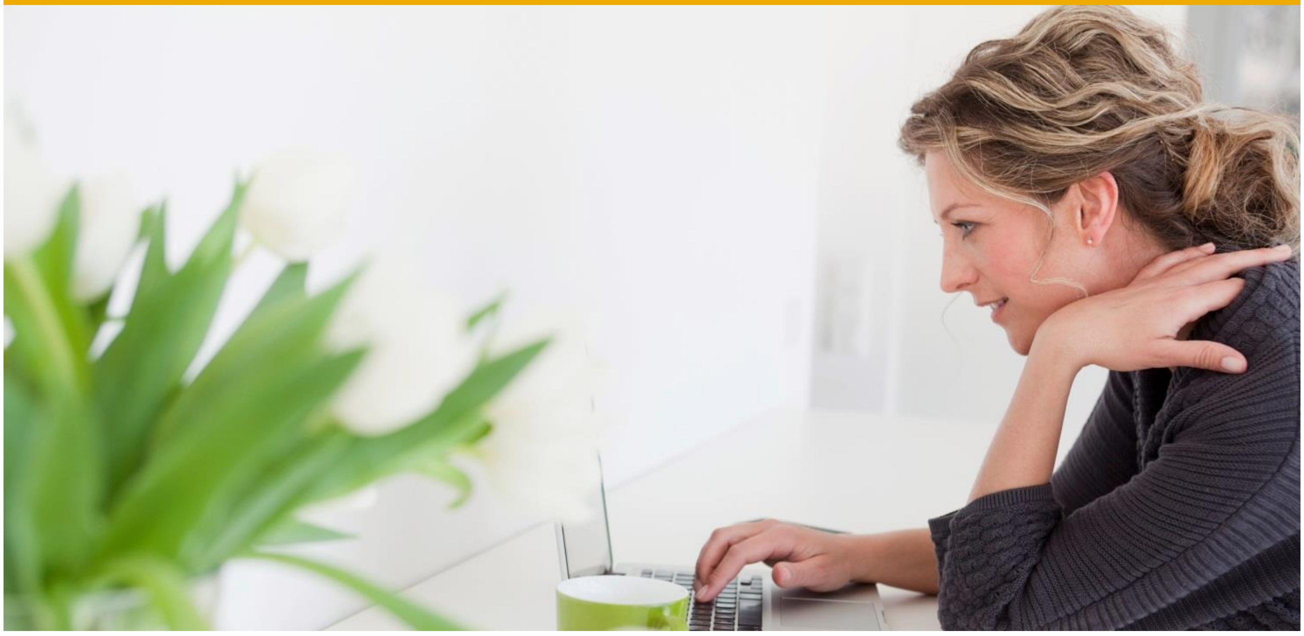


SAP HANA Smart Data Streaming Sizing and Configuration Guidelines

document version 1.0

Applies to SAP HANA smart data streaming version 1.0 SPS09

TABLE OF CONTENTS

1 INTRODUCTION

A very natural question to ask when planning a streaming application that will run on an SAP HANA smart data streaming server (SDS) is:

What should the machine size and configuration be to run our streaming project(s)?

The short answer to this question is it depends on the project. Streaming projects vary widely in the number of streams/windows it has and in the logic complexity required to meet the business need.

The size and configuration of the machine depends on several factors:

- Number of Streams/Windows in the critical path of the project.
- Number of complex operations in the critical path. Especially Aggregations, Joins and Flex operations.
- Latency and throughput requirements.
- Size of each record and number of records in various windows and data structures.
- The number of publishers and subscribers and adapters attached to the project.

Typical streaming projects, which process large volumes of fast flowing data without the need to store large amounts of data in memory, benefit from having a large number of fast CPUs and fast networks rather than large amounts of memory and disk space.

This document is a guide to estimating approximately the machine configuration required to run your streaming project(s). It provides information about:

- Quickly sizing a machine for a streaming project.
- Recommended usage for machines of various sizes
- Detailed estimation of memory and CPU's requirements for a project.
- Disk size requirements.
- Networking requirements.

2 DISCLAIMERS

The formulae used to determine memory consumption tells you the minimum memory needed by the server to store the necessary state (in windows, CCL SCRIPT data structures, etc.) to process your model. The document describes only the significant memory allocations done by the server. You should add another 20% of the computed value for other memory allocations.

In addition to this memory the server requires significant amounts of transient memory to process records, handle subscriber requests, execute ad-hoc queries and temporarily store records in intermediate queues. There are too many variables that are application specific to determine the amount of transient memory required, so this document only mentions at a high level the things to look for to determine transient memory requirements.

Similarly, determining the number of processors is only a course approximation and is heavily dependent on the complexity of the logic in each stream, the number of choke points in the model, the number and nature of subscribers etc. It is highly recommended that at least a 20% margin of error be added to the computed number of processors.

The resources computed using this document is only a minimum requirement. It is highly advisable to add a margin of safety for any future changes to the model or to changes in the dynamic uses of the application such as subscribers, publishers, ad-hoc queries etc.

The actual resources used can change from release to release depending upon the changes to the compiler, server and libraries used by SDS or other reasons. The document is based on SDS version 1.0 SPS09.

3 TERMINOLOGY

We use the term “processor” in this paper to refer to an independent processing unit for instructions. This is sometimes referred to as a “core”. There is no distinction between a physical processor and cores on a chip.

The notation MB used in this document does not represent the standard value of 1,048,576 bytes but rather 1,000,000 bytes.

4 RESTRICTIONS, REQUIREMENTS, AND ASSUMPTIONS

In this document, we assume that the hardware is dedicated to an instance of the SDS server. The SDS server instance includes the project to be run, any internal adapters and the resources required to service ad-hoc queries and subscribers and publishers. It does not include the resources required for the studio, external adapters and UDFs written in Java and C++.

We also assume there is little or no contention for LAN, network services, or disk resources.

To reasonably size the machine that runs an SDS project it is necessary to understand the logic and the rate (messages/second) at which data will flow through the critical sections in the model.

5 SIZING ESTIMATION – A QUICK WAY

This section describes a quick way to estimate the amount of CPU and memory required for an SDS project, while making certain assumptions about record size and throughput. It also provides usage recommendations for machines of various sizes. This information can be extrapolated to suit the needs of a specific project. A more detailed explanation of how to compute memory and CPU requirements is provided in later sections.

SDS can run on machines of various sizes - i.e. number of processors and amount of memory. The machine size dictates the number of projects that can be run, the number of operations that can be performed, the complexity of the project, number of clients (subscribers, publishers and adapters) etc.

In general the performance of an SDS project is dependent on the number of cores and the processor speed. Depending on the complexity of the logic a Stream/Window performs it can use up to one full core. To be able to use more than one core for a particular logic consider using the partitioning feature.

The number of Streams/Windows that can be processed, for a given number of cores and throughput, can be increased by simplifying the complexity of the logic performed by the Streams/Windows, reducing the number of clients (especially subscription clients) and using micro batching (envelopes or transaction blocks).

5.1 CPU Usage for Various Operations

The CPU usage for each operation depends significantly on the number of columns in each Stream/Window, complexity of the expressions and the throughput. The table below provides a rough idea of the relative CPU usage of various commonly used operations while making the following assumptions:

- A constant throughput of 200K messages per second.
- Processor is an Intel Xeon 2.6 GHZ on machine running Linux
- Assumes there is one dependent stream on the stream/window whose CPU utilization is listed.
- Assumes rows are processed one at a time i.e. no micro batches using envelopes and transaction blocks. Micro batching reduces CPU significantly and thereby increases throughput at a slight cost of latency.
- Reasonable complexity is used for the various operations. Note that CPU usage increases with increase in complexity.

