Paper SAS2027-2018

Scalable Cloud-Based Time Series Analysis and Forecasting

Thiago Quirino, Michael Leonard, and Ed Blair, SAS Institute Inc.

ABSTRACT

Many organizations need to process large numbers of time series for analysis, decomposition, forecasting, monitoring, and data mining. The TSMODEL procedure provides a resilient, distributed, optimized generic time series analysis scripting environment for cloud computing. It comes equipped with capabilities such as automatic forecast model generation, automatic variable and event selection, and automatic model selection. It also provides advanced support for time series analysis (in the time domain or in the frequency domain), time series decomposition, time series modeling, signal analysis and anomaly detection (for IoT), and temporal data mining. This paper describes the scripting language that supports cloud-based time series analysis. Examples that use SAS® Visual Forecasting software demonstrate the use of this scripting language.

INTRODUCTION

More information than ever before is being collected with associated timestamps. Computers, cell phones, smart devices, detectors, and other devices record timestamped data. These timestamped data can be modeled, forecasted, or mined (or any combination of these) for better decision making. In most cases, the decisions are critical and have immense financial and ethical implications. For example:

- Retailers rely on both seasonal and nonseasonal forecasts of product demand in order to make profitable decisions about staff scheduling and stocking levels for millions of products across thousands of stores (such as Walmart, Food Lion, and The Home Depot).
- Manufacturers rely on accurate forecasts of time to component failure in order to make decisions about the maintenance schedule of critical machinery components.
- Railroad companies rely on accurate time series forecasts of shipping demand per region of the country in order to preemptively stock their railroad cars across different regions. Accurate forecasts enable them to better meet the predicted demand, minimize shipping delays, and improve customer satisfaction.
- Energy companies rely on the ability to both monitor and analyze, in real time, sensor data that stream from wind turbines. Time series of sensor data are analyzed in order to quickly detect and respond to critical anomalous behavior and to maintain high turbine performance over time.
- Hospitals can aggregate patient sensor data, lab results, and physician notes in order to monitor patient progress and better predict patient outcome. Similarly, a physician can monitor a patient's pacemaker remotely in order to quickly determine when the patient's heart is behaving anomalously.
- Governments rely on time series decomposition techniques in order to decompose series of economic variables into their long-term trends and short-term seasonal effects so that they can gain a better insight into the real status of the economy.

In recent years, there has been an enormous increase in the amount of timestamped data being collected. It is now commonplace for companies (such as banks, manufacturers, retailers, websites, hospitals, universities, governments, and taxi, insurance, stock trading, phone, energy, and many more companies) to maintain large databases of timestamped data whose sizes range from hundreds of gigabytes to hundreds of terabytes. These databases are gold mines for insights into consumer behavior. These insights can help organizations optimize their internal processes to better meet consumer demands.

In addition, the amount of timestamped data being collected is expected to further escalate because of the ongoing proliferation of the Internet of Things (IoT). IoT enables all types of objects (cars, toasters, pacemakers, water and gas meters, and so on) to be discovered, monitored, and controlled remotely via the existing internet infrastructure. In short, “big data” has become pervasive in today’s society; it is everywhere and anything, it is here to stay, and it has a lot to say. Processing this ever-increasing amount of timestamped data in an intelligent way poses both architectural and analytical challenges. For example, because of the sheer amount of data and the ever-increasing demand to gain decision-making insights from data in close to real time, time series analysis of big data is inherently a distributed computing problem and is thus an architectural challenge. In addition, big data solutions must be generic enough to accurately handle the time series analysis requirements of different applications and thus are an analytical challenge.

SAS Visual Forecasting provides procedures for some of the most common analyses that are performed on timestamped data: forecasting, decomposition and price analysis, time series monitoring and anomaly detection, and temporal data mining. This paper provides an overview of the SAS Visual Forecasting Procedures—in particular of the TSMODEL procedure, which was specifically designed to support advanced, efficient, and cloud-based time series analysis of big data.

HOW DOES THE TSMODEL PROCEDURE WORK?

The goal of cloud-based time series analysis and forecasting is to perform an analytical task in a single pass through the data and by using a distributed file system or distributed computing environment (or both). Moving data can be I/O intensive, whether internal to a node, external (between computing nodes), or both. A single pass through the data allows for enormous performance gains. By providing a system that both moves data and computes efficiently, time series analysis and forecasting are possible on an enormous scale. The TSMODEL procedure provides a scalable, cloud-based time series analysis environment, which includes distributed file system, scripting, and parallel reading, execution, and writing. The following sections describe these elements in more detail.

DISTRIBUTED FILE SYSTEMS

The TSMODEL procedure is designed to enable your analysis to use a distributed file system (DFS). A DFS allows for redundant and resilient storage of data; it breaks up large files into chunks and stores each chunk on several storage media. In addition, it makes several redundant copies of each chunk in order to forgo the need for making periodic backup copies. If a particular file system fails, the distributed file system can resiliently heal itself without needing to restore backup copies, which can cause delays. However, the data are not stored contiguously in such a file system, so sorting on a particular file system is not possible. This is particularly problematic for time series analysis, where the ordering of the data is crucial. In addition, the data that are needed for time series analysis might be stored in several files. These distributed files must be read, sorted, and merged with respect to time in a scalable and efficient way. The SAS Visual Forecasting procedures automatically perform all these operations on the input time series data in preparation for the analysis.

Figure 1 illustrates a distributed file system that contains two tables, A and B. Each table is organized by classification (BY) variables that delineate the time series rows, which are grouped into seven BY groups. Each BY group represents one time series. One or more computing (worker) nodes are connected to the distributed file system; neither the tables nor the BY groups are stored on a single machine.

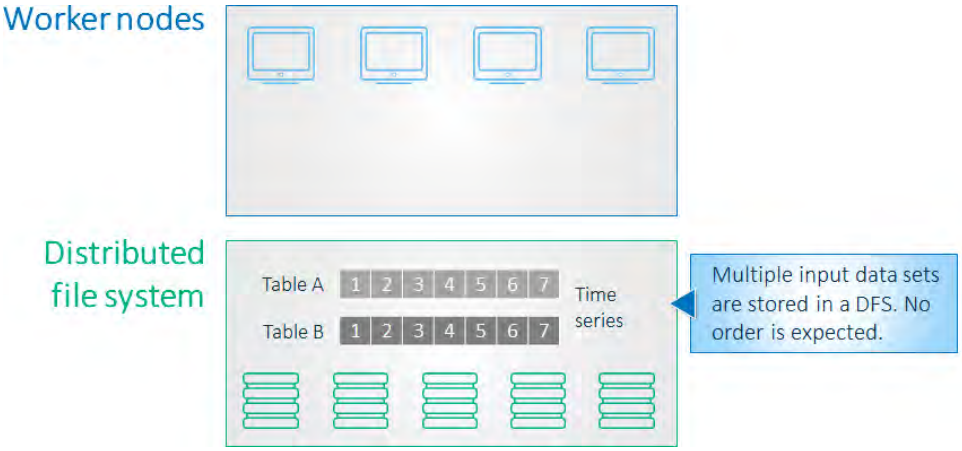

Figure 1. Distributed File System (DFS)

SCRIPTING LANGUAGE, DISTRIBUTION, AND COMPILATION

The vast amount of data that cloud computing can support calls for a time series analysis scripting environment in order to process the data efficiently. SAS Visual Forecasting provides a scripting language that facilitates the use of various capabilities, such as the following:

- automatic forecast model generation, automatic variable and event selection, and automatic model selection
- advanced support for time series analysis (in the time domain or in the frequency domain), time series decomposition, time series modeling, signal analysis and anomaly detection (for IoT), and temporal data mining
- preparation of the input data prior to analysis and postprocessing of the final results in the same script
- reading of multiple input data files and creation of multiple output data files

These features make the scripting language very flexible and useful for numerous applications. Figure 2 illustrates the use of a script. The script is created outside the computing server and can be submitted to the computing server by SAS, Python, Lua, or R clients.

