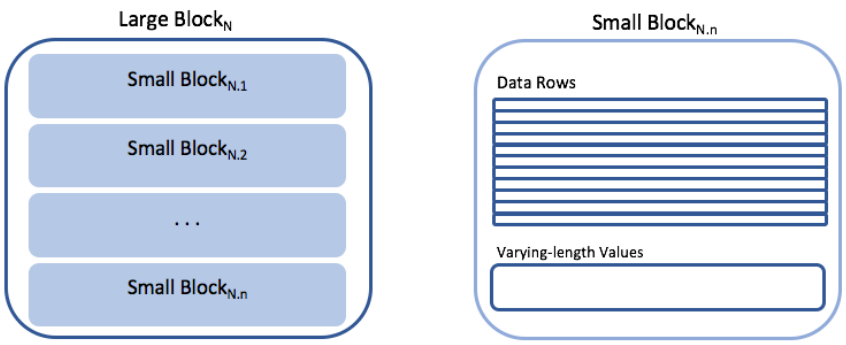
Using Cloud Analytic Services language (CASL) programming examples, this paper explores how your applications can benefit from CAS table indexes. With minor modification, these examples will work with any programming client language supported by SAS Viya, including Python, Java, R, Lua, and REST API.

## CAS DATA DISTRIBUTION

CAS data distribution fundamentals are prerequisite to understanding CAS table indexing. CAS can run in a Symmetric Multiprocessing (SMP) configuration on one compute node or in a Massively Parallel Processing (MPP) configuration on multiple nodes. In both configurations, CAS processes distributed data on concurrent threads in parallel, taking advantage of modern multi-core CPUs.

A CAS table consists of physical data segments called Large Blocks. Each of these segments is subdivided into a set of Small Blocks. Each Small Block holds a contiguous set of fixed-length rows, and each row holds the values for columns defined in the table. The varying-length data values for any VARCHAR or VARBINARY column types occupy a separate area following the rows. When varying-length column types are defined in the table, each row contains corresponding references to its values in the varying-length data area.

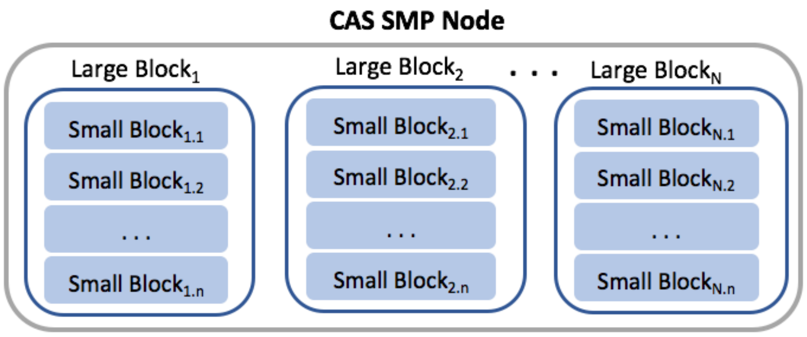
**Figure 1. Large and Small Blocks**



### DISTRIBUTED DATA EXAMPLES

In SMP configurations, all Large Blocks for a table are located on one compute node.

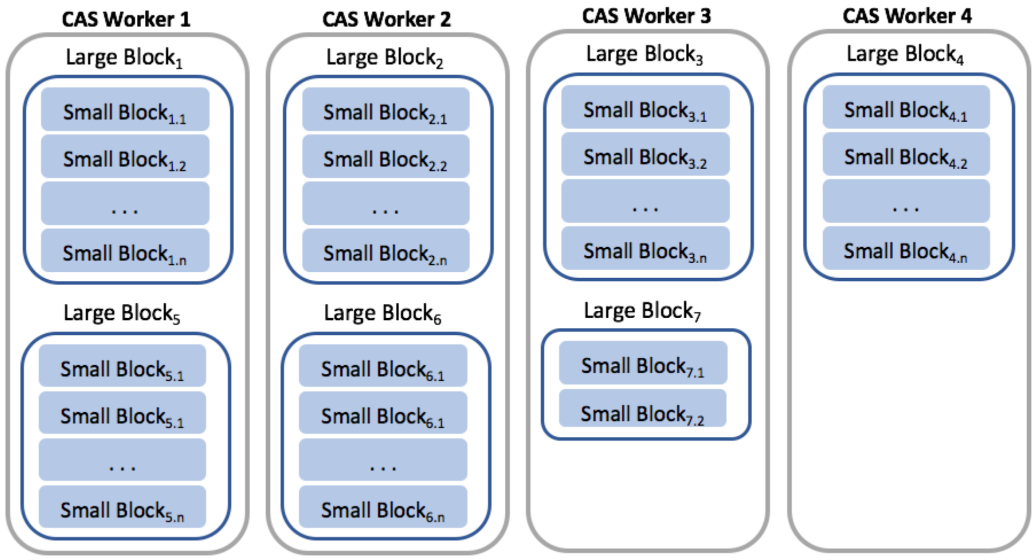
**Figure 2. SMP Data Distribution**



MPP Large Blocks are distributed across available CAS Workers. These configurations can scale to hundreds or thousands of CAS Workers, with each holding many tables, including huge tables composed of thousands (or more) of Large Blocks. CAS uses a *mapped-memory* model to manage Large Blocks. For more information, see "How SAS Cloud Analytic Services Uses Memory" in *SAS Cloud Analytic Services 3.3: Fundamentals*.

Figure 3 is an example CAS table with Large Blocks in a *round-robin* distribution on 4 Workers.

**Figure 3. MPP Data Distribution**



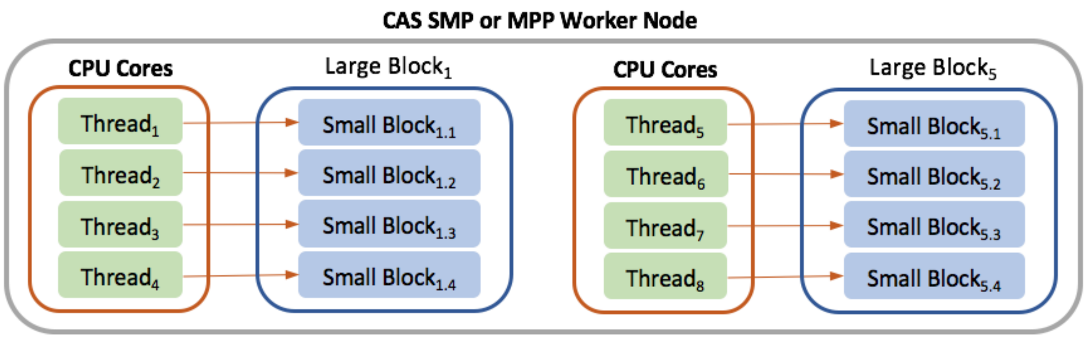
**NOTE:** Figures 2 and 3 depict in-memory table distributions independent of the data source (caslib type) or CAS action that the table originated from.

## THREAD ASSIGNMENT

Figure 4 depicts a simple case with threads assigned to Small Blocks on a single CAS Worker. Multiple Small Blocks might be queued to process on each thread when the total number of Small Blocks exceeds the available thread quantity. Similar queuing occurs when the number of assigned threads is greater than the quantity of available CPU cores on the machine. These same principles apply on SMP and across multiple Workers in a CAS MPP configuration.