Operation	CPU Used	Comments
Input Stream	0.45	
Input Window	0.60	
Compute Stream	0.60	

Compute Window	0.75	
Filter Stream	0.55	
Filter Window	0.65	
Aggregate Window	1.0	
Window – Window Join	1.0	
Stream – Window Join	0.9	
Retention Window	0.7	With KEEP Clause
CCL Script operation		
Each publisher	0.2	
Each subscriber	1.8	For first subscriber to Stream/Window. 1.2 CPU's for subsequent subscriptions.
Hana Adapter	0.7	Properly tuned.

In addition, it is recommended to assign 0.5 cores for the cluster manager and additional 1-2 cores for the SDS Web Service Provider if it is used.

The total number of CPU's required for the project can be determined by adding the total CPU used by each operation. The CPU usage can be increased or decreased proportionately depending on whether the expected throughput is more or less than the 200K assumed here.

Also, when using micro batching, i.e. transactions and envelopes, the CPU consumption reduces by about 30%. So for example the CPU usage for an aggregation operation will be 0.7 vs 1.0. This means that by using micro batching, the same number of cores can be used to run more Streams/Windows or more complex logic than otherwise would be possible without it.

NOTE: The CPU used by a Stream/Window cannot exceed 1.

5.2 Memory Usage for Various Operations

The table below summarizes the memory used per entry in various CCL constructs while making the assumption that each record has 100 bytes of packed user data and 10 columns.

NOTE: Each column in a row has a 4 byte fixed overhead plus the memory needed to store the value.

Construct	**Overhead (Bytes) per Row/Entry**	**Total Per Row/Entry Memory Usage** (100 bytes data per row + 4 bytes/column overhead (40 bytes))	**Comments**
Stream/Keyed Stream	0	0	Only transient memory
Window in Memory Store	128	268	
Windows in Log Stores	0	0	The space specified for the Log Stores is used for the events in the Window. Log store size cannot exceed amount of physical memory in the machine.
Window (Named/Unnamed) with Retention	200	340	If data is stored in a preceding memory store window subtract 140 bytes
Secondary Index	80	80	Additional indices

(Joins)			require additional memory
Aggregation Index	Aggregation Entry – 128 Aggregation Group – 328	Aggregation Entry – 356	Aggregation Entry represents each input row for the aggregation. Aggregation Group Entry represents the memory required to maintain the aggregation group assuming 1 GROUP BY COLUMN. If record is stored in a preceding memory store window then subtract 228 bytes
Dictionary(integral, integral)	170		An integral type is like an integer, long, seconddate etc. For strings and binary values add the average length of each string/binary value
Dictionary(integral, record)	210	438	If record is stored in a preceding memory store window then subtract 228 bytes
Dictionary (record, record)	250	706	If record one record is stored in a preceding memory store window subtract 228 bytes. If both are stored subtract 456 bytes.
Vector(integral)	70	70	An integral type is like an integer, long, seconddate etc. For strings and binary values add the average length of each string/binary value
Vector(record)	110	338	Subtract 228 Bytes if record is stored in a preceding memory store Window.
Event Caches	See Section 6.1.7.3		

Note: See the section 6.1 for more details and explanation on how to estimate memory for the various constructs

To calculate memory usage for a construct say a Window, the Total per Row/Entry value needs to be multiplied by the number of rows in the data construct. If the Window has 1 million rows then the memory used by the Window will be 268MB assuming that a row has 10 columns and a total of 100 bytes in across all columns. Section 6.1.3 below describes how to calculate the row size for custom rows.

NOTE: When calculating memory usage for Joins and Aggregations, where the target is a Window, both the memory consumed by the corresponding indices and the memory consumed by Window needs to be taken into account.

In addition to the memory requirements for the various CCL constructs the following memory requirements also must be accounted for

- 20% of the memory in machine should be reserved for temporal memory usage. This is the memory required for processing events and intermediate storage.
- 1-2GB for the OS assuming that the machine is fully utilized for SDS.
- 45 MB for the server + 4 MB for each stream/Window. For example a model with 200 streams/windows will require 845 MB of memory.
- 120 MB for the cluster manager + 5 MB for each project that is added.

5.3 Suggested Usage for Various Machine Sizes

Here is a suggested usage for machines of various sizes while assuming the following

- Throughput of about 200K messages per second (assumes logic can support it)
- Average record size of 100 bytes of actual data (ignoring record overhead)
- Processor is an Intel Xeon 2.6 GHZ on machine running Linux
- One Core on 4-8 core machine 2 cores on larger machines dedicated to other purposes.
- 1 GB memory on 8-16GB machines and 2 GB on larger machines reserved for OS
- 20% of memory dedicated for temporal memory needs for the SDS project.
- The machine is dedicated to the SDS project.
- The suggested number of Streams/Windows is for the ones continuously processing data and not for ones occasionally processing data or holding reference data.
- The suggested number of rows that can be held in memory or log stores assumes that there are no CCL Script (formerly called SPLASH) data structures such as vectors, dictionaries and event caches being used. If there are then the number needs to be reduced appropriately.

Size	Specifications	Suggested Usage
XS	4 Cores / 8 GB	❖ One SDS project ❖ One continuous publisher ❖ One continuous subscriber for example Hana adapter. ❖ 3 to 4 Streams or 2 to 3 windows performing simple operations like computes, additive aggregation and filters ❖ Reference elements with no or very small amount of caching ❖ About 20 million rows across all windows in memory and log stores.
S	8 Cores / 16 GB	❖ Not more than 2 small projects (see XS size project) or one larger project. Can be connected using project bindings ❖ 1-2 continuous publishers including adapters. ❖ 1-2 continuous subscribers including adapters. ❖ 5 to 6 Streams/Windows doing simple operations like computes, additive aggregation and filters ❖ Reference elements with small cache of a few 100 thousand rows. ❖ 1-2 complex operations like joins/aggregations or Flex ❖ About 40 million rows across all windows in memory and log stores.
M	16 Cores / 32 GB	❖ 2 – 3 project with various operations or 1 larger project. Projects can be connected with bindings. ❖ 20 to 25 continuously processing streams

		and windows. Number of operations and complexity that can be supported can be computed using formula in section 5.1 ❖ Reference element with cache of few million rows. ❖ Partitioning one or two complex logic to get better throughput. ❖ Guaranteed Delivery ❖ About 90 Million rows across windows in memory store, log stores and guaranteed delivery storage.
L	32 Cores / 64 GB	❖ Large projects with 40 to 50 continuously processing streams/windows. Actual number and complexity can be estimated using formula in section 5.1 ❖ 3 - 4 smaller projects, which may be connected with bindings. ❖ Reference elements with cache of several million rows. ❖ Partitioning 3-4 complex logic to get better throughput. ❖ About 200 Million rows across windows in memory store, log stores and guaranteed delivery storage. ❖ Guaranteed Delivery with longer periods where subscribers don’t connect.

6 SIZING ESTIMATION – DETAILED APPROACH

In a number of cases the extrapolating from the quick approach to sizing described above will suffice to determine the size of the machine. Use the detailed approach if the sizing requirements cannot be determined by extrapolating from the quick approach or a more accurate estimation is desired.