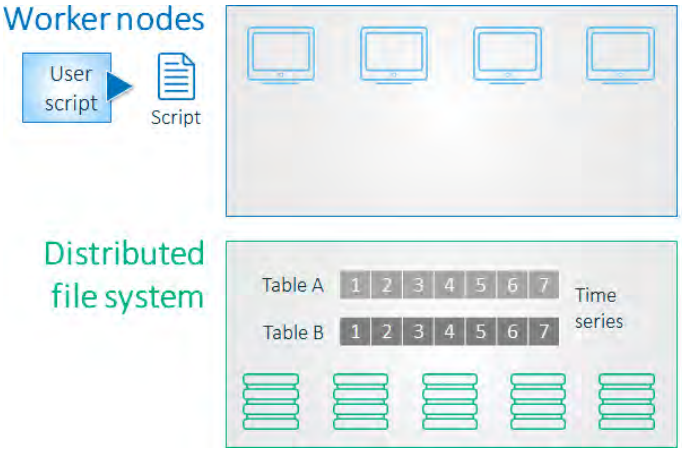

Figure 2. Scripting Language

The distributed network can consist of one or more computing (worker) nodes. After being submitted to the computing server, the user-specified script is distributed to each computing (worker) node to permit parallel execution of the specified analysis, as shown in Figure 3.

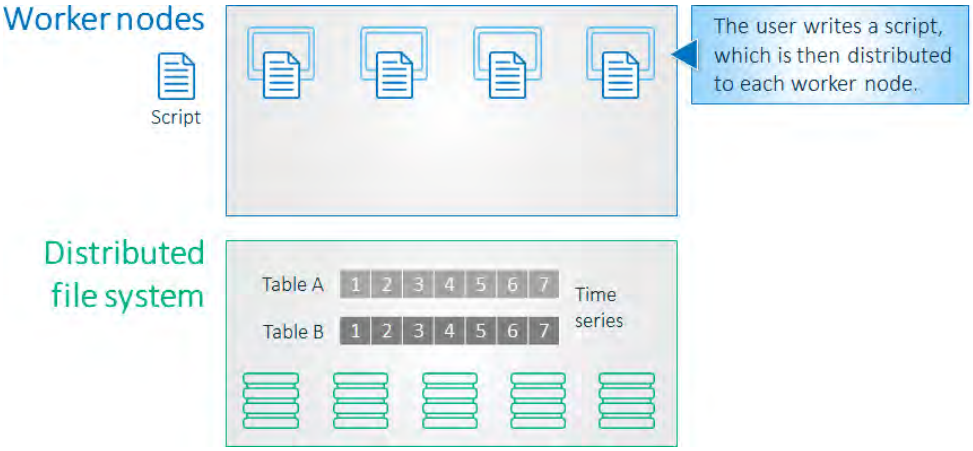

Figure 3. Script Distribution

The user-specified script is then compiled on each of the computing nodes. The compiler optimizes the resulting executable for the specific operating system of the computing node (Linux, Windows, and so on). This optimized executable permits very fast execution of the specified analysis.

Figure 4 illustrates the script compilation. After the script is distributed to the computing (worker) nodes, it is optimally compiled.

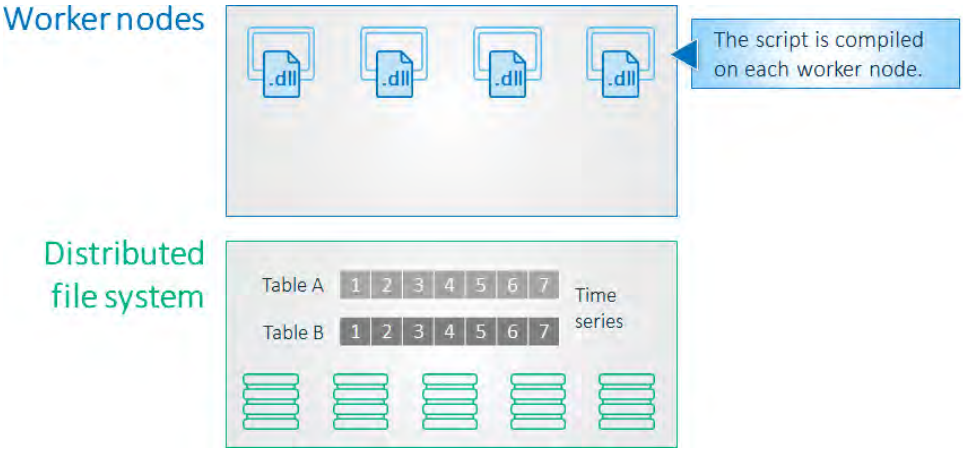

Figure 4. Script Compilation

PARALLEL READ

All the computing nodes read one or more input data files simultaneously. Each input data file contains unsorted, timestamped transactional data that might be recorded at no fixed interval. However, time series analysis algorithms typically require that the input time series data be stored contiguously in memory, in temporal order, and with a fixed-time interval. Therefore, the transactional data must be transformed into a suitable form prior to analysis. The TSMODEL procedure relies on the properties of the input data in order to determine how to transform the data with optimal performance. For example, when the input data consist of multiple time series (BY groups), then the transformation occurs via a two-step process that is illustrated in Figure 5 and subsequently described in detail.

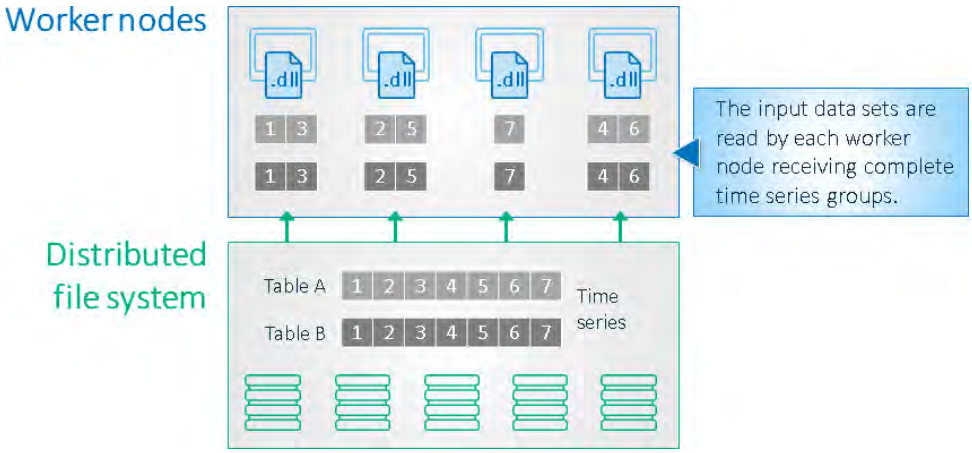

Figure 5. Parallel Read

Each time series is read from the DFS, shuffled, assigned to contiguous memory on a single computing (worker) node, and finally accumulated prior to the execution of the compiled script in the following two steps:

1. Automatic partitioning (auto-partitioning): This first step of the parallel read process involves reading and "shuffling" the transactional data so that each time series (each BY group) is copied into the memory of a single computing node. BY groups are delineated based on the distinct values of one or more columns of the input data files (BY-group variables). In the TSMODEL procedure, you can optionally specify BY-group variables via the BY statement. The TSMODEL procedure expects the main input data file, which you specify in the DATA= option in the PROC TSMODEL statement, to include all the BY-group variables. Any remaining auxiliary input data files, which you specify in the AUXDATA= option in the PROC TSMODEL statement, can include either all or none of the BY group variables. The TSMODEL procedure currently does not support partial BY-group variable matching.

