THREEGLAV

stream-horizon.com

StreamHorizon in a Nutshell

Version 3.x

STREAM HORIZON

DataProcessingPlatform

TABLE OF CONTENTS

STREAM HORIZON IN A NUTSHELL

Reading this document which is predominantly comprised of xml snippets (StreamHorizon configuration listings/fragments) will make you familiar with core features of StreamHorizon. Document covers simple Data Warehousing project which loads flat files into Data Mart in massively parallel fashion (Demo is available for download at StreamHorizon website).

This document does not focus on advanced features and connectors (for Hadoop, Messaging, Thrift etc), Performance Tuning, Customisations of StreamHorizon platform and other more advanced topics.

BUILDING SIMPLE FILE TO DATABASE ETL PROCESSING PIPELINE

This example is taken from Sales Demo which comes with StreamHorizon demo trial distribution. Sample project receives flat files as input to ETL stream and inserts processed data into Data Mart. It also gives example how to achieve high level of parallelism while inserting data into database. We will focus on both JDBC and External Table modes of inserting data into the database. This example is using Oracle syntax, however, if you wish to use MSSQL or MySQL syntax please find database specific configurations in $ENGINE_HOME/demo_sales/ directory.

DEFINING CONNECTION TO YOUR "MASTER DATABASE"

Master database is usually target database where your ETL routines will insert/update processed data. Sales demo has Kimball style Data Mart defined in database schema named 'sh' (for StreamHorizon). First we should define connectivity to Master Database by altering $ENGINE_HOME/config/engine-config.xml file:

```
<connectionProperties>
        <jdbcUrl>jdbc:oracle:thin:@localhost:1522:orcl</jdbcUrl>
        <jdbcUserName>sh</jdbcUserName>
        <jdbcPassword>sh</jdbcPassword>
        <jdbcPoolSize>100</jdbcPoolSize>
</connectionProperties
```

CONFIGURE NEW ETL FLOW

To create new ETL flow/pipeline/stream we need to define new <feed> element within the config/engine-config.xml file. Every feed element must have unique name in config/engine-config.xml file. Feed element is child of <feeds> element. The <feeds> element is place where all ETL flows for given instance of StreamHorizon are defined. We will name our feed "myDataFeed1".

```
<feed name="myDataFeed1" type="full">
```

```
</feed>
```

CONFIGURE SOURCE TYPE

After new ETL flow is created we need to define source type. Source type can be any of Hadoop, Messaging, Relational, File, Thrift etc (and since StreamHorizon is pluggable you can create your own source types very easily). In our example we will use files as data source.

We first create <source> element which is child of <feed> element, then we assign type='file' which signifies that files are used as data input (rather than relational, Hadoop or any other source).

For file source type, StreamHorizon requires that directory where it will retrieve files from (file source directory) to be set up "directoryPath" as well as file mask which will identify feed files from given directory.

```
<source type="file">

<properties>

        <property name="directoryPath">/sh/data/source</property>

        <property name="fileNameMask">.*csv</property>

</properties>

</source>
```

CONFIGURE SOURCE FORMAT DEFINITION

After source type has been defined we will need to create "source format definition".

Source format definition defines structure (attributes) for given feed. In our case, source format definition defines header, data and footer attributes of the file. Header and footer are not mandatory.

If file was not to have header and footer it is easy to see how by simply changing "source type" and leaving "source format definition" as it is, we can reconnect StreamHorizon from file to relational or any other source type.

- "Source format definition" defines structure of data input
- "Source type" defines connectivity to data source (file, Hadoop, relational etc.)

Important parameters of source format definition:

- Source format definition element <sourceFormatDefinition> is child of <feed> element.