6.1 Memory Requirements

SDS, for the most part, is a memory based application so it is important that there is enough physical memory for it to perform efficiently. If the server starts paging out memory because of insufficient physical memory then performance will be impacted negatively. So it is important to determine how much memory SDS will require for your application.

Memory requirements can be classified into two categories namely persistent memory and temporal memory. Persistent memory is the memory required to store Window content, aggregation/join/retention indexes and CCL SCRIPT (formerly called SPLASH) data structures. Temporal memory is the memory required to move data from one stream to another, process subscriptions, ad-hoc SQL queries and the memory required to buffer records for streams or clients that are slower than its source.

In general it is fairly straight forward to determine the persistent memory requirements than to determine temporal memory requirements as the later depends on the nature of the dynamic uses of the application such as subscriptions, publications and ad-hoc queries.

6.1.1 Determining Persistent Memory Requirements

As mentioned earlier, persistent memory is the memory required to store the state of Windows, different indices and any CCL SCRIPT data structures. This section describes how to compute the size of the various components that make up persistent memory.

Note: For brevity and simplicity this section does not describe all the memory allocated by the server but only the major memory allocations. Add a 20% buffer to the computed value.

6.1.2 Primitive Data Types

Primitive data types are the fundamental building blocks of the more complex structures such as records, dictionaries, vectors and event caches. The size for each type is specified in the table below.

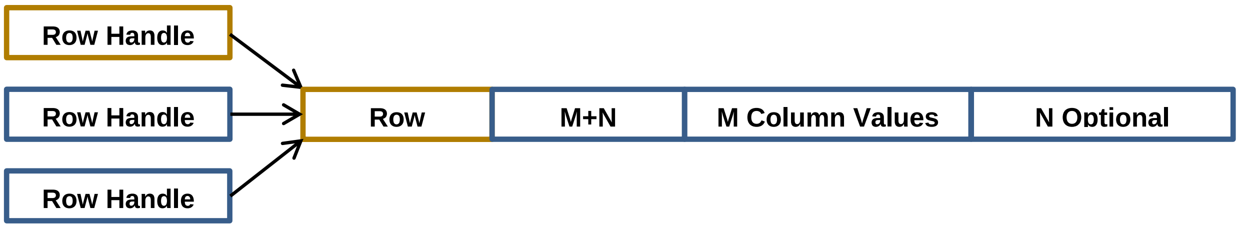
Data Type	Size in Bytes
Boolean	1
Integer	4
Long	8
String	1 + no of characters in the string.
Float	8
Money(n)	8
SecondDate	8
MSDate	8
Time	9
BigDateTime	8
Binary	4 + no of bytes in the binary value.
Decimal	24

6.1.3 Internal Records

An internal record is a structure that represents a row in a Window. It consists of two parts namely the data portion and the handle to the data. The data portion contains a fixed size header plus a variable size payload containing the column offsets, actual column data and any optional fields. The data type for each column value is one of the primitive data types and its length can be determined from the table above.

The row handle is a fixed size structure that points to the data. There may be more than one row handle for a given data. Typically this is the case when there are indices and multiple windows referring to the same data.

The figure below illustrates this concept.



In the above figure M represents the number of columns and N represents the number of optional values. The formula for calculating the internal record size is as follows

$$\text{HandleSize (40)} + \text{HeaderSize(56)} + (4 * (\text{M+N})) \text{ Offsets} + \sum_1^M \text{PS} + \sum_1^N \text{OV}$$

In the above equation PS stands for primitive type size for each of the M columns and OV stands for the option value size for each of the N options.

The following example illustrates how to compute the size of an internal row for a given schema. Typically the number of options is zero, so this example ignores options for the sake of simplicity.

```
CREATE SCHEMA TradesSchema AS  (
        TradeId     LONG,
        Symbol      STRING,
        Price       MONEY(4),
        Volume      INTEGER,
        TradeDate   BIGDATETIME
);
```

Step 1: Compute the size of the column values

Using the primitive size specified in section 6.1.2 compute the total size in bytes for the data values. For variable length types namely STRING in this example estimate an average length. For the purposes of this example let's assume the average length is 4. Bases on this the total length will be

8 + (4+1) + 8 + 4 + 8 = 33 bytes

Step 2: Add the offsets

Now add the size of the offsets to the size of the column values. The offsets is calculated as (4 * M) where M is the number of columns and is 5 in this example

(4 * 5) + 33 = 53 bytes

Step 3: Add the size of the Row Header

Each row head is 56 bytes. So the total size including the header is

56 + 53 = 109 bytes

Step 4: Add the size of the Row Handle

Each Row Handle is 40 bytes. So the size including the row header is

40 + 103 = 149

6.1.4 Windows

The amount of memory each Window consumes depends on the logic (Input Logic, SELECT Statement or CCL SCRIPT code) associated with it. The SELECT statement is either made up of one simple relational operation such as Filters, Unions, Aggregations, Joins or Projection or more complex statements that contain one or more of the simple relational operations.

In the latter case the compiler breaks down the complex operations into the simple operations listed above. So it is important to understand how much memory a Window that uses a specific simple relational operator consumes.

For the different types of logic described below you must know the following

- Average size in bytes for each row the Window produces as output using the formula described in section 6.1.3. For the purposes of this discussion let us assume that the average value size is ‘X’.
- The maximum number of rows the Window is estimated to have at any given time. Let us assume this number is ‘N’.
- The size of each Row Handle is known to be 40 bytes. Note that one instance of the row handle is already included average size of a record.
- The size of each entry in a Memory Store Index is 32 bytes.

Note: that for this discussion it is assumed that all the Windows are stored in a Memory Store. For Windows stored in a Log Store refer to section 6.1.8

Note: To determine the total size that the Window will require you must add the memory requirements for Retention, Aging and CCL Script Data Structures described later to the size computed using the mechanisms describe below.

6.1.4.1 Input Window

An Input Logic is the simplest form of logic associated with a Window. An Input Window is created for the following CREATE WINDOW statement

```
CREATE INPUT WINDOW Trades SCHEMA TradesSchema PRIMARY KEY (TradeId)
```

To estimate the memory for an Input Window use the following formula

$$(32 * N) + (N * X)$$

In the formula above ‘N’ is the number of rows in the Window and ‘X’ is the average internal record size of each row.

Plugging in sample numbers of 149 for ‘X’ and 1 Million for ‘N’ we get the following store size

$$(32 * 1M) + (1M * 149) = 181MB$$

6.1.4.2 Let SDS Do Your Work

The SDS server can compute the memory cost of storing a record in a Window assigned to a memory store. To do this, follow these steps

1. Create a Project with an Input Window having a desired schema and assign it to a memory store.
2. Load one typical event into a Window using the streamingconvert/streamingupload tool.
3. Use the streamingprojectclient tool and run the command ‘get_memory_usage’. Now the server logs contain the output for the memory used by all the Windows.