   One of the goals of auto-partitioning is to evenly distribute the BY-group load across the computing nodes such that each node receives a balanced amount of both small and large BY groups. In this way, the overall computational load across the worker nodes is evened out. The shuffling process occurs differently for input data files that contain all the BY-group variables than it does for input data files that contain none of them. In the first case (the files contain all the BY-group variables), the process starts with worker nodes reading in parallel all the rows of the input data file from the DFS. Because of the unordered distribution of time series rows across the DFS, worker nodes end up with only chunks of different BY groups (partial BY groups) rather than complete time series in memory. Consequently, in order to attain complete BY groups, worker nodes must then transport the rows of these partial BY groups to their destination nodes (shuffling). In order to determine the destination node for each partial BY group, worker nodes convert the formatted text value of the ordered set of the BY-group variables that identify the BY group into a unique numeric hash value modulo the total number of computing nodes. The resulting numeric value corresponds to the ID of the computing node that will receive the partial BY-group data rows. After shuffling the rows of the partial BY groups, worker nodes then merge the rows that they received from other nodes into complete BY groups that are stored in contiguous memory. At the end of this process, the distributed transactional data that belong to each BY group are migrated via the network to the memory of a single computing node. At this stage, the input data file is said to have been "auto-partitioned." Moreover, the resulting new, in-memory data file is marked with metadata fields that prevent it from being auto-partitioned by the same ordered set of BY-group variables in the future. In turn, the TSMODEL procedure recognizes these metadata fields in input data files and skips the auto-partitioning step accordingly. In contrast, input data files that contain none of the BY-group variables are replicated across all computing nodes; each worker sends the data rows that it receives from the DFS to all

other nodes. Subsequently, these replicated transactional data are added to every BY group during the second step, the accumulation step. Figure 6 illustrates the auto-partitioning step for a single table, Table A.

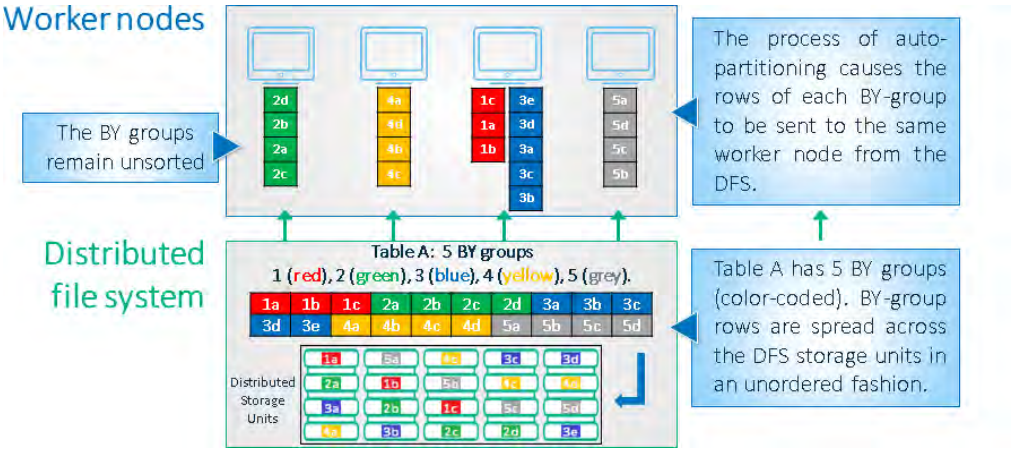

Figure 6. BY-Group "Auto-Partitioning" during Parallel Read

In Figure 6, Table A contains five BY groups that are color coded: elements of BY-group 1 in red, BY-group 2 in green, BY-group 3 in blue, BY-group 4 in yellow, and BY-group 5 in gray. The elements of each BY group (for example, 1a, 1b, and 1c) correspond to table rows that contain timestamped transactional data. The BY-group rows are spread in an unordered fashion across the different storage units of the DFS. Each BY-group is read from the DFS, shuffled, and assigned to contiguous memory on a single computing (worker) node. At the end of the auto-partitioning step, the computing nodes have approximately equal numbers of time series (BY groups) stored in memory awaiting execution. However, these time series remain unsorted and have no fixed interval; they are not yet suitable for analysis.

2. Accumulation occurs immediately prior to the execution of each BY group. This step involves taking the in-memory transactional data of each BY group that resulted from the previous step (auto-partitioning) and accumulating them into sorted, fixed-interval, and fixed-length time series that are suitable for analysis. In the TSMODEL procedure, the time span of the main data set, which you specify in the DATA= option, is used to determine the fixed length of all accumulated BY groups. Alternatively, you can specify the lower bound and upper bound (or both) of the accumulation time span in the START= and END= options, respectively, in the ID statement. In addition, the transaction data from the replicated input data files that contained none of the BY-group variables are accumulated to each BY group during this step. The resulting time series are supplied as array variables to the compiled script for processing by program statements. Figure 7 illustrates the accumulation step for the input Table A that is depicted in Figure 6. The worker nodes accumulate the five BY groups of Table A to fixed-length time series prior to executing the compiled script.