A thread is assigned to each Small Block to enable concurrent processing of its data rows. The Linux OS and the machine it runs on cooperate to schedule available CPU cores that execute threaded code assigned to process Small Blocks. Parallel execution occurs as multiple threads are each given a time slice on their respective independent CPU cores.

**Figure 4. Small Block Thread Assignment**



With this model, subsetting elapsed time performance might appear to be acceptable without indexes, because many concurrent threads execute in parallel on highly distributed data. However, this method must scan each table row and compare values for the columns defined in the subsetting

WHERE expression. This necessity results in substantial CPU time consumption for tables having a large quantity of rows. Formal performance evaluation done by SAS R&D measured a significant and growing decline in overall CAS server performance as an increasing number of concurrent CAS sessions perform subsetting against large tables *without* the use of indexes. The following sections explore how indexing helps to improve CAS server performance and reduce CPU time when subsetting large tables is common practice.

## CAS TABLE INDEX DESIGN

### SUBSETTING PERFORMANCE

Subsetting CAS table data without indexes is constrained by linear time because argument values specified in a WHERE expression must be compared against *every* corresponding column value in the table. As already discussed, elapsed time can be reduced by sub-dividing the work among many concurrent threads executing in parallel CPU cores, at the cost of considerably more CPU time for tables with a large quantity of rows.

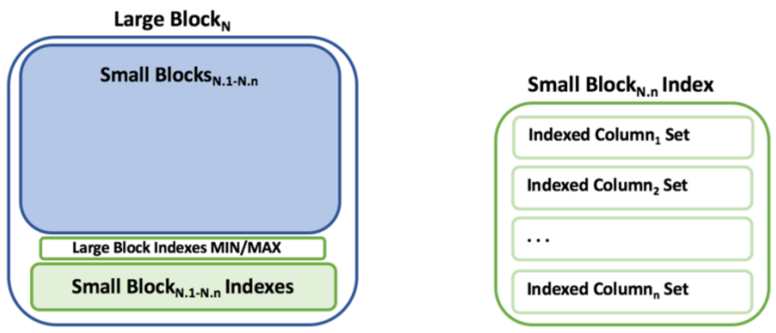
CAS table indexing reduces CPU and elapsed time for WHERE subsetting operations with algorithms that search the index data structures in *sublinear-to-logarithmic time*. Row data in Small Blocks is collected and ordered for each indexed column. References to all rows containing duplicates matching each unique value are stored compactly as part of an Indexed Column Set. Instead of performing a linear scan of all table rows, optimized algorithms traverse the index structure to determine the subset of data rows that match WHERE expression arguments. This space-time tradeoff requires additional RAM and storage for index structures but is mitigated by reduced CPU and elapsed time consumption during subsetting WHERE operations. As shown later, the space required for CAS table index data structures is usually nominal.

### INDEX STRUCTURE

A CAS table might be indexed on one to as many as all its columns. However, it is important to choose indexed columns carefully, and cases that benefit from indexing *all* table columns are probably rare. In current SAS releases, the WHERE processor limits CAS table indexes to column types of DOUBLE, CHAR, and VARCHAR. CAS table indexing will work with the expanded set of CAS data types when WHERE supports them in a future release.

CAS distributed table data requires an indexing design that differs significantly from the B+Tree index structures found in Base SAS as well as many database systems. As Figure 5 illustrates, CAS indexes are scoped to their corresponding Small Block and integrated within the containing Large Block. For example, Small Block$_{N.1}$ has a corresponding Small Block$_{N.1}$ Index containing column-related values in the set of optimized data structures for each table column that has an index.

**Figure 5. Large and Small Block Indexes**


Large Block$_N$
Small Blocks$_{N.1\text{-}N.n}$
Large Block Indexes MIN/MAX
Small Block$_{N.1\text{-}N.n}$ Indexes
Small Block$_{N.n}$ Index
Indexed Column$_1$ Set
Indexed Column$_2$ Set
...
Indexed Column$_n$ Set

Note that Indexed Column$_1$ Set refers to the first indexed column in the CAS table. This is not necessarily the first column in the table, and the same is true for Indexed Column$_{2–n}$ Sets.

### Large Block MIN/MAX

Indexed columns have MIN and MAX values scoped to each Large Block. Index-based subsetting uses Large Block MIN/MAX to determine whether argument values in the WHERE expression fall within the range of corresponding indexed column values in the Large Block. This quickly disqualifies Large Blocks that cannot possibly satisfy the WHERE expression from further subsetting at the Small Block level. This can substantially reduce CPU and elapsed time for WHERE subsetting when the table contains many Large Blocks where few meet the subsetting criteria.

### Saving Indexed CAS Tables

When an indexed CAS table is saved to a SASHDAT file, the indexes are not written to a separate file. As depicted in Figure 5, indexes are already part of each Large Block and are written with it to underlying storage. This method differs significantly from that used in the Base SAS file format, where the SAS7BDAT file when indexed has a separate SAS7BNDX file to store the indexes.

## THREAD ASSIGNMENT WITH INDEXING

Principles explained in the earlier THREAD ASSIGNMENT section apply to the following discussion, with a noticeable difference: In SAS Viya 3.3 subsetting with CAS table indexes, threads are assigned at the Large Block level. Figure 6 provides a high-level depiction of how index-based threads can disqualify Large Blocks by using their scoped MIN/MAX values. It also show how those that qualify use indexes to select Small Block rows containing data values that match a WHERE expression. The *red arrows* proceeding from the Thread$_{1–4}$ boxes illustrate this. In this example, Large Blocks 1 and 4 were disqualified, and a subset of rows from Small Blocks 2.1, 2.3, and 3.2 were selected using indexes.