Here is a sample output for a Window with the schema used in the previous section.

```
[SP-6-131100] (97.154) sp(13087) CompiledSourceStream(Trades)::Memory Usage:
```

```
[SP-6-131040] (97.154) sp(13087) Store: 181 bytes in 1 records.
[SP-6-131094] (97.154) sp(13087) Input Queue: 0 bytes in 0 records.
[SP-6-131093] (97.154) sp(13087) Transaction Export Queue: 0 bytes in 0 records.
```

6.1.4.3 Compute Window

The Compute Logic essentially performs a projection on a source. The following CREATE WINDOW statement creates a Window that does Compute Logic.

```
CREATE SCHEMA TradesSchema AS  ( TradeId  LONG,  Symbol STRING,  Price MONEY(4),
Cost (MONEY(4),  TradeDate  BIGDATETIME);

CREATE WINDOW TradeCost PRIMARY KEY DEDUCED AS
SELECT trd.TradeId, trd.Symbol, trd.Price * trd.Volume  Cost, trd.TradeTime
FROM Trades trd;
```

The formula to determine memory used by a Window that does Compute Logic is the same as that for an Input Window i.e.

$$(32 * N) + (N * X)$$

Based on the computation in section 6.1.3 each row in this sample Compute Window is on average 149 bytes long. Using a sample value of 1 Million for ‘N’ we get the following store size the Compute Window

$$(32 * 1M) + (1M * 149) = 181MB$$

If instead of a doing a projection the Compute Window just copied the contents identically from the source i.e. Trades then the value of ‘X’ is just the size of a new Row Handle, which is 40. So the memory used by such a Window can be computed to be

$$(32 * 1M) + (1M * 40) = 72MB$$

6.1.4.4 Filter Window

A filter window does not do any projections on the source but just copies the contents of source directly for those rows that meet the filter condition. The following CCL defines a Window that does filter logic

```
CREATE WINDOW TradeFilter PRIMARY KEY DEDUCED AS
SELECT trd.* FROM Trades trd WHERE trd.Volume > 10000;
```

The formula to calculate the size of a Window that does Filter Logic is

$$(32 * N) + (N * 40)$$

The '40' refers to the size of a new row handle that points to the data values of the source record. Assuming the size of 'X' is 100K the size of a Filter Window is computed as

(32 * 100K) + (100K * 40) = 7MB

6.1.4.5 Union Window

A Union Window performs the logic of combining the data from multiple Streams and storing it in the Window. The following CCL statement creates a Window that does a Union logic.

```
CREATE WINDOW TradesUnion PRIMARY KEY DEDUCED AS
SELECT trd.* FROM Trades1 trd;
UNION
SELECT trd.* FROM Trades2 trd;
```

The formula for determining the size of a Union Window that just copies all the columns from its sources without any changes is as follows

$$\sum_{1}^{M}((32 * \mathrm{N}) + (\mathrm{N} * 40))$$

In the above equation M represents the number of SELECT statements that makes up the UNION statement and 'N' represents the number of rows in each Source Window in the SELECT statement.

In the above UNION CCL statement if the number of rows in Trades1 is 1M and the number of rows in Trades2 is 500K then the total memory consumed by the Window that performs a Union Logic is determined as follows

((32 * 1M) + (1M * 40)) + ((32 * 500K) + (500K * 40) = 108MB

If one or more of the SELECT statement did more than just exactly copy the columns from the source then you need to also additionally estimate the size of the intermediate Window(s) that will be generated by the compiler.

6.1.4.6 Aggregation Window

An Aggregation Window as the name suggests performs aggregation operation over a specified group of rows and produces one result per group. It is defined in CCL as follows

```
CREATE WINDOW TradeAggr PRIMARY KEY DEDUCED AS
SELECT trd.Symbol,  MAX(trd.Price) MaxPrice, MAX(trd.Volume)  MaxVolume FROM Trades trd
GROUP BY trd.Symbol;
```

The formula to determine the memory requirements for a Window that performs aggregation operations is as follows

(224 * M) + (K * M) + (128 * N) + (M * X) + (32 * M)

In the above formula,
‘N’ - represents the number of rows in the Source to the Aggregation Window,
‘M’ - represents the number groups and hence the number of records in the Aggregation Window and finally
‘X’ - represents the size of each output record as determined by the formula in section 6.1.3.
‘K’ – represents the size of the key columns converted into an internal row whose size can be determined using the formula in section 6.1.3.

Additionally ‘224’ represents the overhead for each group, ‘128’ represents the size in bytes of the aggregation indices required for each source row and ‘32’ represents the size of each node in the store.

Assuming the number of records in the source (N) is 1M and the number of unique Symbols in the source (M) is 10000, the size of the key record is 112(K) and the size of each output record as computed using the internal record size formula is 156 (X). Then the total memory consumed by this Window is

$$(224 * 10K) + (112 * 10K) + (128 * 1M) + (10K * 156) + (32 * 10K) = 133MB$$

6.1.4.7 Join Window

A Join Window combines data from multiple source windows and produces one or more rows per input row depending on the join condition. The following CCL statement produces a Join Window

```
CREATE WINDOW PortfolioValue PRIMARY KEY DEDUCED AS
SELECT pf.Symbol,  pf.SharesHeld, pf.PurchaseDate, pf.SharesHeld * trd.Price
FROM Portfolio pf INNER JOIN LatestTradesPrice trd ON pf.Symbol = trd.Symbol;
```

The formula to calculate the size of a Window that performs a Join Logic is as follows

$$\sum_{1}^{M}((72 \;* \mathrm{N})) + (X * S) + (32 * S)$$

Here ‘M’ represents the number of Windows being joined and ‘N’ represents the number of rows in each Window to be joined. ‘X’ represents the size of each output row of the Join Window computed using the formula in section 6.1.3 and ‘S’ is the number of output rows. Finally ‘72’ represents the sum of the Row Handle, which is 40 bytes, plus the size of an entry in the store of the Unnamed Window created over the source, which is 32 bytes.

Note: If a retention policy is specified in one of the sources in the FROM clause then set the value of ‘N’ for this source to *zero* to avoid double counting the storage used by this source. See 6.1.5.2 for more information.

In the above CCL assume the number of rows in the Portfolio Window (represented by N in the formula) is 5K and 100K for the LastTradePrice Window. Also assume that the size of each output row (X) is 152 bytes and there are 5K output rows (S). Then the size of the PortfolioValue Window is computed to be

$$(72 * 5K) + (72 * 100K) + (152 * 5K) + (32 * 5K) = 9MB$$

When the Join requires secondary indices then you need to add another (72 * N) to the above calculation for each secondary index created. Secondary Indices are created automatically by the server when the (DYNAMIC) keyword is used in the from-clause when specifying a Join and the join is performed on one or more non-key columns. Note that ‘N’ here refers to the number of rows in the Window being joined too and not the Window on which the (DYNAMIC) keyword is specified.

6.1.4.8 Flex Window

A Window whose logic is defined using CCL SCRIPT is a Flex Window. A Flex Window size uses the same formula as a Compute Window for the purposes of estimating the Window size except that there is no optimization performed when copying the source record exactly.

Flex Windows can additionally use CCL SCRIPT data structures to maintain intermediate state. The size computation for these data structures is described in section 6.1.7

6.1.5 Retention

Retention specifies the maximum amount of data that a Window may contain. In order to process the retention logic efficiently the server needs to maintain certain data structures that will consume memory. The size of these data structures depend on the maximum size of the window that needs to be maintained.

If the retention is row based then the maximum number of rows is already specified in the retention policy. The only caveat to this being that if the slack option is used for row based retention. If this is the case then the slack value must be added to the number of rows specified in the retention policy to get the maximum number of rows that can be stored in a Window.