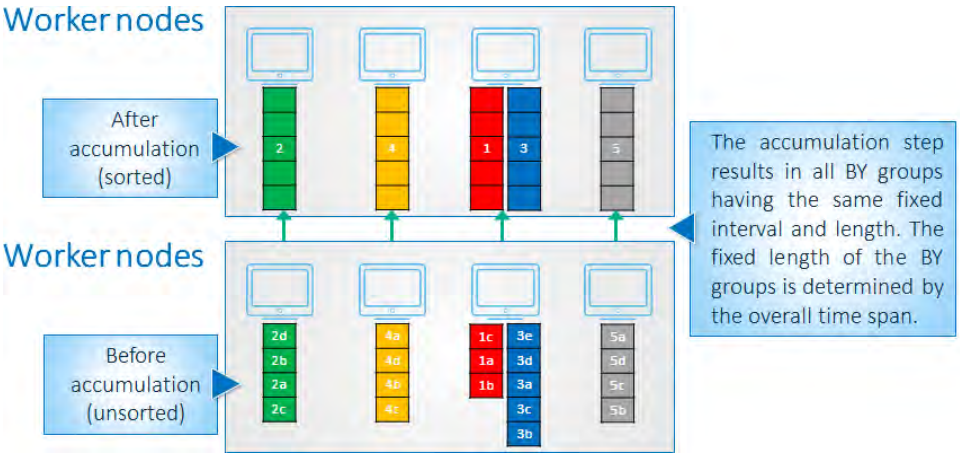

Figure 7. BY-Group "Accumulation" during Parallel Read

When you do not specify BY-group variables in a BY statement, the TSMODEL procedure assumes that all input data files that you specify in the DATA= and AUXDATA= options correspond to a single overarching time series. When compared to the previous case, this assumption is equivalent to working with a single BY group (although no BY group variables are actually involved). Consequently, data rows from all input files must be collected into the contiguous memory of a single worker node prior to the analysis. This scenario poses a potential network transport bottleneck because a single worker node acts as the sole recipient of the auto-partitioning (shuffling) step. In contrast, all worker nodes in the previous case evenly shared the burden of distributing the input data during the shuffling process. In this current scenario, the TSMODEL procedure minimizes the burden on the destination node by compressing the time series data prior to transport. This compression is achieved via an additional accumulation step that is performed by the worker nodes prior to sending their data to the destination node.

As in the previous case, the process starts with the worker nodes reading in parallel all the rows of an input data file from the DFS. Worker nodes end up with chunks of the single time series (partial time series) rather than the complete time series in memory. Because the DFS provides data redundancy, duplicate data rows appear in the partial time series of multiple worker nodes. These extraneous data simply increase the network transport time. Next, and in contrast to the previous case, worker nodes then perform an accumulation step on their partial time series prior to sending their data to the single destination node (shuffling). This accumulation step tends to significantly reduce the cardinality of the partial time series (such as when data that are collected every second or minute are accumulated into weeks). In other words, the smaller accumulated (compressed) partial time series can be more quickly transported to the single destination node. After the destination node receives the accumulated partial time series from all other worker nodes, it merges and accumulates them into a complete time series that is ready for analysis by the compiled script. At the end of this process, the distributed transactional data that was stored across the DFS has been efficiently migrated via the network to the contiguous memory of a single computing node.

PARALLEL AND THREADED EXECUTION

Each computing node executes (in parallel) the compiled, optimized script for each time series that has been assigned to it. Each time series is executed on one thread of the computing node. Each of the computing node’s threads is kept busy until all the time series that have been assigned to it have been processed. If any problems occur during the execution of a particular time series (BY group), they are logged into an in-memory table so that you can investigate them further.

Figure 8 illustrates the parallel execution. Each time series is executed independently on a computing (worker) node.

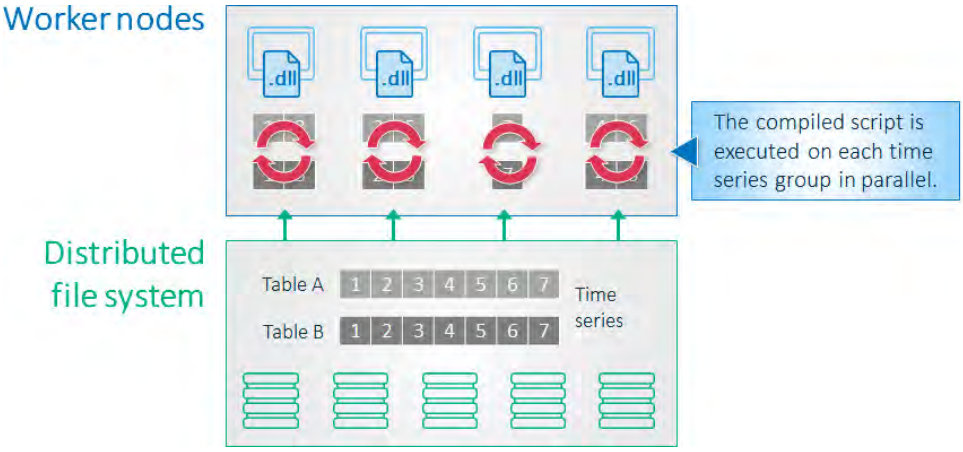

Figure 8. Parallel Execution

Optimization

The time series modeling and forecasting capabilities of the TSMODEL procedure have been optimized in order to ensure that parallel execution remains efficient and scalable with increased thread usage. The optimization helps ensure a graceful degradation of service when concurrent sessions of the TSMODEL procedure are spawned in a shared computing server. A single session can tax the server's hardware resources. The optimization focuses primarily on efficiently reusing allocated memory in order to minimize the load that is placed by increased thread usage on the computing server's virtual memory (VM) subsystem. The VM subsystem requests (allocates) or returns (deallocates) memory in a serial fashion, one thread at a time, in order to prevent a race condition from corrupting its internal data structures. Both of these actions are commonplace in any program. Consequently, in multithreaded architectures, threads can stall for a significant amount of time as they wait in the queue for their turn to interact with the VM subsystem. Increased thread usage can severely exacerbate this problem. For optimal performance, it is crucial to minimize the amount of time that is wasted by threads being stalled.

Memory Management

The TSMODEL procedure has an efficient memory management strategy that minimizes the number of VM subsystem queries that are performed by the BY-group threads during the parallel execution step. As a result, thread stalls are prevented, and more time is spent performing useful time series modeling and forecasting computations. Furthermore, this strategy also enables a graceful degradation of service when multiple PROC TSMODEL sessions are run in parallel on a shared computing server. The following memory management strategy performs the costly memory allocations once and reuses the memory across BY groups:

1. At the start of a BY-group processing step, BY-group threads allocate large chunks of memory, called "memory containers," from the VM subsystem. This momentarily creates contention on the VM subsystem.
2. Each BY-group thread then subdivides its containers into smaller blocks of memory in order to instantiate both the required data objects (time series arrays) and the computational objects (time series models). As the execution of the compiled script progresses, each thread requests more memory containers on-demand from the VM subsystem as their local resources are exhausted.

3. As a BY-group thread finishes processing a BY group, it does not give back its memory containers to the VM subsystem. Instead, the BY-group thread retains its containers and reuses them first to process subsequent BY groups, requesting another container only when needed. This results in a significantly reduced number of memory allocation queries to the VM subsystem. To accomplish this, the portions of the memory containers that hold the computational objects are left intact, enabling the reuse of time series modeling and forecasting objects across BY groups. Critical portions of the computational objects are reset and made ready to build new time series models. This effectively eliminates the need to perform costly instantiations of computational objects at the start of every BY-group process. In contrast, the portions of the memory containers that hold data objects are released to hold the subsequent BY-group data.
4. After all BY groups have been processed, all BY-group threads then give back their memory containers to the VM subsystem, which momentarily causes some contention.

This memory management strategy reduces the BY-group threads' contention on the VM subsystem to only the amount that is generated by the execution of the first few BY groups, when the majority of queries for memory containers occur. The contention for interaction with the VM subsystem is greatly diminished after that. Also, in the worst-case scenario, each BY-group thread owns at any time just enough memory to process its largest BY group. This strategy enables the time series modeling and forecasting capabilities of the TSMODEL procedure to scale extremely well with increased thread usage.

PARALLEL WRITE TO THE DISTRIBUTED FILE SYSTEM

After the specified analysis is executed for a particular time series, the computing nodes write one or more output data sets asynchronously and independently. Multiple output data files can be created simultaneously.