**Figure 6. Threaded Subsetting with Indexing**

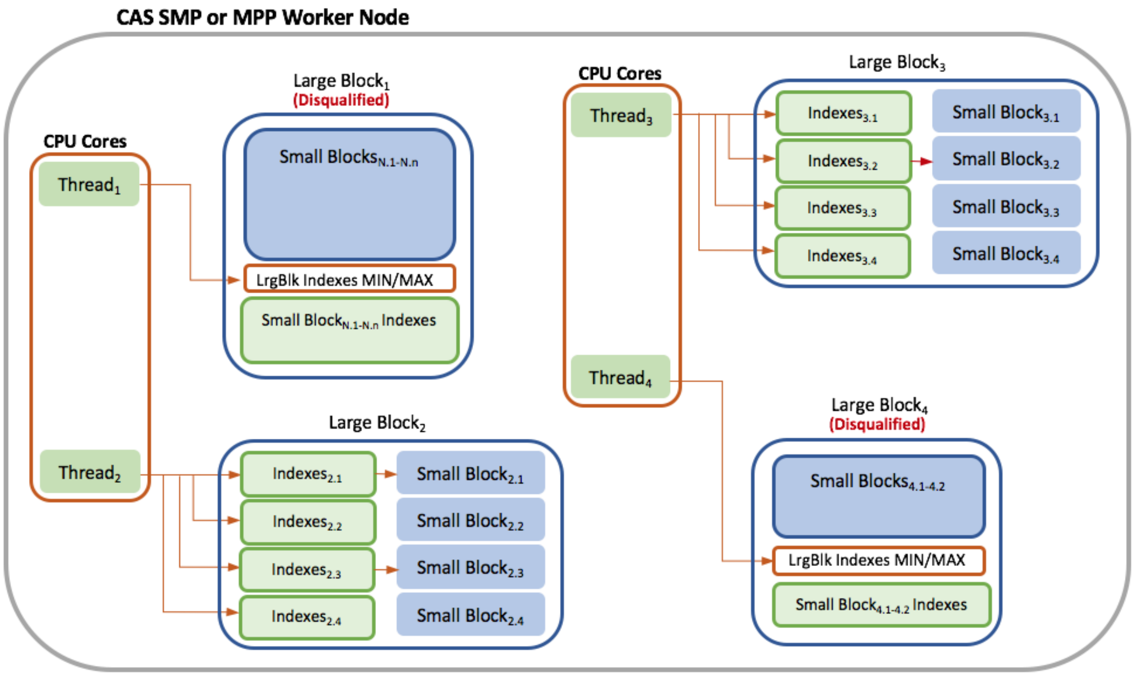
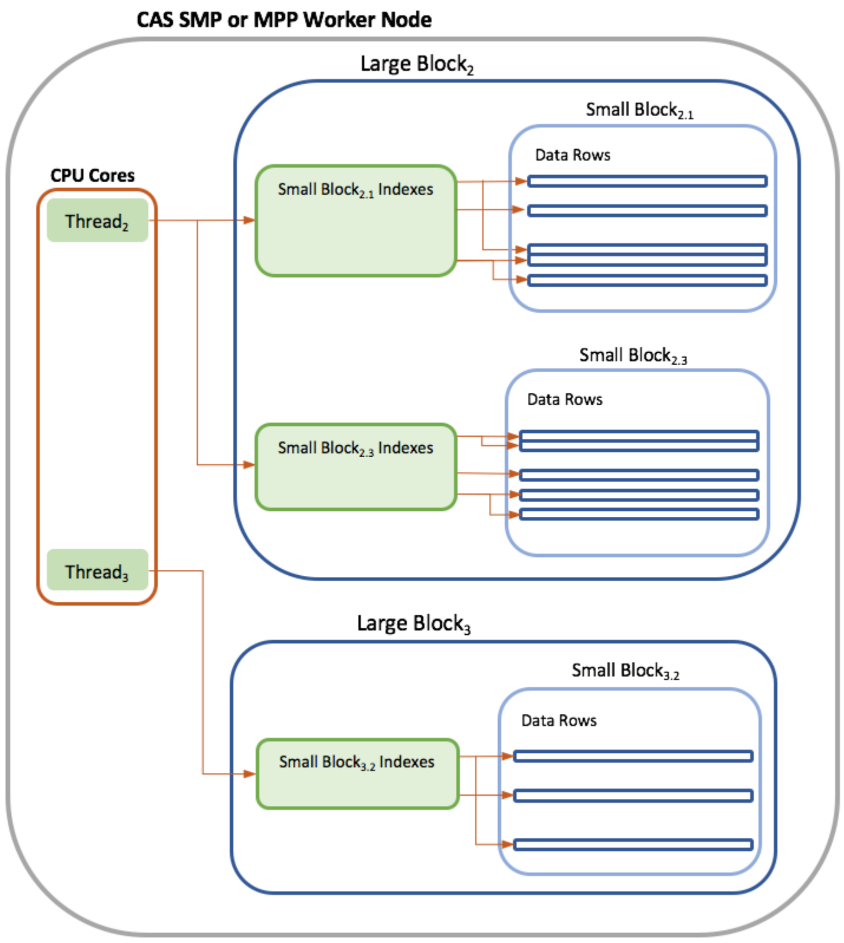


**Figure 7. Indexed Subset Results**



*To simplify Figure 7, varying-length data is not shown*

Figures 6 and 7 illustrate how CAS table indexing quickly reduces the data to only rows that are qualified by the WHERE expression. The threaded subsetting operations of Figure 6 and the resulting row subset of Figure 7 involve comparing MIN/MAX index values in Large Block scope or scanning Small Block indexes. A CAS SMP node is shown to simplify depicting the data distribution and subsetting row qualification among multiple Large Blocks. The same subsetting operations occur when a distributed table has multiple Large Blocks distributed across Worker nodes in a CAS MPP configuration. The subsetting logic depicted in Figure 6 and its resulting row subset shown in Figure 7 could be produced by many different WHERE expressions operating on CAS tables of different column types and value cardinality.

The following WHERE expression suffices to guide the following step-by-step explanation of index-based subsetting logic for Figures 6 and 7:

```
WHERE campaign = ‘Pace’ AND date IN(‘11Mar2015’d,‘21Apr2015’d)
```

## Indexing-based Logic Illuminated Using the Example WHERE Expression

1. Assume the example CAS table shown in Figures 6 and 7 has many columns, including two referenced in the WHERE expression—**`campaign`** AND **`date`—**which are both indexed.
2. Figure 6 demonstrates that Large Blocks 2 and 3 qualified using the Large Block scoped MIN/MAX indexes for columns **`campaign`** AND **`date`**.

3. Because the WHERE expression is an AND conjunction of values from both indexed columns, Large Blocks might be disqualified after comparing MIN/MAX index values for *only one* column.
4. With the current index threading model, Threads 1 and 4 complete as soon as their assigned Large Blocks are disqualified. Although this reduces thread concurrency for the remaining Small Block index subsetting work, it also means that fewer CPU cores are required to complete it. Using fewer CPU cores has a positive impact on overall CAS server performance when CPU resources are constrained, because other CAS sessions (or non-CAS processes) can use them instead.
5. Figure 6 shows that Threads 2 and 3 go on to qualify Small Blocks 2.1, 2.3, and 3.2 using Small Block index scope MIN/MAX values. All other Small Blocks in Large Blocks 2 and 3 were disqualified during these MIN/MAX comparisons.
6. In Figure 7, Threads 2 and 3 proceed to use the qualified Small Block indexes to locate the subset of data rows matching the WHERE expression.
7. Note the relative size differences of the Small Block 2.1, 2.3, and 3.2 indexes shown in Figure 7. Although the overall figure is *not* drawn to scale, the different size depiction is intentional and illustrates an important attribute about the Small Block index internal structure.
8. The nature of distributed data values is such that cardinality and the profile of unique versus duplicate values might differ considerably for values in the same column across different Small Blocks. CAS table indexes are highly optimized at the Small Block level to accommodate this variability so that each Small Block index occupies the least space possible without sacrificing algorithmic speed during index-based subsetting operations.
9. To summarize points 7 and 8, each Small Block index structure is constructed to optimally adapt to its column data values in the Small Block.
10. When a conjunctive WHERE expression contains multiple indexed columns (for example, `campaign` AND `date`) being compared for equality, the CAS table index processor chooses the column index to search first based on which index has the *highest unique value set cardinality.*
11. The simple rationale for point 10 is that searching among a greater number of unique values in a Small Block index yields a higher probability of returning fewer matching rows (that is, more unique data values in the Small Block generally mean fewer duplicate rows per unique value).
12. When taking the intersection of values in two or more subsets, it is optimal to first determine the subset that is likely to have the fewest values. Doing this at the Small Block level is another way CAS table indexes *supercharge* data subsetting.
13. Only the subset of data rows containing values that match the WHERE expressions for `campaign` AND `date` are returned. The Small Block indexes provide row locators for matching unique values and all their duplicates. The rows might be physically non-contiguous within each Small Block. The WHERE processor works cooperatively with CAS table indexing to optimize the intersection of selected column data values based on the expression arguments.