- The “delimiterString” element supplies delimiter used for parsing the file.
- The “header” identifies header, header always starts with same character “1”, note that this is not mandatory element and can be omitted. The header element also defines attributes which appear in header. In our case we have single header attribute named bookingDate.
- The “data” element defines actual body of the file. Most of file extracts do not have header and footer; rather they just have a data (or body).

```
<sourceFormatDefinition>
        <delimiterString>,</delimiterString>
        <header process="normal">
                <eachLineStartsWithCharacter>1</eachLineStartsWithCharacter>
                <attributes>
                        <attribute name="bookingDate" />
                </attributes>
        </header>
        <data process="no_validation">
                <attributes>
                        <attribute name="productName" />
                        <attribute name="productModel" />
                        <attribute name="productCategory" />
                        <attribute name="productCost" />
                        <attribute name="customerName" />
                        <attribute name="customerAddress" />
                        <attribute name="customerCountry" />
                        <attribute name="customerPhone" />
                        <attribute name="employeeNumber" />
                        <attribute name="employeeName" />
                        <attribute name="salesChannelName" />
```

```
                        <attribute name="promotionName" />
                        <attribute name="discountPCT" />
                        <attribute name="shipmentDate" />
                        <attribute name="salesDate" />
                        <attribute name="deliveryDate" />
                        <attribute name="priceBeforeDiscount" />
                        <attribute name="priceAfterDiscount" />
                        <attribute name="saleCosts" />
                        <attribute name="supplierName" />
                </attributes>
        </data>
</sourceFormatDefinition>
```

Notes:

- Neither header nor **eachLineStartsWithCharacter** are mandatory for data definition
- All attributes supplied in this case are treated as strings so no data type definition is required.

CONFIGURE TARGET TYPE

After source type and source format definition have been created we will define target type. Analogue to source type, target type defines connectivity of StreamHorizon which will be used when executing given ETL flow. In our case we will use JDBC connectivity to relational data source (again, it is possible to define your own target types by using StreamHorizon plugins). Our relational target in this example will be “Master Database” which we have already configured in one of previous sections.

Target type is defined by type="jdbc" to be JDBC, all that remains is to define sql query which will be called by StreamHorizon ETL threads to insert data into the target fact table. Defining sql query is defined as a command of sql type (<command type="sql">) within < bulkLoadInsert> element. The <bulkLoadInsert> can be considered as an event which StreamHorizon will raise/execute when processing each file. This ensures that all data is inserted into database.

```
<target type="jdbc">
        <bulkLoadInsert>
                <command type="sql">
insert  /*+ append_values */ into sales_fact subpartition (P_${bookingDate}_SP_${feedProcessingThreadID})
```

```
(product_id, customer_id, employee_id, supplier_id, sales_channel_id, promotion_id, booking_date_id,
sales_date_id, delivery_date_id, priceBeforeDiscount, priceAfterDiscount, saleCosts, sub, file_id) values
(?,?,?,?,?,?,?,${bookingDate},?,?,?,?,?,${feedProcessingThreadID},-1)

                </command>

        </bulkLoadInsert>

</target>
```

CONFIGURE TARGET TYPE (FOR BULK LOADING)

When using bulk loading utility instead of ETL threads doing reading of the file and inserting into database via JDBC, StreamHorizon runs two thread pools:

- ETL threadpool which process feed data files and creates bulk files
- DB threadpool which picks up bulk files and inserts them into database via adequate method (BULK INSERT for MSSQL, External Tables for Oracle etc.)

If we were to change previous JDBC example to work via Oracle External tables target type will need to define directory into which ETL threads will create files (and from which DB threads will read files). In addition we can define that bulk files created by the ETL threads have extension “data”. To insert data into the database we will use stored procedure “p_sh_external_table_load” to which we pass parameters “bulkProcessingThreadID” and “bulkFileName”. Target definition will in such case be:

```
<target type="file">

        <properties>

                <property name="bulkOutputDirectory">/sh/data/bulk</property>

                <property name="bulkLoadOutputExtension">data</property>

        </properties>

        <bulkLoadInsert>

                <command type="sql">

                CALL p_sh_external_table_load(${ffn[2]}, ${bulkProcessingThreadID},'${bulkFileName}', 0 )

                </command>

        </bulkLoadInsert>
```

```
</target>
```

PARALLELISM, GLOBAL & CUSTOM DEFINED VARIABLES, VARIABLE SUBSTITUTION

StreamHorizon will at startup print out global (implicitly defined) variables like bulkProcessingThreadID, feedProcessingThreadID and bulkFileName. Global variables enable you to gather metadata metrics about StreamHorizon performance, execution, error messages etc.