Figure 9 illustrates the parallel write. Each time series analysis result is written back to the distributed file system.

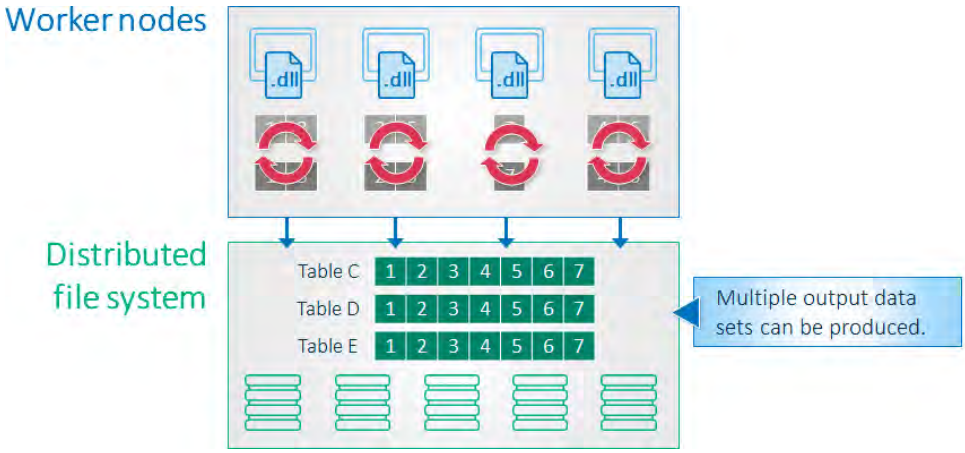

Figure 9. Parallel Write

IMPLEMENTATION

SAS Visual Forecasting enables you to use a variety of methods to implement solutions to your time series forecasting problems. You can use procedures, scripts, packages, and actions.

THE TSMODEL PROCEDURE

The TSMODEL procedure is a SAS® Viya® procedure that executes user-defined programs (scripts) on time series data. The TSMODEL procedure analyzes timestamped transactional data with respect to time and accumulates the data into a time series format.

The TSMODEL procedure forms time series from timestamped, transactional input data and writes the accumulated time series variables to an output table. Time series are delineated by distinct values of the variables that are specified in the BY statement.

Timestamped transactional data are not usually recorded at a fixed interval. Because analysts often want to use time series analysis techniques that require fixed-time intervals, the transactional data must be transformed into a fixed-interval time series, such as daily, weekly, or monthly.

The TSMODEL procedure forms time series vectors from timestamped data and then provides these vectors as array variables for subsequent processing by program statements (a script). The script is processed independently for each BY group. The syntax of the TSMODEL procedure is the same as that of the TIMEDATA procedure, which is similar to the SAS DATA step for time series data. The SAS DATA step processes data row by row, whereas the TSMODEL procedure processes time series vectors (columns) for the BY groups.

For more information about the TSMODEL procedure, see *SAS Visual Forecasting 8.2: Forecasting Procedures*.

SCRIPTS

Scripts consist of actions, which perform the desired analysis on each time series. For more information about scripts, see the FCMP procedure in *Base SAS® Procedures Guide*.

PACKAGES

Packages contain computational services that can be used in your script. A package is a set of related specialized objects and functions that tackles a unique facet of the time series analysis problem. You can use specialized objects and functions to write custom SAS code in order to gain access both to cutting-edge data analysis tools and to utilities that are designed to significantly speed up code development and improve the quality of the resulting code. The following packages are available for the TSMODEL procedure:

- SFS (simple forecast service)—Tools for automatic forecasting of time series with a simple-to-use interface; these tools use only exponential smoothing (ESM) and ARIMA models
- ATSM (automatic time series modeling and forecasting) —Tools for automatic modeling and forecasting of time series by using various model families such as exponential smoothing (ESM), ARIMA, intermittent demand (IDM), and unobserved component (UCM) models
- TSA (time series analysis) —Tools for efficient statistical analysis of time series (transformations, decompositions, statistical tests for intermittency, seasonality, stationarity, forecast bias, and so on)
- TFA (time-frequency analysis) —Tools for efficient analysis of time series in both time domain and frequency domain
- TSM (time series modeling) —Tools for efficient time series modeling and forecasting
- SSA (singular spectrum analysis) —Tools for decomposing a time series into additive components and categorizing those components based on the magnitudes of their contributions
- MTF (time series motif discovery) —Tools for the discovery of frequent patterns or repeated subsequences in time series
- UTL (utility) —Tools for performing basic statistical computations on pairs of actual and predicted time series

Some of these packages were developed as cloud-based analogues of traditional SAS products and procedures. For example, the ATSM, SFS, and TSM packages carry the features available in SAS® Forecast Server. Similarly, the TSA, TFA, and SSA packages carry various features that are available in SAS/ETS®. For more information about these packages, see *SAS Visual Forecasting 8.2: Time Series Packages*.

ACTIONS

Actions are executed on the computing server and can be invoked by using a variety of languages: SAS, Python, Lua, and R. For more information about actions, see *SAS Visual Forecasting 8.2: Programming Guide*.

EXAMPLES

This section provides two examples that demonstrate the capabilities of the TSMODEL procedure. The first example illustrates how you can use the procedure to perform automatic time series forecasting. The second example shows the procedure's ability to perform fast time series analysis on big data.

AUTOMATIC TIME SERIES FORECASTING

This example demonstrates how to use the TSMODEL procedure to perform automatic forecast model generation, automatic variable and event selection, and automatic model selection. This example uses the objects in the ATSM package to automatically model the Sashelp.PriceData data set, which provides simulated monthly sales data that are hierarchically organized by region, line, and product. The Sashelp.PriceData data set contains 1,020 observations, which are divided into 17 BY groups. Although this data set is relatively small, the sample code provided in this section can also readily handle time series data sets that contain millions of BY groups.