*Unless otherwise stated, elapsed and CPU times listed in this paper were measured for cases where all CAS table Large Blocks, including their integrated index structures, fit entirely in real memory.*

The cost of subsetting on indexed columns depends very much on the size of the result set. CPU and elapsed time consumption is reduced in proportion to decreases in the resulting subset size (the smaller the subset, the more dramatic the time savings). Subsetting on indexed columns should deliver measurable performance benefits when all CAS table Large Blocks fit in memory. When real memory is constrained to the point where most or all Large Blocks do *not* fit, then WHERE expressions referencing indexed variables might perform worse than without indexing. CAS R&D is actively working on this problem.

NOTE: for paritioned CAS tables, SAS Viya 3.3 does not utilize index-based subsetting for WHERE expressions that specify indexed columns *and* columns based on the partition key. In SAS Viya 3.3 optimized partition key lookup takes precedence and the WHERE processor compares all other specified column values, including indexed columns, *within* selected partitions *without* querying indexes. Future releases will optimize subsetting to use partitioning and indexing together.

## DETERMINING COLUMNS TO INDEX

### EXPLORING CAS TABLE DATA

The following PROC CAS (CASL) programming examples are for an ADS table containing over 78 million rows of sample internet-related advertising data. Total data size is just under 17 gigabytes, making ADS a good medium-scale case for determining the benefits of indexing CAS table columns of various data types, cardinality, and value distribution.

The following PROC CAS (CASL) code fragments assume a session connection from SAS®9 to a running SMP CAS server in a SAS Viya 3.3 deployment. The ADS table is already loaded into CAS for these examples.

**Output 1. Table Info**

```
table.tableInfo / name="ADS";

TableInfo Table Information for Caslib ADSLIB
Name         Rows     Columns Indexed Columns Encoding Loaded Source
----------- -------- ------- --------------- -------- -------------
ADS         78774180      16               0 utf-8    ADS.sashdat
```

**Output 2. Column Info**

```
table.columnInfo / table="ADS";

ColumnInfo Column Information for ADS in Caslib ADSLIB
Column        ID Type    RawLength FormattedLength Format
------------- -- ------- --------- --------------- ------
device_type    1 char            6               6
bounce         2 char            1               1
origin_type    6 char           17              17
campaign       7 char           18              18
landing_page   8 varchar       494             494
visit_counts   9 double          8              12
page_views    10 double          8              12
page_duration 11 double          8              12
conversions   12 double          8              12
bouncers      13 double          8              12
revenue       14 double          8              12
session_cnt   15 double          8              12
date          16 double          8               9 DATE
```

**Output 3. Distinct Counts**

```
simple.distinct / table={name="ADS"};

Distinct Distinct Counts for ADS
Analysis Variable Number of Distinct Values Number of Missing Values
----------------- ------------------------- ------------------------
device_type                               6                        0
bounce                                    2                        0
origin_type                              11                        0
campaign                                 16                        0
landing_page                          96282                        0
visit_counts                           3544                        0
page_views                             3932                        0
page_duration                        139287                        0
conversions                             247                        0
bouncers                                151                        0
revenue                                8958                        0
session_cnt                             579                        0
date                                    366                        0
```

Along with information about the table, its columns, and their distinct values, common data use cases suggest that the columns highlighted in gray in **Output 3. ADS Column Values Distinct Counts** will benefit from indexing. The **`campaign`** column is chosen for the first indexing example because each unique `campaign` value subset is often analyzed. Consider the distribution of ADS `campaign` values in the following output. Sixteen distinct values at first seems like very low cardinality column data relative to the ADS row count of ~78 million. Nevertheless, the following measurements show that indexing on `campaign` represents a good space-time tradeoff of computing resources when subsetting over the range of its distinct values is required.

**Output 4. Column Values Frequency Distribution**

```
simple.freq / table={name="ADS",vars={{name="campaign"}}};

Frequency Frequency for ADS
Analysis Variable Character Value Formatted Value Level Frequency
----------------- --------------- --------------- ----- ---------
campaign          AOL             AOL                 1     19032
campaign          Bing            Bing                2    714798
campaign          Bookmark        Bookmark            3  37426062
campaign          Brand Aware     Brand Aware         4   6133428
campaign          com             com                 5    845460
campaign          CRM             CRM                 6    129198
campaign          Facebook        Facebook            7   2458056
campaign          google          google              8  26411292
campaign          Instagram       Instagram           9      2196
campaign          Pace            Pace               10   1267824
campaign          pricecomp       pricecomp          11       366
campaign          Prospects       Prospects          12     25986
campaign          Trading         Trading            13   2068998
campaign          Twitter         Twitter            14      4026
campaign          very.co.uk      very.co.uk         15   1092510
campaign          Yahoo           Yahoo              16    174948
```

## INDEXING ONE COLUMN

Here the new SAS Viya 3.3 CAS action `table.index` is used to create an index on the `campaign` column. This results in the new in-memory table ADS_INDEXED, as shown here in Output 5.

**Output 5. Indexing ADS on One Column**

```
   table.index /
         casout={indexVars={"campaign"},name="ADS_INDEXED"}
         table={name="ADS"};

   [Performance] Elapsed 5.718, CPU 20.471, Nodes 1
```

As evidenced by the greater than 3-to-1 ratio of CPU time to elapsed time, the `table.index` action executes concurrent threads in parallel as it scans ADS table Small Blocks and construct indexes based on the `casout={indexVars={<vars>}}` specification. The ADS_INDEXED table with its `campaign` column indexed on 78,774,180 rows is created in less than 6 seconds elapsed time!

To validate indexing performance benefits, it is useful to compare CPU time and elapsed time for the ADS table versus the ADS_INDEXED table (with the `campaign` column indexed). For the first comparison, the two highest frequency `campaign` column values are taken together, resulting in a very large result subset (81% of table rows). The four `summary` pair measurements that follow subset on a sample of progressively lower frequency values from the distribution.

Here is the CASL code invoked for each subsetting measurement pair:

```
  simple.summary / subSet={"N" "MIN" "MAX" "MEAN"}
                   table={name="ADS",vars="revenue",where="campaign=<value>"};

  simple.summary / subSet={"N" "MIN" "MAX" "MEAN"}
                   table={name="ADS_INDEXED",vars="revenue",where="campaign=<value>"};
```

The summary actions compute only N, MIN, MAX, and MEAN. This is done so that WHERE expression processing time is not masked by the additional overhead required to compute all summary statistics. **Table 1. Time Reductions Samples with campaign  Indexed** shows the elapsed and CPU time reductions calculated for each resulting subset in the ADS versus ADS_INDEXED tables.