In two previous listings for JDBC and Bulk Load target type you have seen these variables being used to achieve massively parallel inserts into database. In JDBC example we use variables like **bookingDate** and **feedProcessingThreadID** to target right subpartition of Oracle table we are inserting data into. This enables us to achieve desired degree of parallel inserts. In case of Bulk Load we use bulkProcessingThreadID and **bulkFileName** as parameters which are passed to stored procedure as parameters.

In JDBC example ETL thread executes inserts. Variable **feedProcessingThreadID** parameter has unique value for every thread in your ETL threadpool, which is how we ensure that ETL threads can insert into database simultaneously.

In Bulk Load example data is inserted into database by DB threadpool, hence **bulkProcessingThreadID** is used (instead of **feedProcessingThreadID**) to ensure that DB threads can simultaneously insert data into the target table.

Note that apart from global variables defined by StreamHorizon engine every attribute you have defined in your "source format definition" like bookingDate; productName etc. becomes implicitly "custom variable". All variable names must be unique for single instance of StreamHorizon. User cannot define custom variable with a same name as already existing global variable defined by StreamHorizon.

Before executing <command> (shell or sql) Stream Horizon performs variable substitution, this ensures that line in configuration file:

```
CALL p_sh_external_table_load(${ffn[2]}, ${bulkProcessingThreadID},'${bulkFileName}', 0 )
```

Before execution against database becomes:

```
CALL p_sh_external_table_load(20140107, 4, myFileName.data, 0 )
```

Every variable (global or custom) defined within <command> element will be substituted for its value if enclosed within ${ and } characters respectively (as you can see in example above).

The "ffn[2]" is yet another type of variable which is defined by custom java code supplied in Sales Demo. This Java method extracts data from feed file name and makes it available during StreamHorizon processing. It is possible to provide plugin for customized parsing of feed file names and making parts of it available as custom context attributes.

CONTEXT

StreamHorizon global and custom defined variables are part of "context". Context is a registry where all variables are held and maintained (updated) during feed processing. For example as ETL thread reads new data

file it will update variable **feedInputFileName** to a new value to reflect the name of the file given ETL thread is processing. All variables and their values are stored in Context of StreamHorizon.

Context enables you to refer to data feed file attribute as feed.salesDate or salesDate. Prefixing attribute with "feed." is instruction to StreamHorizon not to process attribute but to rather simply map it to output as we will see shortly.

In few sections we will get introduced to dimensional caches, they can be referenced as "dimension.employee_dim" where "employee_dim" is name given to dimension. Dimensional names (cache names) are also attributes which are considered to be custom defined variables and which reside in context. They must also have unique name for a given StreamHorizon instance.

CREATING TARGET FORMAT DEFINITION

Analogue to source format definition, target format definition defines structure (attributes) of output format of ETL flow. Target format definition is supplied within < targetFormatDefinition> element. All variable values are mapped from Context variables of StreamHorizon engine.

From example below it is obvious that:

- All dimension.* attributes are keys of dimensions
- Other attributes are mapped directly as they appear in the source feed and prefixed with "feed." (note that "feed." Prefix can be omitted)

Knowing that insert statement, format of which we are defining with target format definition is actually insert statement into fact table it is obvious that all we map in fact tables are dimensional surrogate keys (id's), dates and measures (like saleCosts for example). This is typical for Kimball style fact table design.

```
<targetFormatDefinition>
        <attributes>
                <attribute name="dimension.employee_dim" type="int" />
                <attribute name="dimension.customer_dim" type="int" />
                <attribute name="dimension.product_dim" type="int" />
                <attribute name="dimension.sales_channel_dim" type="int" />
                <attribute name="dimension.promotion_dim" type="int" />
                <attribute name="dimension.supplier_dim" type="int" />
                <attribute name="bookingDate"  type="int" />
                <attribute name="feed.salesDate"  type="int" />
```

```
                <attribute name="feed.deliveryDate"  type="int" />

                <attribute name="feed.priceBeforeDiscount" type="float" />

                <attribute name="feed.priceAfterDiscount" type="float" />

                <attribute name="feed.saleCosts" type="float" />

        </attributes>

</targetFormatDefinition>
```