The first step of the analysis is to appropriately configure the DIAGSPEC and DIAGNOSE objects. Combined, the DIAGSPEC and DIAGNOSE objects are the cloud-based analogue of the HPFDIAGNOSE procedure from SAS® High-Performance Forecasting. You use the DIAGSPEC object to specify model diagnostic control options for the DIAGNOSE object. For example, you can specify which model families should be diagnosed (ESM, ARIMA, IDM, and UCM). After that, the FORENG object is used to produce the analysis results, which are based on the model diagnostic choices that the DIAGNOSE object makes. Various results are then collected from the FORENG object and saved into output tables. The program comments provide a detailed description of these execution steps (for clarity, all SAS keywords, commands, package names, object method names, and object method parameters that are immutable are presented in upper case letters in the program).

```
/*
 * A connection to the CAS server (that is, a session) is established, and
 * a CAS library called 'mycas' is created. The 'mycas' library enables you
 * to transfer data sets to the CAS server where the distributed time
 * series analysis is performed.
 */

CAS mycas;
LIBNAME mycas CAS SESSREF = mycas;

/*
 * A DATA step transfers the Sashelp.PriceData into the CAS 'mycas'
 * library.
 */

DATA mycas.pricedata;
  SET sashelp.pricedata;
RUN;

/*
 * The PROC TSMODEL statement specifies the input data set
 * ('mycas.pricedata') and a variety of output tables ('mycas.priceFor',
 * 'mycas.priceEst', and so on). It also specifies the output table where
 * the execution logs are to be stored ('mycas.priceLog').
 */
```

```
PROC TSMODEL DATA=mycas.pricedata OUTLOG=mycas.priceLog
  OUTOBJ=(priceFor=mycas.priceFor priceEst=mycas.priceEst
  modInfo=mycas.modInfo priceSelect=mycas.priceSelect);

  /*
   * The ID statement specifies the variable 'date' as the time index
   * variable, and the INTERVAL= option indicates that the data are
   * monthly.
   */

  ID date INTERVAL = MONTH;

  /*
   * The BY statement specifies that each unique combination of the data
   * set variables 'regionname', 'productline', and 'productname'
   * correspond to a unique time series BY group. BY groups are processed
   * independently.
   */

  BY regionname productline productname;

  /*
   * The first VAR statement specifies the input data set variable 'sale'.
   * This is the dependent series (also called the Y series) to be modeled.
   * The ACC=SUM option requests a total sum accumulation for the 'sale'
   * variable.
   */

  VAR sale / ACC = SUM;

  /*
   * The second VAR statement specifies the input data set variable
   * 'price'. This is an independent series (also called an X series). The
   * ACC=AVG option requests an average value accumulation for the 'price'
   * variable.
   */

  VAR price / ACC = AVG;

  /*
   * The REQUIRE statement specifies the ATSM package, which is needed for
   * the analysis.
   */

  REQUIRE ATSM;

  /*
   * The PRINT OUTLOG statement causes the contents of the OUTLOG= table to
   * be automatically displayed on the client. Printing this table is
   * useful if you need to debug your program. This statement is valid only
   * when the OUTLOG= option is specified.
   */

  PRINT OUTLOG;

  /*
```

```
 * The program statements between the SUBMIT and ENDSUBMIT statements use
 * the ATSM package objects to perform the actual analysis on the CAS
 * server.
 */

SUBMIT;

  /*
   * The first part of the program creates a variable 'dataFrame', which
   * is a TSDF object that contains the necessary analysis variables. The
   * variable roles (target or input) and the default season length that
   * is associated with the data frame are also assigned in this step. In
   * addition, the predefined events 'Easter' and 'Christmas' are added
   * to the data frame. When an event is added to the data frame, a
   * corresponding dummy variable is automatically created for use during
   * the event selection process.
   */

  DECLARE OBJECT eventDB(EVENT);
  rc = eventDB.Initialize();

  DECLARE OBJECT dataFrame(TSDF);
  rc = dataFrame.INITIALIZE();
  rc = dataFrame.ADDY(sale);
  rc = dataFrame.ADDX(price);
  rc = dataFrame.SETOPTION('SEASONALITY', 12);
  rc = dataFrame.ADDEVENT(eventDB, 'CHRISTMAS', 'REQUIRED', 'YES');
  rc = dataFrame.ADDEVENT(eventDB, 'EASTER', 'REQUIRED', 'NO');

  /*
   * The second part of the program configures the model identification
   * process in two steps. First a DIAGSPEC object, 'priceDiagSpec', is
   * configured. Next, it is used to initialize a DIAGNOSE object,
   * 'priceDiag'. The DIAGSPEC object in this example uses the default
   * settings, which amounts to selecting the best-fitting model from two
   * model families: exponential smoothing models (ESMs) and ARIMAX
   * models. As a result of setting the HOLDOUT parameter for the
   * DIAGNOSE object to 12, the best-fitting model is chosen within each
   * family on the basis of the root mean square error (RMSE) criterion
   * (the default CRITERION choice) in the holdout region (the last 12
   * observations).
   */

  DECLARE OBJECT priceDiagSpec(DIAGSPEC);
  rc = priceDiagSpec.OPEN();
  rc = priceDiagSpec.SETESM();
  rc = priceDiagSpec.SETARIMAX();
  rc = priceDiagSpec.CLOSE();
```

```
    DECLARE OBJECT priceDiag(DIAGNOSE);
    rc = priceDiag.INITIALIZE(dataFrame);
    rc = priceDiag.SETSPEC(priceDiagSpec);
    rc = priceDiag.SETOPTION('HOLDOUT', 12);
    rc = priceDiag.RUN();

    /*
     * The third part of the program uses a FORENG object, 'priceEng', to
     * select the final model based on the DIAGNOSE object results and to
     * produce forecasts. This FORENG object is initialized by using the
     * 'priceDiag' object that is created in the second part of the
     * program.
     */

    DECLARE OBJECT priceEng(FORENG);
    rc = priceEng.INITIALIZE(priceDiag);
    rc = priceEng.SETOPTION('LEAD', 12);
    rc = priceEng.RUN();

    /*
     * The following IF statement requests that the script execution stop
     * if the FORENG object fails to produce a forecast. You should always
     * check the return code of every function and object method call.
     * Negative return codes generally indicate that an error has occurred.
     * The STOP statement ceases the execution of the script for the
     * current BY group.
     */

    IF (rc < 0) then STOP;

    /*
     * The last part of the program uses collector objects of appropriate
     * type to create various output tables. These output tables carry
     * parameter estimates, model forecast information, and model selection
     * information for all of the BY groups.
     */

    DECLARE OBJECT modInfo(OUTMODELINFO);
    rc = modInfo.COLLECT(priceEng);

    DECLARE OBJECT priceFor(OUTFOR);
    rc = priceFor.COLLECT(priceEng);

    DECLARE OBJECT priceEst(OUTEST);
    rc = priceEst.COLLECT(priceEng);

    DECLARE OBJECT priceSelect(OUTSELECT);
    rc = priceSelect.COLLECT(priceEng);

  ENDSUBMIT;
RUN;
```

The TSMODEL procedure prints a summary of the time series processing that is performed, as shown in Output 1. This summary includes the number of BY groups that are processed, the total processing time, and some information about the accumulation process.

Summary of time series processing for PRICEDATA	
Number of analysis variables	2
Number of rows read	2040
Number of groups read	17
Memory for group packages (KB)	2
Time to load groups (seconds)	0.0467860699
Minimum time ID	JAN1998
Maximum time ID	DEC2002
Minimum time periods	60
Maximum time periods	60
Number of nodes run	1
Number of nodes with data	1
Number of nodes with groups	1
Number of threads budgeted	32
Minimum thread group count	0
Maximum thread group count	2
Minimum threads active	32
Maximum threads active	32
Number of groups processed by submitted code	17
Number of groups failing	0
Elapsed time to process groups (seconds)	0.1150372028

Output 1. Summary of Time Series Processing for Sashelp.PriceData

Shown next are the time series analysis results for the first BY group in the Sashelp.PriceData data set. Output 2 shows the results of the final model selection step, which were collected into the mycas.priceSelect output table by the OUTSELECT collector object and are printed by the following PRINT procedure step:

```
PROC PRINT DATA=mycas.priceSelect NOOBS;
  VAR _MODEL_ _SELECTED_ _LABEL_;
  WHERE regionname='Region1' & productline='Line1' & productname='Product1';
RUN;
```

MODEL	_SELECTED_	_LABEL_
ARIMAX	YES	ARIMA: sale ~ P = 1 D = (1,12) Q = (12) NOINT + INPUT: Dif(1,12) price
ESM	NO	Linear Exponential Smoothing