**Table 1. Time Reductions Samples with `campaign` Column Indexed**

| campaign value | Table | N | Elapsed Time | Elapsed Time Reduction | CPU Time | CPU Time Reduction |
|---|---|---|---|---|---|---|
| Bookmark \| google | ADS | 63837354 | 0.612 | | 13.282 | |
| | ADS_INDEXED | 63837354 | 0.475 | 1.28x | 9.255 | 1.44x |
| Bookmark | ADS | 37426062 | 0.494 | | 9.910 | |
| | ADS_INDEXED | 37426062 | 0.298 | 1.66x | 5.909 | 1.68x |
| Brand Aware | ADS | 6133428 | 0.310 | | 9.087 | |
| | ADS_INDEXED | 6133428 | 0.040 | 7.75x | 0.703 | 14.04x |
| Bing | ADS | 714798 | 0.436 | | 9.767 | |
| | ADS_INDEXED | 714798 | 0.022 | 19.8x | 0.138 | 70.77x |
| pricecomp | ADS | 366 | 0.411 | | 8.196 | |
| | ADS_INDEXED | 366 | 0.009 | 45.7x | 0.026 | 315.23x |

**NOTE:** these measurements were done with a SAS Viya 3.3 Linux CAS SMP server running on a real machine configured with Intel Xeon CPUs E5-2450 v2 @ 2.50GHz having a total of 32 total cores. Each summary action run used 32 threads.

As expected, these measurements prove that indexing the **`campaign`** column reduces subsetting time progressively as the result set gets smaller and smaller. Although the WHERE expression `campaign IN( “Bookmark”,“google”)` returns 81% table data rows, ADS_INDEXED still measures elapsed and CPU time reduction!

## Memory Overhead with the `campaign` Column Indexed

The `table.details` action provides detailed CAS table space usage information.

**Output 6. Table Details with One Column Indexed**

```
table.tableDetails / name="ADS_INDEXED";

TableDetails Detail Information for ADS_INDEXED in Caslib ADSLIB.
Node Blocks Active Rows     IndexSize DataSize    VardataSize Mapped MappedMemory
---- ------ ------ -------- --------- ----------- ----------- ------ ------------
ALL    2048   2048 78774180   2523376 16981403382  5637921462   2048  16984278424
```

The following values are directly relevant to data distribution and index space consumption. The Blocks value (2048) represents the total number of Small Blocks in the table. DataSize is the total number of bytes occupied by both fixed and variable (VardataSize indicates this quantity) table row data. IndexSize is total byte quantity (2543520) consumed by all indexing-related data structures in the table. With the **`campaign`** column indexed, this is 0.015% of DataSize.

This computation along with previous measurements validate our hypothesis that indexing the **`campaign`** column reduces both elapsed and CPU time, with very minimal overhead for the index structures. The performance benefits of indexing the **`campaign`** column are clearly demonstrated. The next section explores alternative methods to create indexes, including indexing multiple columns in one pass and using different input sources.

## CREATING CAS TABLES WITH INDEXES

In this section, we begin by creating a new version of the ADS_INDEXED table indexed on the four selected columns shown here in Output 7.

**Output 7. Indexed Column Distinct Counts**

```
Analysis Variable Number of Distinct Values
----------------- -------------------------
campaign                                 16
landing_page                          96282
page_views                             3932
date                                    366
```

### INDEXING MULTIPLE COLUMNS WITH THE INDEX ACTION

The `table.index` action shown in Output 8 creates a new ADS_INDEXED table indexed on the four columns specified in the `indexVars={<vars>}` parameter list.

**Output 8. Indexing a Table on Four Columns**

```
table.index /
      casout={indexVars={ "date" "campaign" "page_views" "landing_page"},
      name="ADS_INDEXED",replace=true}
      table={name="ADS"};

[Performance] Elapsed 22.003, CPU 78.623, Nodes 1
```

Here indexing four columns results in CPU and elapsed times approximately 4 times greater than shown in **Output 5. Indexing ADS on One Column**. The near linear CPU time scaling is due to the `table index` action passing the ADS input data rows only once and working on many Small Blocks in parallel as it indexes the four columns. CPU time or elapsed time variations might occur between different runs when other workloads are active on the machine(s) running CAS. This is often due to the CAS table mapping model requiring page-ins to real memory or index action threads waiting for available CPU cores. Elapsed time can increase sharply when significant portions of the input table are not memory-resident and significant page-ins result.

**Output 9. Table Details with Four Columns Indexed**

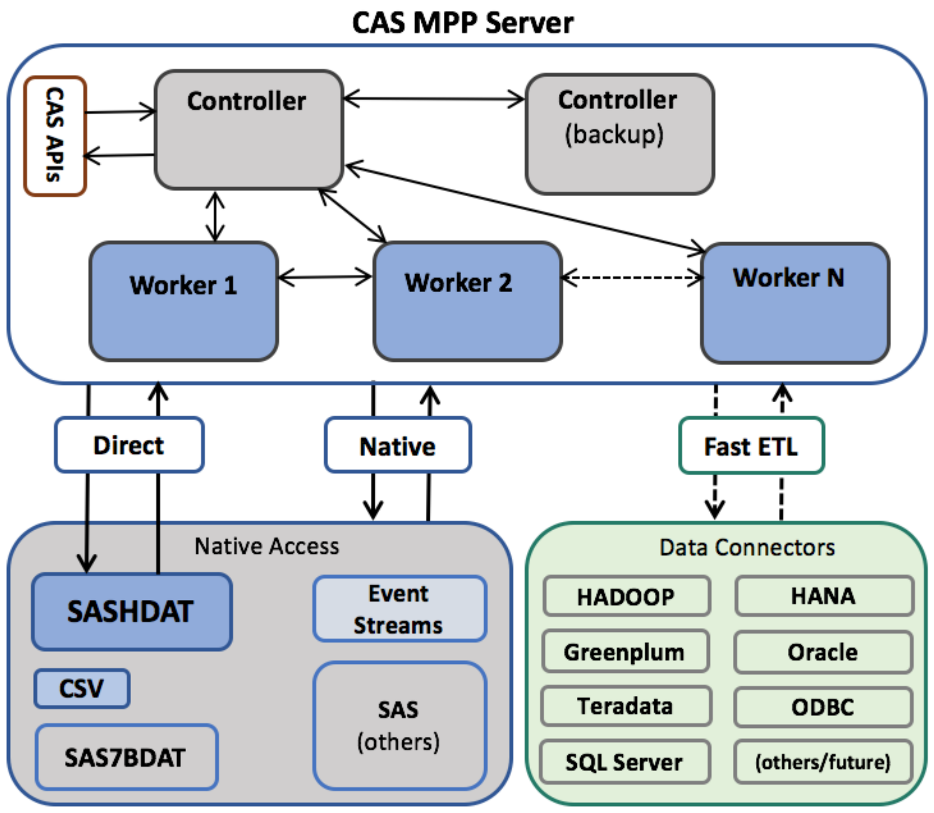
```
table.tableDetails / name="ADS_INDEXED";

TableDetails Detail Information for ADS_INDEXED in Caslib ADSLIB.
Node Blocks Active Rows     IndexSize DataSize    VardataSize Mapped MappedMemory
---- ------ ------ -------- --------- ----------- ----------- ------ ------------
ALL    2048   2048 78774180 345351664 16981403382  5637921462   2048  17327156104
```

With four columns now indexed, total memory overhead has grown to 2% of DataSize for ADS_INDEXED. This is nominal compared to the CPU and elapsed time reductions that normally result when these columns are referenced in subsetting WHERE expressions. MEASURING INDEXING PERFORMANCE shows these reductions for several WHERE expression variants.