PARALLELISM, ETL & DB THREADPOOLS

Every data feed <feed> regardless of type of source and target feed can define:

- ELT threadpool
- DB Threadpool
- ETL and DB Threadpool

Few use cases:

- If you wish to process data from a file (as source) to the file (as target) all you need to define is ETL threadpool and let your DB threadpool be zero (have zero threads in the threadpool).
- If you wish to read data from file and JDBC insert into database all you need to define is ETL threadpool and let your DB threadpool be zero (have zero threads in the threadpool).
- Same as above applies for any other combination of source and target type (any connector) supported by StreamHorizon.
- If you wish to process file source feeds and insert them into database via bulk loader you need to configure ETL threadpool to process source files into bulk files. In addition you need to define DB threadpool to read bulk files and insert them into the database (as we have shown in example above when configuring target type for bulk loading).
- If you wish to run StreamHorizon instance in dedicate bulk loader mode you could arrange that other StreamHorizon instances (or any other software for that matter) creates bulk files for you in a designated directory. Then you can configure only DB threadpool (ETL threadpool has size zero) and StreamHorizon instance will just perform bulk load of bulk files into the database.

ETL & DB threadpools are separate pools and depending on balance of your CPU and I/O resources you will most likely want to run more ETL than DB threads or other way around. It is rare that when tuning system best overall throughput is achieved when ETL and DB threadpool have equal number of threads.

ELT STREAM - COMPLEX & INTEGRATED ETL FLOWS

Every <feed> element defined within <feeds> element represents one ETL flow. By defining multiple <feed> elements and ensuring that second feed has an input same as output of the first feed we can build ETL stream of any level of complexity.

If we have use case that we need to achieve the following:

1. Process input data files and insert them into database db1
2. Extract data from database (inserted in previous step) and generate output files
3. Extract data from database (inserted in previous step) and generate Hadoop extract files
4. Load Hadoop extracts into another database db2

We could solve problem by:

1. Define first feed which takes input data files and inserts them into db1 (either via jdbc or bulk load as inserting method, so we will configure both ETL and DB threadpools for bulk load inserts or only ETL threadpool in case of jdbc inserts)
2. Define second feed which will read data from database db1 (produced by first feed) and create output files. This can be achieved with feed which only use ETL threadpool.
3. Define third feed which will read data from database db1 (produced by first feed) and create Hadoop extract files. This can be achieved with feed which only use ETL threadpool.
4. Define fourth feed which will read data from Hadoop and load it into database db2 (either via jdbc or bulk load as inserting method, so we will configure both ETL and DB threadpools for bulk load inserts or only ETL threadpool in case of jdbc inserts)

DB THREADPOOL IS JUST A GENERIC POOL (OLAP INTEGRATION EXAMPLE)

DB Threadpool is named as “DB” because most of deployments utilize it that way. However, that threadpool is generic and can be used for any purpose. For example, after we have loaded data via bulk loader into the database (as previously elaborated in “Configure target type for bulk loading”) we could simply add another command into target type of type of “shell” which will call OLAP load executable and pass it file name which need be uploaded to OLAP cube. For OLAP integration please see Tech FAQ page of StreamHorizon website.

User can define any number of “sql” or “shell” commands which will be executed in order as given in <bulkLoadInsert> element.

```
<target type="file">
        <properties>
                <property name="bulkOutputDirectory">/sh/data/bulk</property>
                <property name="bulkLoadOutputExtension">data</property>
        </properties>
```

```
        <bulkLoadInsert>
                <command type="sql">
                CALL p_sh_external_table_load(${ffn[2]}, ${bulkProcessingThreadID},'${bulkFileName}', 0 )
                </command>
                <command type="shell">
                /application/OLAP/loadIntoOlap.sh ${bulkFileName}
                </command>
        </bulkLoadInsert>
</target>
```

DEFINING ETL & DB THREADPOOL SIZE

ETL and DB threadpools are defined in < threadPoolSettings> element, child of <feed> element. Best starting point is to configure number of ETL and DB threads to match number of free CPU's on your server. Try increasing/decreasing values for each pull in steps of 20% until you reach sweet spot of the system. For more performance tuning related topics please refer to "Performance Tuning" chapter.