If the retention is time based, then the maximum number rows that the Window can contain needs to be approximated based on two factors namely

- Approximate rate at which the rows are produced by a Window
- How long the rows need to be maintained in the Window

Based on these two factors the approximate size of a Window with time based retention can be calculated as follows

$$R * S$$

Where R is the number of rows produced by the Window every millisecond and S is the maximum number of milliseconds to hold a record.

For example if the Window produces approximately 10 rows per millisecond and each row needs to be maintained for 5 Minutes (300,000 milliseconds) then the approximate maximum number of rows the Window will contain is computed as

$$10 * 300K = 3M \text{ Rows}$$

Another variable when specifying retention is whether the retention is specified on a Named Window, an Unnamed Window or *both.* The memory used between these two modes is significantly different and they are shown below. When both kinds of retention are used you need to add the size computed for both modes to get the total memory required for the retentions data structures.

6.1.5.1 Named Windows

The following CCL statement defines a retention policy on a Named Window

```
CREATE OUTPUT WINDOW TradesSubSet PRIMARY KEY DEDUCED KEEP 500000 ROWS AS
SELECT * FROM Trades;
```

To compute the size of the retention structures use the following formula

$$72 * S$$

Where ‘S’ is the maximum number of rows that can be held in the Window and ‘72’ is the number of bytes required for each row for the retention data structures.

So in the example above the total memory needed for the retention data structures is

$$72 * 500000 = 360 \text{ MB}$$

Note that this size needs to be added to the Window Size computed in section 6.1.4 to get the total Window size that uses retention.

6.1.5.2 Unnamed Windows

Specifying a retention policy on the source(s) i.e. Window/Delta Stream creates an unnamed Window. The following CCL statement creates an Unnamed Window

```
CREATE OUTPUT WINDOW TradesSubSet PRIMARY KEY DEDUCED AS
SELECT * FROM Trades KEEP 500000 ROWS;
```

Here an Unnamed Window is created over the source Window i.e. “Trades”. To compute the size of retention data structures in such a case use the following formula

$$\sum_1^M ((144 * S))$$

In this formula ‘M’ represents the number of source Window/Delta Streams have a retention policy specified and ‘S’ represent the maximum rows that the Unnamed Window can hold based on the retention policy.

To compute the size of the retention data structures used in the above CCL substitute ‘1’ for ‘M’ and 500000 for ‘S’ and calculate the size as follows

$$(144 * 500K) = 720MB$$

6.1.6 Aging Policy

Aging identifies rows that have not been updated for the specified period of time. The following CCL statement specifies an Aging Policy for the records in a Window

```
CREATE OUTPUT WINDOW AgedTrades PRIMARY KEY DEDUCED
AGES EVERY 10 SECONDS SET Age 5 TIMES
AS SELECT trd.*, 0 Age FROM Trades;
```

To determine the size of the data structures required to process the Aging logic efficiently you need to first determine approximately the maximum rows the data structures can contain at any given time. This is dependent on the average rate at which the Window, on which the Aging policy is specified, produces rows. This in turn is dependent on the how much data is flowing into the Window and the complexity of its logic.

Once the average output rate is determined the maximum size that the Aging data structures will require can be determined using the following formula

$$T * N * R * 72$$

Here 'T' represents after how long a record, which has not received an update, is marked as aged. 'N' represents the number of times a record must be marked as aged and 'R' represents the rate at which the Window is producing rows. Finally, '72' is the size of the information in the Aging data structures for one record.

Note: The above formula assume the worst case scenario where there are only inserts with no updates. The best case scenario is where there is an initial set of inserts followed by only updates. In this case T*N*R can be substituted by the number of initial updates.

Using the above CCL as an example and assuming that 'R' is 100K rows per second then the Aging Data Structure size can be calculated as

$$10 * 5 * 100K * 72 = 360MB$$

6.1.7 CCL Script Data Structures

CCL SCRIPT (formerly called SPLASH) supports three types of data structures apart from the basic types and the record type. These data structures are

- Vectors
- Dictionaries
- Event Caches

These data structures may be used with almost all Windows, Streams and Delta Streams and also in the global declare block. These data structures may consume a significant amount of memory and therefore it is important to estimate the size of every occurrence of these data structures in your model.

For vectors and dictionaries use the following table to determine the size of each entry

Data Type	Size in Bytes
All primitive types except String and Binary	48
String	33 + string length*
Binary	32 + binary length*
String	1 + no of characters in the string.
Record Type	40 + (Internal Record Size or

	40)**
Vector	64 + Vector Size***
Dictionary	96 + Dictionary Size***

*For variable length types such string and binary use the average length of type across all the rows.

**For Record Types you may or not have to include the Internal Record Size depending on whether the record is referenced only in the vector/dictionary or is it referenced in a Window that is not in a Log Store or Event Cache. If it is the former then just add 40 bytes for the Row Handle or else add the Internal Record Size, the computation for which is shown in section 6.1.3

***When a vector or dictionary is nested inside another vector or dictionary, you need to estimate the size of each of the nested entries to get a size estimate of the top-level vector dictionary.

6.1.7.1 Vectors

To calculate the memory requirement for a vector use the following formula

$$N * (8 + S)$$

Here ‘N’ is the number of entries in the vector and ‘S’ is the size of each entry in the vector as determined from the table above.

For example the size of a vector of integers having a 1M entries will be

$$1M * (8 + 48) = 56MB$$

6.1.7.2 Dictionary

To calculate the size of a dictionary use the following formula

$$N * (64 + K + S)$$

Here ‘K’ is the size of the dictionary key and ‘S’ is the size of the value. Both ‘K’ and ‘S’ is calculated using the table described in section 6.1.7

For example the size of the dictionary defined as dictionary(string, integer) is computed as follows assuming there are 1M entries and the average size of each string is 50 bytes including the fixed overhead of 33 bytes.

$$1M * (64 + 50 + 48) = 162MB$$

6.1.7.3 Event Caches

The Event Cache size can be determined by adding the memory requirements for each of the features of the event cache that is used. This can be expressed as follows

$$(G * 1.5KB) + (64 * N) + (64 * N * O) + (72 * G * O) + (64 * N * C)$$

In the above formula

G – No of Groups i.e. number of unique key values in the event cache. Default is the primary key o the record.
N – Number of rows in the event cache. This will be the number of events specified in the events options * G if the events option is specified.
O – Set this to 1 when ordering events, otherwise set this to 0.
C – Set this to 1 when coalescing events, otherwise set this to 1.

For example, if the defined event cache is *eventcache(trades[Symbol], coalesce, Price desc).* Also, assuming the number of unique symbols is 5000 (G) and N is 1M. Then the size of the event cache is

(5K * 1.5KB) + (64 * 1M) + (64 * 1M * 0) + (72 * 5K * 0) + (64 * 1M * 1) = 135MB

Note: If the event cache is defined over a Stream or Delta Stream and the event is not stored as is anywhere else then you need to add N * (S - 40) to the above total where N is the number of events and S is the internal record size of each event as computed in section 6.1.3

6.1.8 Log Stores

Log Stores are essentially a memory mapped file. So the majority of each Log Stores memory requirement can be determined from the ‘maxfilesize’ property of the log store. Refer to the section in the administration guide that describes how to size a log store for more information on setting the ‘maxfilesize’ property.