### INDEXING COLUMNS AS DATA IS LOADED

In-memory CAS tables might be created from various data sources, as illustrated in **Figure 8. CAS Data Sources.** Currently, the only data source CAS can process without intermediate transformation are tables saved in the native SASHDAT format. SASHDAT Large Blocks are memory-mapped *directly* into CAS server nodes *without* the ETL process all other data sources require. In order to index columns in a saved SASHDAT file in SAS Viya 3.3, the file must first be memory-mapped into CAS by the `loadTable` action, followed by running the `table.index` action. Columns might be indexed for a new in-memory table as it is created from data sources during the fast ETL process shown in Figure 8.

**Figure 8. CAS Data Sources**



## CSV Data Source

The following CASL example creates the same ADS_INDEXED table shown in **Output 8. Indexing a Table on Four** Columns by indexing specified columns (as shown in the highlighted code) as the ADS.csv file is parsed and rows are added to the in-memory CAS table:

```
table.loadTable /  path="ADS.csv",
    casout={name="ADS_INDEXED",
            blockSize=8388608,
            indexVars={"campaign","landing_page","page_views","date"}},
    importOptions={fileType="csv",
       vars={
             {name="device_type"    ,type="char"    ,formattedLength=6   ,length=6     },
             {name="bounce"         ,type="char"    ,formattedLength=1   ,length=1     },
             {name="visitor_type"   ,type="char"    ,formattedLength=1   ,length=1     },
             {name="campaign"       ,type="char"    ,formattedLength=18  ,length=18    },
             {name="landing_page"   ,type="varchar" ,formattedLength=494               },
             {name="visit_counts"   ,type="double"  ,formattedLength=12                },
             {name="page_views"     ,type="double"  ,formattedLength=12                },
             {name="page_duration"  ,type="double"  ,formattedLength=12                },
             {name="conversions"    ,type="double"  ,formattedLength=12                },
             {name="bouncers"       ,type="double"  ,formattedLength=12                },
             {name="revenue"        ,type="double"  ,formattedLength=12                },
             {name="session_cnt"    ,type="double"  ,formattedLength=12                },
             {name="date"           ,type="double"  ,formattedLength=9   ,format="DATE"}
            }}
;
```

**NOTE: blockSize=8388608** sets the approximate size of each Small Block created in the ADS_INDEXED table. This value helps reduce total index memory overhead and slightly improves subsetting performance for the cases measured in this paper.

### Info Actions Output with Indexes

Here's how `tableInfo` and `columnInfo` output looks when the table has indexed columns:

#### Output 10. Table Info with Indexes

```
table.tableInfo / name="ADS_INDEXED";

TableInfo Table Information for Caslib ADSLIB
Name        Rows     Columns Indexed Columns Encoding Loaded Source
----------- -------- ------- --------------- -------- -------------
ADS_INDEXED 78774180      16               4 utf-8    ADS.csv
```

#### Output 11. Column Info with Indexes

```
table.columnInfo / table="ADS_INDEXED";

ColumnInfo Column Information for ADS_INDEXED in Caslib ADSLIB
Column        ID Indexed Type    RawLength FormattedLength Format
------------- -- ------- ------- --------- --------------- ------
device_type    1 N       char            6               6
bounce         2 N       char            1               1
origin_type    3 N       char           17              17
campaign       4 Y       char           18              18
landing_page   5 Y       varchar       494             494
visit_counts   6 N       double          8              12
page_views     7 Y       double          8              12
page_duration  8 N       double          8              12
conversions    9 N       double          8              12
bouncers      10 N       double          8              12
revenue       11 N       double          8              12
session_cnt   12 N       double          8              12
date          13 Y       double          8               9 DATE
```

### SAS7BDAT Data Source

The following CASL example creates an instance of the ADS_INDEXED table shown in **Output 8. Indexing a Table on Four** Columns by indexing the specified columns as the ads.sas7bdat file is imported into CAS:

```
table.loadTable /
    casOut={name="ADS_INDEXED",
            indexVars={"date" "campaign" "page_views" "landing_page"},
            replace=true}
    importOptions={charMultiplier=2.0,fileType="basesas"}
    path="ads.sas7bdat";
```

### Saving CAS Tables That Have Indexes

No special parameters are required to write a CAS table having indexed columns to a SASHDAT file with the `table.save` action. In SAS Viya 3.3, SASHDAT files are supported in caslib types HDFS, DNFS, and Path. The following CASL code saves ADS_INDEXED to the path referenced by the current default caslib:

```
table.save / name="ADS_INDEXED" replace=true table={name="ADS_INDEXED"};
```

As explained in the INDEX STRUCTURE section, CAS table indexes are integrated within table blocks and are *not* saved to a different file.

## INDEXING VARIABLES IN CAS ACTION OUTPUT TABLES

As shown in the previous `loadTable` examples, the `casout={indexVars={<vars>}}` parameter creates indexes on new CAS in-memory tables. This is good news, because `casout` is a common parameter on many CAS analytics and data management actions. Just as `loadTable` can index columns for the new in-memory table created as the data is imported, other CAS actions can index columns in their output tables. Programs using CAS APIs might get a big win from indexing columns on the output of invoked CAS actions, especially when that output is large or contains high-cardinality values that are often subset.

This example runs the `simple.freq` action to create a frequency distribution of the ADS table with the `_Column_` and `_Frequency_` columns indexed in the frequency output:

```
   simple.freq /
         casOut={name="ADS_FREQ_INDEXED",
                 indexVars={"_Column_" "_Frequency_"},
                 replace=true}
         table={name="ADS"};
```

**Output 12. FREQ Output Table with Indexed Columns**

```
ColumnInfo Column Information for ADS_FREQ_INDEXED in Caslib ADSLIB
Column      ID Indexed Type    RawLength FormattedLength Format
----------- -- ------- ------- --------- --------------- ------
_Column_     1 Y       char           13              13 $
_Numvar_     2 N       double          8              12 BEST
_Charvar_    3 N       varchar       494             494
_Fmtvar_     4 N       varchar       494             494
_Level_      5 N       double          8              12 BEST
_Frequency_  6 Y       double          8              12 BEST
```

To compare the benefits of indexing, a non-indexed version is also created:

```
   simple.freq /
         casOut={name="ADS_FREQ",
                 replace=true}
         table={name="ADS"};
```

Now for some real fun. Suppose that your SAS®9 CASL program needs to fetch the conjunction of indexed columns from the output table of a SAS analytics CAS action. This FREQ table only has 253,381 rows, and your SAS program gets this row back 7 times faster with 14 times less CPU time. For truly big data with lots of high-cardinality valued columns, these FREQ tables could have millions of rows. Imagine the benefit, in those cases, of indexing analytics action output table columns for down-stream subsetting, even when the result subset has hundreds of rows.

**Output 13. Fetching FREQ Table Subsets with Index Columns**

```
table.fetch /
      table={name="ADS_FREQ_INDEXED",where="_Column_ = 'device_type' AND _Frequency_ = 9882"}

Fetch Selected Rows from Table ADS_FREQ_INDEXED
_Index_ _Column_    _Numvar_     _Charvar_ _Fmtvar_ _Level_      _Frequency_
------- ----------- ------------ --------- -------- ------------ ------------
      1 device_type            . tv        tv                  3         9882
 [Performance] Elapsed 0.002, CPU 0.001, Nodes 1

table.fetch /
      table={name="ADS_FREQ",where="_Column_ = 'device_type' AND _Frequency_ = 9882"}

Fetch Selected Rows from Table ADS_FREQ
_Index_ _Column_    _Numvar_     _Charvar_ _Fmtvar_ _Level_      _Frequency_
------- ----------- ------------ --------- -------- ------------ ------------
      1 device_type            . tv        tv                  3         9882
 [Performance] Elapsed 0.014, CPU 0.014, Nodes 1
```

## MEASURING INDEXING PERFORMANCE

In this section, CPU and elapsed times are measured for three different WHERE expression cases using the minimal `summary` action parameters introduced in the INDEXING ONE COLUMN section. The measurements compare the ADS table, which has no indexed columns, to ADS_INDEXED with four indexed columns, with at least one referenced in the WHERE expression of each case.