Example below sets ETL treadpool to size of 24 (24 ETL threads) and size of DB threadpool to zero (no DB threads). This mode of operation is used when reading flat files and inserting them into database via jdbc or any other mode of operation when no bulk load into database is required.

```
<threadPoolSettings>
        <etlProcessingThreadCount>24</etlProcessingThreadCount>
        <databaseProcessingThreadCount>0</databaseProcessingThreadCount>
</threadPoolSettings>
```

ERRORS & ARCHIVING

In case of exceptions occurring while processing (either in ETL or in DB thread) StreamHorizon will move given data file into error directory defined <errorDirectory> element. All errors are logged in StreamHorizon engine logs.

When successfully processed (by either ETL or DB thread) files will be moved into archive directory <archiveDirectory> from source directory.

Bulk files are deleted upon successful completion (insertion into database).

EVENTS

Both ETL and DB threadpools have events which are executed by every ETL and DB thread during processing of every data entity (file, message, sql query). Both threadpools have events which will be executed upon success, failure or “finally” (always executed, regardless of processing status (success or failure)).

Additional events are events raised before processing of data entity starts and event raised at startup of StreamHorizon instance. Event <onStartupCommands> should contain commands of sql and shell type which will restore database, files, any other data resource and StreamHorizon to consistent state in case of power outage or any other unpredictable infrastructure (or programmatic) failure.

Events accept <command> elements, they enable you to invoke sql or shell (sql statements and commands/executables) after ELT or DB thread has finished processing data entity (file, message etc.). This is very useful feature if you need to react with cleanup logic in case of failure of if you want to simply gather metrics when particular feed has been processed etc. All metrics (sh* views) which come with Sales Demo of StreamHorizon utilize event architecture to log performance metrics into the metric table sh_metric. All metric views are based on this single table which stores execution metadata of StreamHorizon engine.

	ETL Thread events	**DB Thread events**	**Startup event**
Before Processing	<beforeFeedProcessing>	<beforeBulkLoadProcessing >	N/A
On Success	<afterFeedSuccess>	<afterBulkLoadSuccess>	N/A
On Failure	<afterFeedProcessingFailure>	<afterBulkLoadFailure>	N/A
Finally (executes always)	<afterFeedProcessingCompletion>	<afterBulkLoadCompletion>	N/A
At StreamHorizon Instance Startup	N/A	N/A	<onStartupCommands>

To configure your ETL and DB threads to log processing metrics you could implement finally type events for both ETL and DB threads. Events will execute stored procedure calls against database and pass appropriate metadata (StreamHorizon context variables) as parameters. This example is taken from Oracle external table load configuration of Sales Demo.

```
<events>

        <onStartupCommands>
```

```
                <command type="sql">truncate table sales_fact</command>

                <command type="sql">truncate table sales_fact_agg</command>
                <command type="sql">truncate table sh_metrics</command>

        </onStartupCommands>

        <afterBulkLoadCompletion>

                <command type="sql">

CALL
log_sh_metrics_bulk('server_1',${engineInstanceIdentifier},${engineInstanceStartTimestamp},'afterBulkLoadC
ompletion','${bulkFileName}'
,${bulkFileProcessingStartedTimestamp},${bulkFileProcessingFinishedTimestamp},
'${bulkCompletionProcessingSuccessFailureFlag}', '${bulkCompletionProcessingErrorDescription}',
${bulkProcessingThreadID})

                </command>

        </afterBulkLoadCompletion>

        <afterFeedProcessingCompletion>

                <command type="sql">

CALL
log_sh_metrics('server_1',${engineInstanceIdentifier},${engineInstanceStartTimestamp},'afterFeedProcessingC
ompletion',${feedInputFileReceivedTimestamp},${feedProcessingThreadID},'${feedInputFileName}',${feedInput
tFileProcessingStartedTimestamp},${feedInputFileProcessingFinishedTimestamp},${feedInputFileJdbcInsertSta
rtedTimestamp},${feedInputFileJdbcInsertFinishedTimestamp},'${bulkFileAlreadySubmittedForLoading}',${bulk
ProcessingThreadID},'${bulkFilePath}','${bulkFileName}',${feedCompletionNumberOfTotalRowsInFeed},${bulkF
ileReceivedForProcessingTimestamp},${bulkFileProcessingStartedTimestamp},${bulkFileProcessingFinishedTim
estamp},'${feedCompletionProcessingSuccessFailureFlag}','${feedCompletionProcessingErrorDescription}')

                </command>

        </afterFeedProcessingCompletion>

</events>
```

END OF ETL FEED DEFINITION

Utilisation of events was last piece of configuration of ETL flow <feed> element. As mentioned previously, ETL flows can be made into ETL Stream by connecting multiple feed <feed> ETL flows in suitable pipeline.