There are a couple points to important points to note about sizing Windows assigned to a Log Store
1. The optimization done when the Window contains the exact copy of the source record is not applicable when the Window is assigned to a Log Store. So Windows that perform Filter and Union operations will not just contain a row handle that points to the column values of the source record but will contain a separate copy of the record. However, you do not need to estimate the size of such Windows separately because this size is already included in the size of the log store.
2. Any unnamed windows, CCL SCRIPT data structures and aggregation indices associated with a Window stored in a Log Store are not stored in a Log Store and its size needs to be computed as previously described.

6.1.9 Intermediate Queue Size

Each Stream regardless of whether it is a Stream/Delta Stream or Window has a fixed size queue that is 1024 transaction block (each block may have more than 1 row) deep. In an ideal model all Streams will process data at approximately the same rate and there will be only a few entries in the queue. However in a number of cases the source Stream/Window will produce rows faster than the dependent stream can process these rows.

When this happens the queue for the dependent stream starts filling up. When the fixed queue size of 1024 is reached then the source is unable to push data to the dependent streams and stops processing more data till there is sufficient room in the dependent stream. This in turn will cause the source streams queue to fill up and so on so forth all the way up to the input streams until no more new rows are processed till the backlog is cleared.

This above described scenario may happen even if all the Streams work efficiently but there is a slow client. Therefore it is prudent to estimate the memory required for the intermediate queues.

The intermediate queue size can be estimated using the following formula

$$\sum_{1}^{N}(1024 \ * \mathrm{B} * 40)$$

In the above example 'N' is the number of Streams/Windows and 'B' is the transaction block size of the rows produced by the source.

The transaction block size will vary from Stream to Stream. In most cases the transaction block size produced by a source is the same as what it receives. The exceptions to this rule are the Input Windows, Flex Streams, Pattern Streams and the cases where Many-Many joins are involved.

An Input Window will at the most produce twice the number of rows in a transaction block that it receives from an external source. For the other cases it varies significantly and you have to make the determination based on the knowledge of the application. At a minimum assume 'B' is twice the size of the transaction block size the adapter sends.

Assuming that a model has 3 Streams and the block sizes are 2, 2 and 5 respectively then the queue size is estimated to be

$$(1024 * 2 * 40) + (1024 * 2 * 40) + (1024 * 5 * 40) = 368K$$

In the example the memory requirement may appear small. However, when using large transaction block sizes in the input and there are 100s of Streams involved the memory requirement will be significant.

6.1.10 Project Memory Usage

In addition to the memory used by Windows and Streams when data is being loaded there is a fixed cost to starting a project and creating Windows and Streams. The cost is as follows

- 45 MB for starting a project
- Approximately 4 MB for each Stream/Window

So for example if a project with 100 Streams and Windows is started then the cost is calculated as

$$45MB + 100 * 4MB = 445MB$$

6.1.11 Cluster Memory Usage

The cluster manager uses a significant amount of memory. The memory usage is as follows

- 120 MB for each cluster manager instance. Typically there is one per SDS cluster node.
- 5 MB for each project that is started.

So for example if there 5 projects started in a cluster node then the cluster manager memory cost is calculated as

$$120MB + 5 * 4MB = 140MB$$

6.2 Estimating Temporal Memory

Temporal memory is the memory required to process the current record in each Window, store records in intermediate queues before it can be processed by clients and the memory required to service ad-hoc queries and subscribers.

6.2.1 Current Record Processing Memory

Typically the memory required for processing the current record is insignificant and can be ignored. However, in the case of Flex Streams it is conceivable that the memory requirements may be significant and this needs to be determined using the techniques described in section 6.1.7

6.2.2 Subscriber Client Queue Size

Each subscriber queue is by default 8192 entries in size. This size can be reconfigured by the subscriber to be a different size. If the client is slower than the source stream then this queue will get filled up. If you suspect that there may be times where there will be bursts of data when one or more clients cannot keep up with the data you need to include this memory needs in your estimation.

The subscriber queue is filled with records in the external record format. This format is roughly 64 bytes smaller than internal row format. So for simplicity you can compute the internal record size and subtract 64 bytes from it to determine the external record size.

To calculate the memory required for slow external subscribers use the following formula

$$\sum_1^M \left((\mathrm{Q} * \mathrm{B} * (\mathrm{S} - 64))\right)$$

Here ‘Q’ is the size of the subscriber queue (default 8192), ‘B’ is the transaction block size produced by the stream being subscribed too, ‘S’ is the internal row size of the stream being subscribed too and finally ‘M’ is the number of potentially slow clients.

6.2.3 Subscriber Client Processing

For subscribers that just does a projection, filter or just subscribe to the data as is there is no more memory required than the memory required for the subscriber queues as discussed in section 6.2.2. For subscribers that do more than that, the memory requirement increases depending on what the subscriber does.

6.2.3.1 <u>Aggregations</u>

For each subscriber that does an aggregation, it is like dynamically introducing an aggregation window so that the server can maintain the aggregation state for the lifetime of the subscriber. You can use the techniques described in section 6.1.4.6 for Aggregation Windows to estimate the memory requirements for such a subscriber.

6.2.3.2 <u>Order By</u>

If a subscriber client does an order by then the server needs to maintain a Window with the contents of the SQL projection along with the necessary indices to inform the client of changes to the order of the rows. You can use the sizing formula for Compute Windows in section 6.1.4.3 for the former.

6.2.3.3 <u>Pulsed Subscription</u>

For subscribers that need pulsed subscription, the server needs to cache the records for the specified interval before releasing it to the customers.

You can use the sizing formula for Compute Windows as described in section 6.1.4.3. For determining the number of rows in the window you need to know the maximum number of inserts that can occur within the pulse interval.

To estimate the size of each row, first determine if the client is using SQL subscription or not. If the client is using SQL subscription you can use that SQL that to determine size of each row. Otherwise you can use the formula that is used when the Compute Window copies the source record identically.

6.2.4 Ad-Hoc Query Processing

Ad-Hoc query memory requirements depend on the nature of the ad-hoc query. A possible worst case scenario is that every ad-hoc query running in parallel will require as much additional memory as the Window being queried uses to stores it records without the copy optimization.

So for example if there are three ad-hoc queries running in parallel against three different windows, which use 100MB, 250MB and 50MB to store its records then the worst case scenario will be that you will require

$$100MB + 250MB + 50MB = 400MB$$

to execute the query.

Note: It is possible to require more memory than what the source window uses if the ad-hoc query projection has more columns than the window being queried or the columns are very different in nature.

6.3 Processor Requirements

When there are not enough processors to handle the requirements of your SDS application, throughput drops and latency increases significantly. So it is important to correctly estimate the number of processors required for you application. Processor requirements not only depend on the size and complexity of the model being executed but also the speed of the processor and data arrival rates. Therefore it is not possible to estimate the processor requirements using an analytical approach.

The approach recommended here is to use an empirical approach to determining processor requirements. In order to do a meaningful estimation the model is broken down into smaller models (sub-graphs) with the size of each sub-graph being constrained by the size and memory of the test machine available for performing the analysis.

To estimate the number of processors using this approach you need to

1. Understand the SDS threading model.
2. Break the model into the required number of sub-graphs.
3. Capture inputs for each sub-graph.
4. Capture and analyze throughput and CPU usage statistics for each Stream in the sub-graph.
5. Compute the number of CPUs required for the model and budget additional CPUs for subscriptions and ad-hoc queries etc.