Output 2: Choice between the ESM and ARIMAX Models

Output 3 shows the parameter estimates of the selected ARIMAX model, which were collected into the mycas.priceEst output table by the OUTEST collector object and are printed by the following PRINT procedure step:

```
PROC PRINT DATA=mycas.priceEst NOOBS;
  VAR _COMPONENT_ _EST_ _STDERR_ _TVALUE_ _PVALUE_;
  WHERE regionname='Region1' & productline='Line1' & productname='Product1';
RUN;
```

COMPONENT	_EST_	_STDERR_	_TVALUE_	_PVALUE_
MA	0.6819	0.1535	4.4420	0.000059
AR	-0.2734	0.1486	-1.8391	0.0727
SCALE	-25.0464	1.2269	-20.4147	4.55E-24

Output 3. Parameter Estimates for the Selected ARIMAX Model

Output 4 shows the forecasts according to the selected ARIMAX model, which were collected into the mycas.priceFor table by the OUTFOR collector object and are printed by the following PRINT procedure step. The 'predict' variable corresponds to the forecast series, 'std' corresponds to the standard error, and 'upper' and 'lower' correspond to the bounds of the confidence interval:

```
PROC PRINT DATA=mycas.priceFor NOOBS;
  FORMAT date DATE.;
  VAR date predict std upper lower;
  WHERE regionname='Region1' & productline='Line1' & productname='Product1'
        & YEAR(date)>=2003;
RUN;
```

date	PREDICT	STD	UPPER	LOWER
01JAN03	391.8	25.6567	442.1	341.5
01FEB03	409.7	31.7148	471.8	347.5
01MAR03	406.9	37.7963	481.0	332.9
01APR03	409.1	42.7786	493.0	325.3
01MAY03	411.6	47.2992	504.3	318.9
01JUN03	430.5	51.4087	531.3	329.7
01JUL03	420.9	55.2170	529.1	312.6
01AUG03	424.8	58.7781	540.0	309.6
01SEP03	400.2	62.1357	521.9	278.4
01OCT03	382.7	65.3208	510.7	254.7
01NOV03	378.4	68.3577	512.3	244.4
01DEC03	391.7	71.2654	531.4	252.0

Output 4: Forecasts Based on the Selected ARIMAX Model

LIGHTNING-FAST BIG DATA ANALYSIS

This example demonstrates the TSMODEL procedure's ability to perform fast time series analysis on big data. The objects in the ATSM package are used to perform automatic forecasts of product demand on a large industrial data set. The input data file, 89 GB in size, contains 375.11 million observations, which are divided into 1.56 million BY groups in a hierarchy of store location, product, and customer. On average, each BY group contains 4.3 years of weekly historical data that are available for analysis. The data set offers a total of 15 independent variables as input to the potential ARIMA models that are automatically generated during the automatic model diagnostics stage. This exercise was conducted using 144 worker nodes and 32 BY-group threads per node. Worker nodes contain Intel Xeon processors with 32 cores running at 2.7 GHz and 252 GB of RAM.

The first step of the analysis is to appropriately configure the DIAGSPEC and DIAGNOSE objects. After that, the FORENG object is used to produce the analysis results that are based on the model diagnostic choices that the DIAGNOSE object made. Various results are then collected from the FORENG object and saved into output tables. Comments in the program provide a detailed description of these execution

steps. For clarity, all SAS keywords, commands, package names, object method names, and object method parameters that are immutable are presented in upper case letters in the program:

```
/*
 * A connection to the CAS server (a session) is established and a CAS
 * library called 'mycas' is created. The 'mycas' library enables you
 * to transfer data sets to the CAS server where the distributed time
 * series analysis is performed.
 */

CAS mycas;
LIBNAME mycas CAS SESSREF = mycas;

/*
 * The CAS procedure loads the 89 GB input file called 'bigdata.sashdat'
 * into the CAS 'mycas' library as the data set 'mycas.bigdata'. The 375.11
 * million observations that make up the 1.56 million BY groups are
 * randomly distributed across the 144 worker nodes.
 */

PROC CAS;
  TABLE.LOADTABLE /
    CASOUT={CASLIB="CASUSERHDFS(&CASUSER)",NAME="bigdata"}
    CASLIB="BigDataCASLib"
    PATH="bigdata.sashdat";
  RUN;
QUIT;

/*
 * The PROC TSMODEL statement specifies the input data set ('mycas.bigdata)
 * and a variety of output tables ('mycas.outFor', 'mycas.outEst', and so
 * on). It also specifies the output table where the execution logs will be
 * stored ('mycas.outLog').
 */

PROC TSMODEL DATA=mycas.bigdata OUTLOG=mycas.outLog
  OUTOBJ=(outset = mycas.outEst
          outfor = mycas.outFor
          outstat = mycas.outStat
          outfmsg = mycas.outFmsg
          outselect = mycas.outSelect
          outmodelinfo = mycas.outModelinfo
          );

  /*
   * The ID statement specifies the variable 'date' as the time index
   * variable, and the INTERVAL= option indicates that the data are weekly.
   */

  ID date INTERVAL = WEEK;

  /*
   * The BY statement specifies that each unique combination of the data
   * set variables 'location, 'product', and 'customer' correspond to a
   * unique time series BY group. BY groups are processed independently.
   */
```

```
BY location product customer;

/*
 * The first VAR statement specifies the input data set variable
 * 'demand_qty'. This is the dependent series (Y series) to be modeled
 * The ACC=SUM option requests a total sum accumulation for this
 * variable.
 */

VAR demand_qty / ACC = TOTAL;

/*
 * The 15 subsequent VAR statements specify a total of 15 data set
 * variables, 'ind1' through 'ind15'. These are the independent series (X
 * series). The ACC=AVERAGE option requests an average value accumulation
 * for the independent variables. Alternatively, you can use a single VAR
 * statement as follows:
 *                  VAR ind1-ind15 / ACC = AVERAGE;
 */

VAR ind1 / ACC = AVERAGE;
VAR ind2 / ACC = AVERAGE;
VAR ind3 / ACC = AVERAGE;
VAR ind4 / ACC = AVERAGE;
VAR ind5 / ACC = AVERAGE;
VAR ind6 / ACC = AVERAGE;
VAR ind7 / ACC = AVERAGE;
VAR ind8 / ACC = AVERAGE;
VAR ind9 / ACC = AVERAGE;
VAR ind10 / ACC = AVERAGE;
VAR ind11 / ACC = AVERAGE;
VAR ind12 / ACC = AVERAGE;
VAR ind13 / ACC = AVERAGE;
VAR ind14 / ACC = AVERAGE;
VAR ind15 / ACC = AVERAGE;

/*
 * The REQUIRE statement specifies the ATSM package, which is needed for
 * the analysis.
 */

REQUIRE ATSM;

/*
 * The PRINT OUTLOG statement causes the contents of the OUTLOG= table to
 * be automatically displayed on the client. Printing this table is
 * useful if you need to debug your program. This statement is valid only
 * when the OUTLOG= option is specified.
 */

PRINT OUTLOG;

/*
 * The program statements between the SUBMIT and ENDSUBMIT statements use
 * the ATSM package objects to perform the actual analysis on the CAS
 * server.
 */
```

```
SUBMIT;

  /*
   * The first part of the program creates the variable 'dataFrame',
   * which is a TSDF object that contains the necessary analysis
   * variables. The variable roles (target or input) are assigned in this
   * step.
   */

  DECLARE OBJECT dataFrame(TSDF);
  rc = dataFrame.INITIALIZE();
  rc = dataFrame.ADDY(demand_qty);
  rc = dataFrame.ADDX(ind1);
  rc = dataFrame.ADDX(ind2);
  rc = dataFrame.ADDX(ind3);
  rc = dataFrame.ADDX(ind4);
  rc = dataFrame.ADDX(ind5);
  rc = dataFrame.ADDX(ind6);
  rc = dataFrame.ADDX(ind7);
  rc = dataFrame.ADDX(ind8);
  rc = dataFrame.ADDX(ind9);
  rc = dataFrame.ADDX(ind10);
  rc = dataFrame.ADDX(ind11);
  rc = dataFrame.ADDX(ind12);
  rc = dataFrame.ADDX(ind13);
  rc = dataFrame.ADDX(ind14);
  rc = dataFrame.ADDX(ind15);

  /*
   * The second part of the program configures the model identification
   * process in two steps. First a DIAGSPEC object, 'diagSpec', is
   * configured. Then it is used to initialize a DIAGNOSE object,
   * 'diagnose'. The DIAGSPEC object in this example specifies the
   * selection of the best fitting model from among three model families:
   * intermittent demand (IDM), exponential smoothing (ESM), and ARIMAX
   * models.
   */

  DECLARE OBJECT diagSpec(DIAGSPEC);
  rc = diagSpec.OPEN();
  rc = diagSpec.SETIDM('INTERMITTENT', 2);
  rc = diagSpec.SETESM('METHOD', 'BEST');
  rc = diagSpec.SETARIMAX('IDENTIFY', 'BOTH');
  rc = diagSpec.CLOSE();

  DECLARE OBJECT diagnose(DIAGNOSE);
  rc = diagnose.INITIALIZE(dataFrame);
  rc = diagnose.SETSPEC(diagSpec);
  rc = diagnose.RUN();

  /*
   * The third part of the program uses a FORENG object, 'forecast', to
   * select the final model based on the DIAGNOSE object results and to
   * produce forecasts. This FORENG object is initialized by using the
   * 'diagnose' object that is created in the second part of the program.
   */
```

```
      DECLARE OBJECT forecast(FORENG);
      rc = forecast.INITIALIZE(diagnose);
      rc = forecast.SETOPTION('LEAD',12);
      rc = forecast.SETOPTION('CRITERION','RMSE');
      rc = forecast.SETOPTION('HOLDOUT',2);
      rc = forecast.RUN();

      /*
       * The following IF statement requests that the script execution stop
       * if the FORENG object fails to produce a forecast. You should always
       * check the return code of every function and object method call.
       * Negative return codes generally indicate that an error has occurred.
       * The STOP statement ceases the execution of the script for the
       * current BY group.
       */

      IF (rc < 0) then STOP;

      /*
       * The last part of the program uses collector objects of appropriate
       * type to create various output tables. These output tables carry
       * parameter estimates, model forecast information, and model selection
       * information for all the BY groups.
       */

      DECLARE OBJECT outest(OUTEST);
      rc = outest.COLLECT(forecast);

      DECLARE OBJECT outfor(OUTFOR);
      rc = outfor.COLLECT(forecast);

      DECLARE OBJECT outstat(OUTSTAT);
      rc = outstat.COLLECT(forecast);

      DECLARE OBJECT outfmsg(OUTFMSG);
      rc = outfmsg.COLLECT(forecast);

      DECLARE OBJECT outselect(OUTSELECT);
      rc = outselect.COLLECT(forecast);

      DECLARE OBJECT outmodelinfo(OUTMODELINFO);
      rc = outmodelinfo.COLLECT(forecast);

    ENDSUBMIT;
  RUN;
```