CREATING DIMENSIONAL CACHES

When loading data into the database (data mart) we often have use case when child table requires attribute which is ID of its parent table. For example sales table sales_fact which stores orders will have keys to parent tables like product_dim (product table/dimension) or customer dimension.

Dimensional caches are essentially lookups which cache particular dimension. For example, for product dimension we may have product_name and product_id as table attributes. As file feed contains productName attribute we need to match incoming file feed attribute with value of table attribute product_name. If match is successful we need to use product_id during the insert of data into fact table (table of customer orders).

Definition of dimensional cache is achieved by:

- Assign dimension name which must be unique in StreamHorizon Context (name="customer_dim")
- Assign type of dimension which can be INSERT_ONLY, TYPE1, TYPE2 or Custom (type="INSERT_ONLY")
- Define how record should be matched (define natural key), in our case all table attributes belong to natural key and will have setting (naturalKey="true")
- Define <insertSingleRecord> sql statement which is invoked if StreamHorizon needs to insert new product which isn't yet available in the Master Database (target database).
- Define <selectRecordIdentifier> sql statement to retrieve single ID for any arbitrary natural key. This is executed when cache need be populated (and after <insertSingleRecord> statement has failed due to unique index constraints)
- Optional: Define <preCacheRecords> if you wish StreamHorizon to cache all your dimensional data during startup.

```
<dimension name="product_dim" type="INSERT_ONLY">
        <mappedColumns>
                <mappedColumn name="productName" naturalKey="true" />
                <mappedColumn name="productModel" naturalKey="true" />
                <mappedColumn name="productCategory" naturalKey="true" />
                <mappedColumn name="productCost" naturalKey="true" />
        </mappedColumns>
        <sqlStatements>
                <insertSingleRecord>
insert into product_dim(product_id, product_name, product_model, product_category, product_cost) values
(product_dim_seq.nextval, '${productName}', '${productModel}', '${productCategory}', '${productCost}')
                </insertSingleRecord>
```

```
                    <selectRecordIdentifier>

select product_id from product_dim where product_name='${productName}' and
product_model='${productModel}' and product_category='${productCategory}' and
product_cost='${productCost}'

                    </selectRecordIdentifier>

                    <preCacheRecords>

select product_id, product_name, product_model, product_category, product_cost from product_dim

                    </preCacheRecords>

          </sqlStatements>

</dimension>
```

Dimensional caches are utilized in target format definition to map output data of ETL flow (for more detail please refer to previous sections of this chapter).

CUSTOMIZING STREAMHORIZON PROPERTIES

For full list of StreamHorizon properties please refer to “General properties” and “Performance tuning “ chapters. It is worth mentioning that:

In case that you use JDBC mode of insert you should (to start with) use JDBC batch size of 2000 records for MSSQL and 10000 records for Oracle

<property name="jdbc.bulk.loading.batch.size">2000</property>

In case that your filesystem cannot guarantee I/O write atomicity you should set bulk file acceptance to value higher than 0 milliseconds. This instructs DB threads to read file only after it hasn’t been written into for supplied number of milliseconds.

<property name="bulk.file.acceptance.timeout.millis">0</property>

Other useful parameters are setting read & write buffer size which help best utilisation of your I/O resources and many others. Please refer to “General properties” and “Performance tuning “ chapters.

END OF SHORT INTRODUCTION TO STREAMHORIZON

This chapter had a purpose to give you flavour of StreamHorizon platform, the way it operates and to introduce you to StreamHorizon terminology. Please refer to other documentation if you wish to understand in more detail all available features of StreamHorizon.