6.3.1 The SDS Threading Model

Before being able to estimate the number of processors, you need to understand the SDS threading model. SDS is a heavily multithreaded application. Each Stream/Window and Publisher (including Internal Adapters)/Subscriber/Ad-Hoc Query servicing logic runs in its own thread. Each of these threads will compete for the limited CPU resources when there is work to do. As each thread can only use one CPU at a time ideally there will be no competition if there are as many CPUs as there are threads. Practically speaking this is not feasible given the number of threads that will be spawned but it is suffice to say that the larger the number of CPU the better

To limit the competition for CPU a self-tuning mechanism of fixed size queues are used to dedicate as much CPU resources to the threads that need it the most. With fixed size queues, a Stream that is processing faster than its dependents will fill up the dependents queue and be blocked from further processing. This reduces competition for the CPUs and gradually the slowest Streams will get larger CPU time slices to drain its queue.

6.3.2 Breakdown the Model into Sub-Graphs

If the number of Streams in your model, excluding any reference data Streams that can be preloaded, is smaller than the number of cores in the largest box you have to test then you can treat the entire model as one sub-graph. Otherwise you will need to split the Streams into smaller sub-graphs. Each sub-graph will have the following characteristics

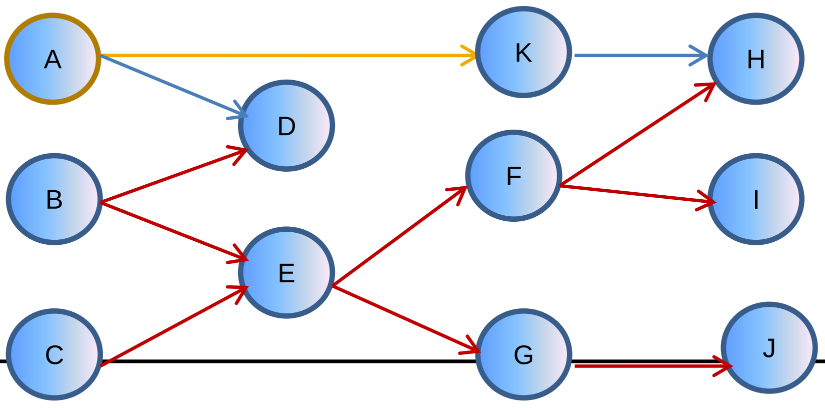
- Each sub-graph is a complete model that can be compiled and executed by the server.
- The number of streams in each sub-graph, excluding Streams containing reference data that can be preloaded, should as much as possible not exceed the total number of processors you have in your test machine. The idea behind this is to determine the throughput in each node in your graph without any contention. Sometimes having one processor per Stream is not feasible because you need to include all the dependent Streams of the Streams you are gathering statistics for to get the expected throughput for the Stream. In this case ensure that you replace the dependent Stream with a dummy insert-only stateless Stream that just copies data from the source. This will keep the CPU and Memory requirements for these dependent Streams to a minimum.
- Each sub-graph will be typically 3 levels deep except when measuring statistics for Input Streams or a leaf Streams in the graph.
    - The first level is all the input nodes (Source Streams) required for this sub-graph.
    - The second level is the nodes you want to capture statistics for.
    - The third level is all the nodes that depend on the nodes you want to capture statistics for. With respect to these dependent nodes it only matters that there be the same number of dependent nodes as there is in your model. It does not matter what they do. So for all practical purposes they can just be a stateless Stream that just copies the contents of the Streams you want to capture statistics for.

- If your machine has more spare cores you can include more than 3 levels in your model as long as you include all the nodes dependent on the last level and there is one core per node.
- All the sub-graphs combined must include all the Streams in the model.
- The second last level of Streams in a given sub-graph will become the input nodes for the next sub-graph. If your graph is 3 levels deep then the second level i.e. the level you are capturing statistics for will become the source for the next sub-graph.

After you have identified the sub-graphs you need to convert these sub-graphs into a self-contained model by converting the intermediate streams that will be sources for a sub-graph into a true Input Stream or Input Window depending on whether the intermediate stream is a Stream or a Window/Delta Stream respectively. You need to use the schema of the intermediate stream as the schema for the Input Stream or Window.

Note: For a typical model you will need at least a 6 core box for this exercise because you will need to reserve 2 cores for miscellaneous server activities such as input handling, subscription handling, handling additional dummy dependent streams that may be required etc.

To illustrate how to break down a model into sub-graphs assume you have a six core machine (2 cores reserved and 4 cores available) and the following directed graph of your model



In the model streams A and K contain reference data that can be preloaded. The Streams that the red arrows are pointing to are the Streams that will be competing for CPU during normal operation of the model and must be profiled.

You can break down the model into the following sub-graphs. The Stream name in bold are the streams being profiled

1. A, **B**, **D**, E': Here E' is just an insert-only Stream that copies the content of B. This is required to satisfy the condition that the number of dependent streams for any stream in the sub-graph needs to be exactly the same as the actual model.
2. **B**, **C**, **E**, F',G': Here again F' and G' prime are just insert-only streams that copy the contents of B and E. Note that there are 5 Streams here and we have only 4 cores. This does break the recommendation that each sub-graph have only as many Streams as cores in the test machine but given the resources this is the best that can be done. Making F' and G' as insert-only streams reduces the CPU and Memory requirements to the dependent Streams to a minimum.
3. E", **F, H, I.** Here E" becomes an Input Stream for this sub graph. The data for E" was captured when E was profiled in the previous step. F is the Stream being profiled. H', I', G' primes are dummy dependent streams that needs to be included to ensure are being profiled.

   If you expect the leaf nodes H and I to have subscribers then you need to attach at least one subscriber to the leaf nodes. If you don't have enough cores available to run a subscriber on your test box, run it on a different box to minimize unnecessary CPU contention. When doing so ensure that the network or the subscriber does not contribute to an artificial slow- down of the model in any way.

   A and J are reference streams required for the processing of **H** and are preloaded before the data in **E"** is loaded.
4. E", **G, J:** Here again E" is the Input Stream for this sub-graph and **G** and **J** are profiled. Again, if you expect **J** to have a subscriber then attach a subscriber running on the either the same machine or different machine depending on the availability of cores.

6.3.3 Capture Inputs for each Sub-Graph

After you have created models for each of the sub-graphs you need to create inputs for each of these models. It is important that all the inputs for the model are captured in a single file. Otherwise it will potentially skew your results.

Note: Capturing all the inputs in the same file has the drawback that you cannot load different sources at different rates given that the upload utility has only one rate option. In such a case choose the fastest rate among all the inputs.

To capture inputs run as large a model that will reasonably perform on your test box and subscribe to one or more streams that will be inputs to one of your models using a single streamingsubscribe instance. Direct the output to a file and name the file such that you can identify this file as being an input to a particular model. You may be able to capture inputs for multiple sub-graphs using multiple streamingsubscribe instances.

You then pre-convert the captured data for each sub-graph into the binary record format using the streamingconvert tool and the model for which the input file belongs. This will ensure that the conversion from xml or csv to binary does not become the bottleneck that artificially slows down your model.

6.3.4 Capture and Analyze Runtime Statistics

Once you have determined the sub-graphs you need to analyze, it is time to run each of the models corresponding to the sub-graphs with real life data loaded at rates that you will expect in a production environment.