The TSMODEL procedure prints a summary of the time series processing that is performed, as shown in Output 5.

Summary of time series processing for BIGDATA	
Number of analysis variables	16
Number of rows read	750222336
Number of groups read	1562593
Memory for group packages (KB)	231948
Time to load groups (seconds)	49.514738083
Minimum time ID	Sun, 27 Dec 2009
Maximum time ID	Sun, 30 Mar 2014
Minimum time periods	223
Maximum time periods	223
Number of nodes run	145
Number of nodes with data	144
Number of nodes with groups	144
Number of threads budgeted	72
Minimum thread group count	272
Maximum thread group count	406
Minimum threads active	32
Maximum threads active	32
Number of groups processed by submitted code	1562593
Number of groups failing	0
Elapsed time to process groups (seconds)	59.908494949

Output 5: Summary of Time Series Processing for the Large Industrial Data Set

The last row of the table in Output 5 reports that PROC TSMODEL required only 59.91 seconds to process all of the 1.56 million BY groups. This elapsed time is the sum of two quantities: 49.51 seconds (reported in the fourth row), which were spent reading and shuffling the input data set (see the section "Parallel Read" for a detailed description of this process), and the remaining 10.39 seconds, which were spent performing the actual automatic forecasts for all 1.56 million BY groups. These results attest to PROC TSMODEL's scalability and efficiency when it comes to solving time series forecasting problems for big data.

CONCLUSION

The TSMODEL procedure enables both scalable and optimized time series analysis in a cloud environment. PROC TSMODEL comes equipped with a generic scripting environment, which enables you to develop custom time series analysis algorithms and prepare your data for analysis (clean, transform, preprocess, and postprocess), all within the same script. This contributes to reduced data movement (because the data remains in the same contiguous memory throughout the analysis) and also optimizes code development. PROC TSMODEL also comes equipped with various specialized time series analysis packages that provide advanced support for time series analysis (in the time domain or in the frequency domain), time series decomposition, time series modeling, signal analysis and anomaly detection (for IoT), and temporal data mining. Because of these features, what can be accomplished by PROC TSMODEL is limited only by your imagination. As is illustrated by the second example, PROC TSMODEL's distributed nature (BY groups are processed in parallel), efficiency and scalability (disk IO is minimized, all data operations are performed in memory, allocated memory is efficiently reused across BY groups), and optimized time series modeling and forecasting capabilities enables big data forecasting problems to be solved with unprecedented speeds. For example, it is estimated that the run time of the analysis shown in the second example would increase from 1 minute to 76 hours if executed on a single processor. In summary, TSMODEL can efficiently perform time series analysis of big data in close to real time.

REFERENCES

Leonard, Michael. 2002. “Large-Scale Automatic Forecasting: Millions of Forecasts.” *Presented at the 2002 International Symposium of Forecasting*. Dublin, Ireland.

Leonard, Michael. 2004. “Large-Scale Automatic Forecasting with Calendar Events and Inputs.” *Presented at the 2004 International Symposium of Forecasting*. Sydney, Australia.

Leonard, Michael, and Bruce Elsheimer. 2015. “Count Series Forecasting.” *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc.
Available http://support.sas.com/resources/papers/proceedings15/SAS1754-2015.pdf

ACKNOWLDEGMENTS

The authors would like to thank Anne Baxter from SAS for her editing.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Thiago Quirino
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-3721
Thiago.Quirino@sas.com

Michael Leonard
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
919-531-6967
Michael.Leonard@sas.com

Ed Blair
EBlair2@twc.com