### METHODOLOGY

CPU and elapsed times measurements were done with a SAS Viya 3.3 Linux CAS SMP server running on a real machine configured with Intel Xeon CPUs E5-2450 v2 @ 2.50GHz having a total of 32 total cores. To approximate running machines of differing computing power or workloads, each `summary` action pair is reported five times per case with the action `nThreads=t` parameter set with t representing the descending sequence: `{32,16,8,4,2}`.

Measuring each action pair by varying maximum available threads accomplishes two things:

1. It provides a relative measure of the time reductions demonstrated by indexing-based subsetting independent of different machine configuration differences.
2. Measuring progressively lower maximum available threads simulates running on machines with less CPU power or a heavier system workload with fewer cores available to CAS.

For all measurements, the ADS and ADS_INDEXED tables fit entirely in real memory. Each summary action was run three times, with the best overall time reported.

### HDAT INDEX QUERY LOGGER

CAS supports the following new logger in SAS Viya 3.3:

```
<logger name="App.cas.table.HDAT.index.query">
   <level value="trace"/>
</logger>
```

The CAS server logs per-thread entries for WHERE expressions that reference indexed columns. **Output 15. Case 1 Index** Query Logger shows an example. This logger reports selected Large Blocks, Small Blocks, and matching rows for each available thread.

### MEASUREMENT CASES

Each of the three cases reports these items in order:

1. a statement introducing the subsetting WHERE expression
2. the `summary` action CASL source code
3. the `summary` action output
4. a table of measurements and time reductions for `nThreads= {32,16,8,4,2}`
5. Index Query Logger output for `nThreads=2`

### INDEX AND WHERE PROCESSOR COOPERATION

CASE 3 shows that logger `rows matching` value is significantly higher than the summary output `N` value. The `rows matching` value from the `Index Use Total` logger line represents the initial subset of rows determined when the index processor searches the index set of one indexed column per Small Block. Then the WHERE expression processor applies the Boolean AND to this initial row subset—an example of index and WHERE processor cooperation. CAS R&D is working on other optimizations when multiple columns from the WHERE expression are indexed.

## CASE 1: VARYING CHARACTER STRINGS

Subset on a column of varying character type containing a web URL value:

```
   simple.summary /
          subSet={"N" "MIN" "MAX" "MEAN"}
          table={name="ADS",
                 where="landing_page='http://www.very.co.uk/joe-browns-fabulous-
                                        faux-fur-collar-jacket/1600107394.prd'
                    AND revenue > 0"}, nthreads=t;

   simple.summary /
          subSet={"N" "MIN" "MAX" "MEAN"}
          table={name="ADS_INDEXED",
                 where="landing_page='http://www.very.co.uk/joe-browns-fabulous-
                                        faux-fur-collar-jacket/1600107394.prd'
                    AND revenue > 0"}, nthreads=t;
```

**Output 14. Case 1 Summary Output**

```
Analysis Variable Minimum Maximum N   Mean
----------------- ------- ------- --- -------
visit_counts          143     143 366     143
page_views            144     144 366     144
page_duration     3637209 3637209 366 3637209
conversions             0       0 366       0
bouncers                0       0 366       0
revenue                55      55 366      55
session_cnt             1       1 366       1
date                20129   20494 366 20311.5
```

**Table 2. Case 1 Measurements**

| nThreads value | Table | N | Elapsed Time | Elapsed Time Reduction | CPU Time | CPU Time Reduction |
|---|---|---|---|---|---|---|
| 32 | ADS | 366 | 0.386 | | 8.569 | |
| | ADS_INDEXED | 366 | 0.009 | **42.89x** | 0.025 | **342.76x** |
| 16 | ADS | 366 | 0.482 | | 6.736 | |
| | ADS_INDEXED | 366 | 0.005 | **96.4x** | 0.025 | **269.44x** |
| 8 | ADS | 366 | 0.725 | | 5.665 | |
| | ADS_INDEXED | 366 | 0.006 | **120.83x** | 0.019 | **298.16x** |
| 4 | ADS | 366 | 1.399 | | 5.493 | |
| | ADS_INDEXED | 366 | 0.006 | **233.17x** | 0.016 | **343.31x** |
| 2 | ADS | 366 | 2.745 | | 5.407 | |
| | ADS_INDEXED | 366 | 0.009 | **305x** | 0.014 | **386.21x** |

**Output 15. Case 1 Index Query Logger**

```
Logger: TRACE App.cas.table.HDAT.index.query

HDAT Large Block total=565, selected=565; Small Block total=1029, selected=1029
HDAT Large Block total=564, selected=564; Small Block total=1019, selected=1019
Thread: 0 : Small Blocks providing rows=1, rows matching query=366
Thread: 1 : Small Blocks providing rows=0, rows matching query=0
Index Use Total : Small Blocks providing rows=1, rows matching query=366
```

## CASE 2: NUMERIC DATA RANGE

Subset on a column of type DOUBLE:

```
simple.summary /
      subSet={"N" "MIN" "MAX" "MEAN"}
       table={name="ADS",
              where="Date BETWEEN '10Feb2015'd AND '10Mar2015'd"},
       nThreads=t;

 simple.summary /
       subSet={"N" "MIN" "MAX" "MEAN"}
       table={name="ADS_INDEXED",
              where="Date BETWEEN '10Feb2015'd AND '10Mar2015'd"},
       nThreads=t;
```

**Output 16. Case 2 Summary Output**

```
Analysis Variable Minimum Maximum     N       Mean
----------------- ------- ----------- ------- ----------------
visit_counts            1      919702 6241670     172.04645263
page_views              1     1650888 6241670     234.77774938
page_duration           0 59263096147 6241670 8556461.83588719
conversions             0       14297 6241670       0.95109882
bouncers                0        9815 6241670       0.71756261
revenue                 0  1427995.16 6241670     134.86656163
session_cnt             1       53403 6241670       6.33652372
date                20129       20157 6241670            20143
```

**Table 3. Case 2 Measurements**

| nThreads value | Table | N | Elapsed Time | Elapsed Time Reduction | CPU Time | CPU Time Reduction |
|---|---|---|---|---|---|---|
| 32 | ADS | 6241670 | 0.433 | | 9.285 | |
| | ADS_INDEXED | 6241670 | 0.091 | 4.76x | 1.826 | 5.08x |
| 16 | ADS | 6241670 | 0.496 | | 7.009 | |
| | ADS_INDEXED | 6241670 | 0.122 | 4.07x | 1.600 | 4.38x |
| 8 | ADS | 6241670 | 0.796 | | 6.236 | |
| | ADS_INDEXED | 6241670 | 0.193 | 4.12x | 1.475 | 4.23x |
| 4 | ADS | 6241670 | 1.558 | | 6.138 | |
| | ADS_INDEXED | 6241670 | 0.370 | 4.21x | 1.457 | 4.21x |
| 2 | ADS | 6241670 | 3.098 | | 6.113 | |
| | ADS_INDEXED | 6241670 | 0.724 | 4.28x | 1.446 | 4.28x |

**Output 17. Case 2 Index Query Logger**

```
Logger: TRACE App.cas.table.HDAT.index.query

HDAT Large Block total=560, selected=560; Small Block total=1036, selected=1036
HDAT Large Block total=561, selected=561; Small Block total=1012, selected=1012
Thread: 0 : Small Blocks providing rows=1012, rows matching query=3083643
Thread: 1 : Small Blocks providing rows=1036, rows matching query=3158027
Index Use Total : Small Blocks providing rows=2048, rows matching query=6241670
```

**NOTE:** WHERE argrument values for the `date` column are uniformly distributed across Small Blocks. This explains why elapsed time reduction is not more dramatic.