Follow these steps to capture run time statistics for each sub-section

1. Start the model with the server option to capture Stream statistics every few seconds enabled.
2. Load any reference data that you sub-section of the model will require to produce correct results.
3. Load the real life data that you have previously captured at a rate that is close to what is expected in production. If the input stream is derived from a intermediate stream in the original model (for example E" in the illustration above) then you need to determine the throughput of E from the statistics you have captured and use that as the upload rate.
   You can use the streamingupload tool for this purpose, which has a mechanism to control the upload rate. If you don't have at least two spare cores to run the streamingupload tool, then run it from a different machine ensuring that the network is not a bottleneck.
4. After all the streams have production volumes of data in them connect the streamingmonitor tool to the server and redirect its output to a file to capture the output. Use a different file for every model that you run.
5. Let the system run in this state for at least a couple of minutes with data flowing at production rates and then stop the system.

Once the statistics have been captured you need to analyze the statistics and consolidate the cpu_pct value for each stream being profiled. There is one entry for cpu_pct in the file for every time period the report is produced. Pick the highest cpu_pct from the set that does not appear to be an anomaly. If a stream is profiled more than once use the lowest cpu_pct value across the profiles. The table below shows a sample consolidated cpu_pct report for all the Streams in the illustration in section 6.3.2.

Stream	CPU %
A	0
B	40
C	35
D	55
E	100
F	75
G	50
H	95
I	65
J	30
K	0

Streams A and K don't have 0 CPU percentages because these are reference streams and its data was preloaded.

6.3.5 Compute the Number of CPUs

To compute the number of CPUs required to run the model alone without any other load just add up the CPU percentages consolidated in previous section. That will be

$$40 + 35 + 55 + 100 + 75 + 50 + 95 + 65 + 30 = 545$$

Rounding this to the closest 100s greater than the sum we get 600, which is equivalent to 6 CPUs.

To this total you need to add the CPU used for the following

- Internal adapters
- Handling external subscribers
- Handling external publishers
- Ad-Hoc Query Handling

6.3.6 Internal Adapters

To compute how much CPU an internal adapter uses you need to know following

- The maximum capacity of each adapter in rows/second.
- For an input adapter you need to know approximately the rows/second of actual data it will be feeding into the input stream it is attached too.
- For an output adapter you need to know the rows/second the stream that the output adapter is attached too is producing. You can determine this based on the rows per second produced by the stream in the statistics captured in section 6.3.4

With this information you can calculate the number of CPUs required for internal adapters as

$$\sum_{1}^{M}\left((\mathrm{X}/\mathrm{Y}) * 100\right)$$

Here ‘M’ is the number of internal adapters. For input adapters ‘X’ is the number of rows per second expected to be fed by the adapter for your application and for output adapters it represents the number of rows per second produced by the stream the adapter is attached too. ‘Y’ represents the maximum throughput of the adapter.

6.3.7 External Subscriber Overhead

It is difficult to estimate the subscriber overhead using an analytical approach so it is best to determine this using an empirical approach using a technique similar to what you used to determine the number of CPUs the model by itself uses.

To determine subscriber CPU overhead for the subscribers to a stream, follow these steps

1. Determine the rate at which the stream being subscribed to produces its output. This can be determined from the table you created as described in section 6.3.4. Assume this value is ‘R’.
2. Next capture the output produced by the source stream being subscribed to, using the streamingsubscribe tool if you have not already done so previously.
3. Create a model with one input Stream or Window depending on whether the subscriber source is a Stream or Window/Delta Stream. The input Stream/Window must have the same schema as the source being subscribed too. Then compile the created model and create the corresponding ccx file.
4. Next convert the captured data to binary format using the streamingconvert tool and the ccx file created in the previous step.
5. Next run the model and load the data at the rate of ‘R’ determined in step 1, using the streamingupload tool, without any subscribers while monitoring cpu usage on the streaming server using a tool such as top. Note the peak cpu usage. Let this value be X.
6. Now rerun the model, while monitoring the server cpu usage, with as many subscribers with the same type of subscriptions as you expect to have in production for this Stream and reload the data again using the sp_upload tool at the rate of ‘R’. Determine the peak cpu usage. Assume this is ‘Y’.
7. The difference between ‘Y’ and ‘X’ will give you the CPU used by your subscribers. You need to add this value to the CPU totals you have determined previously.
8. Repeat steps 1 – 7 for each Stream.

Note: If the throughput of the Stream is fairly high then you may notice that the throughput of the Stream being subscribed too drops significantly after adding a few subscribers. This is most likely due to the fact that the Stream Exporter thread for the Stream, which services the subscribers, is not able to feed data fast enough to the subscribers and is becoming the bottleneck. Another reason could be that one of the subscribers is too slow or the logic that needs to be applied on the data before it is sent to the client is too expensive.

When the Stream Exporter or one of the subscribers become the bottleneck then adding more subscribers will not significantly increase CPU use because the model will start slowing because of the fixed size of intermediate queues. This reduces overall CPU requirements of the model.

6.3.8 External Publisher Overhead

When an external publisher publishes to a Stream or Window the server incurs an overhead to handle the publisher. For each publisher this overhead is roughly 50% of the CPU used by the target stream when the target is an Input Window and 100 % when the target is an Input Window.

So for example if the target Input Window for an adapter is running at 50% CPU utilization then an external publisher overhead is 25% CPU. Similarly if the target Input Stream is running at 50% CPU utilization then CPU overhead for each publisher is 50% of a CPU.

Note: Once the target Input Window or Stream is saturated i.e. it is running close to 100% CPU utilization with its queues constantly filling up to its maximum i.e. 1024 then adding any more publishers to that Stream or Window has a very small amount of CPU overhead.

6.3.9 Ad-Hoc Query Overhead

Each Ad-Hoc query pretty much consumes 100% of a CPU for the finite time that it runs. The length of the query depends on the complexity of the query and size of the Window that is being queried. When the query runs the source Window is prevented from processing data and therefore does not consume any CPU.

Depending on how often you expect ad-hoc queries to run, how many parallel queries you anticipate and how long the queries may run you may want to add additional CPUs.

6.4 Other Requirements

6.4.1 Disk Requirements

SDS requires a modest amount of disk to install and do basic operations. This is roughly requires about 1GB of space. SDS does not have to be installed on a high performance disk.

However, when using Log Stores a high performance disk is required. It is recommends that a RAID controller stripping across several high performance disks with a stable write cache be used.

It is recommended that the size of the disk be at least three times the memory requirements of the SDS server.

6.4.2 Networking Requirements

For networking, the use of a Gigabit Ethernet or better is recommended for high volume applications. But, even a Gigabit might not be enough especially when you have high message rates and/or also have clustering or HA enabled. Consider a scenario where you are publishing 150K messages per second with each message size being 150 bytes in size. This will require 1.8 Giga bits of bandwidth.

For such cases you may use a 10 Gigabit Ethernet card or have multiple network interfaces (e.g. multiple Gigabit Ethernet NICs) to increase bandwidth.

7 CONCLUSION

Estimating processor and memory requirements for your SDS application is not trivial. This paper showed you ways to

- Estimate CPU and Memory requirements by extrapolating from pre-computed values.
- Estimate the size of Persistent Memory requirements using formulas.
- Shows you what to consider when estimating temporal memory requirements.
- Shows you how to empirically determine Processor usage.

You need to add about 20% margin to the computer memory and processor requirements for items that are not large enough to document but still significant in the larger picture.

Finally, the computed resource requirement is an approximation and a minimum requirement. It is strongly recommended that you add a margin of safety to these computed values for any future changes to the model and any unanticipated requirements.

www.sap.com