## CASE 3: CONJUNCTION OF TWO INDEXED COLUMNS

Subset on a column of type DOUBLE containing date values:

```
simple.summary /
      subSet={"N" "MIN" "MAX" "MEAN"}
      table={name="ADS",
             where=" campaign = 'Prospects' AND Date = '15Apr2015'd"},
      nThreads=t;

simple.summary /
      subSet={"N" "MIN" "MAX" "MEAN"}
      table={name="ADS_INDEXED",
             where=" campaign = 'Prospects' AND Date = '15Apr2015'd"},
      nThreads=t;
```

### Output 18. Case 3 Summary Output

```
Analysis Variable Minimum Maximum   N  Mean
----------------- ------- --------- -- ----------------
visit_counts            1      4508 71     113.69014085
page_views              1      4508 71     113.69014085
page_duration         182 137262510 71 4633955.11267606
conversions             0         0 71                0
bouncers                0         2 71       0.11267606
revenue                 0         0 71                0
session_cnt             1        70 71        2.5915493
date                20193     20193 71            20193
```

### Table 4. Case 3 Measurements

| nThreads value | Table | N | Elapsed Time | Elapsed Time Reduction | CPU Time | CPU Time Reduction |
|---|---|---|---|---|---|---|
| 32 | ADS | 71 | 0.386 | | 7.396 | |
| | ADS_INDEXED | 71 | 0.013 | **29.69x** | 0.116 | **67.76x** |
| 16 | ADS | 71 | 0.413 | | 5.877 | |
| | ADS_INDEXED | 71 | 0.011 | **37.55x** | 0.099 | **59.36x** |
| 8 | ADS | 71 | 0.659 | | 5.124 | |
| | ADS_INDEXED | 71 | 0.014 | **47.1x** | 0.090 | **56.93x** |
| 4 | ADS | 71 | 1.264 | | 4.941 | |
| | ADS_INDEXED | 71 | 0.025 | **50.56x** | 0.088 | **56.15x** |
| 2 | ADS | 71 | 2.470 | | 4.873 | |
| | ADS_INDEXED | 71 | 0.045 | **54.89x** | 0.085 | **57.33x** |

### Output 19. Case 3 Index Query Logger

```
Logger: TRACE App.cas.table.HDAT.index.query

HDAT Large Block total=560, selected=560; Small Block total=1036, selected=1036
HDAT Large Block total=561, selected=561; Small Block total=1012, selected=1012
Thread: 0 : Small Blocks providing rows=1012, rows matching query=106340
Thread: 1 : Small Blocks providing rows=1036, rows matching query=108890
Index Use Total : Small Blocks providing rows=2048, rows matching query=215230
```

## COST AND SCALABILITY CONSIDERATIONS

This paper argues for performance benefits obtained by indexing CAS table columns that are frequently used in subsetting WHERE expressions. Example tables ADS and ADS_INDEXED each contain 78,774,180 rows with a storage footprint of approximately 17 gigabytes (ADS_INDEXED is about 340 megabytes larger due to its integrated index data structures). This is certainly on the small side of big data by 2018 standards.

After examining the CPU and elapsed time reductions reported by measuring WHERE expressions with indexed variables, the reader might observe that some of the measured cases produce the desired subset in less than one second elapsed time *without* indexing. As a result, the reader might conclude that indexing is a waste of time and storage overhead. The following questions and points help address this conclusion:

1. Does your application pay directly for CPU utilization?  If it does, and your programs perform WHERE operations that return small-to-medium sized subsets, then CAS table indexing is likely to be a win, especially if the minimal number of columns are indexed to keep memory and storage overhead low.
2. Do your programs frequently perform WHERE operations that return resulting row subsets that are 5% or less of the total number of table rows?  If they do, then indexing is a sure win.
3. Are you subsetting on medium-to-long varying character strings with WHERE operations that return relatively small resulting row subsets?   If you are, then indexing will definitely provide benefits.
4. Are subsetting performance gains so important to your application that you would need to create smaller CAS tables by copying subset portions of a larger table if indexing were not available?  If so, then indexing is huge win, because the memory or storage overhead of indexing even several columns is infinitesimal compared to having disparate copied subsets of a very large CAS table.
5. Are your tables promoted to global-scope and shared among concurrent CAS sessions where subsetting WHERE operations are frequently performed?  If so, then indexing should improve overall CAS server performance and help with scalability as the number of concurrent sessions increases.

All performance measurements in this paper were done on a CAS SMP server to demonstrate the most direct benefits of indexing for the smaller side of big data. For tables the size of ADS, non-indexed measurements will report better elapsed times on a CAS MPP server with the data distributed across more CPU cores. However, for many subsetting cases, the CPU times could be up to three orders of magnitude greater than *without* indexes. The author has measured much larger (for example, 200 million rows with 250 gigabytes data total) tables distributed across many CAS MPP Worker nodes. Indexing produces similar benefits for these tables as those shown in the SMP measurements reported for the ADS cases here.

## CONCLUSION

This paper summarizes the benefits and computing resource tradeoffs for the first version of CAS table indexing now available in SAS Viya 3.3. The design and implementation of this complex new facility was guided by these objectives:

- optimized index structures
- seamless integration in CAS
- measurable performance gains

Reducing the time required to query and subset data is the central reason for having indexes. Judicious use of CAS table indexes can significantly reduce CPU time consumption, resulting in better utilization of an expensive computing resource.

Different WHERE expressions can represent nearly endless combinations of column name references, argument values, Boolean operators, functions, and so on. Optimizing CAS table indexing for more WHERE expression variations is an on-going effort within CAS R&D. Future releases will support more data types for indexed columns and deeper integration between the WHERE processor and CAS table indexing.

## REFERENCES

SAS Institute Inc. 2017. *SAS Cloud Analytic Services 3.3: Fundamentals*. http://go.documentation.sas.com/?cdcId=calcdc&cdcVersion=3.3&docsetId=casfun&docsetTarget=titlepage.htm&locale=en (accessed February 15, 2018).

## AKNOWLEDGEMENTS

The author is grateful to fellow SAS Cloud Analytic Services R&D team members Gordon Keener, Steve Krueger, Mark Gass, and Jason Secosky; Ed Burnette, Matthias Ender and the SAS Cloud Analytic Enablement Team; Nabil Hoyek, and others in R&D who helped validate and refine CAS table indexing.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Brian Bowman
100 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc.
mailto:brian.bowman@sas.com
LinkedIn Profile: https://www.linkedin.com/in/brian-bowman-a791b5111/
http://www.sas.com