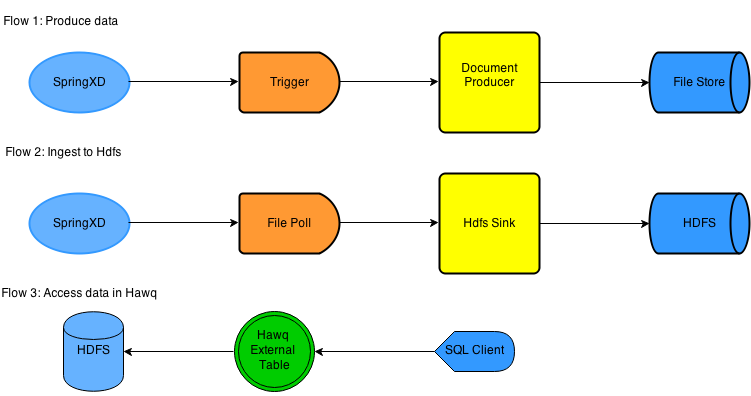
# Streaming Binary data in HAWQ.

Pivotal offers HAWQ, SQL-on-Hadoop engine. HAWQ is designed as a MPP SQL processing engine optimized for analytics with full transaction support. HAWQ breaks complex queries into small tasks and distributes them to MPP query processing units for execution. The query planner, dynamic pipeline, leading edge interconnect, and the specific query executor optimization for distributed storage are designed to work seamlessly, to support highest level of performance and scalability. Based on Hadoop’s distributed storage, HAWQ has no single point of failure and supports fully automatic online recovery. System states are continuously monitored, therefore if a segment fails; it is automatically removed from the cluster. During this process, the system continues serving customer queries, and the segments can be added back to the system when necessary. To learn about HAWQ please visit <http://www.pivotal.io/big-data/pivotal-hd>.

HAWQ PXF feature let us access information from HDFS and access the data in ANSI SQL on all the primitive data types. PXF can query data from support HBase, Hive, Sequence files, csv files stored in HDFS. Let us examine a scenario where we are streaming in a sequence file in HDFS and expose the data via HAWQ external table.

To demonstrate the scenario Let us generate some sequence files, store them in HDFS and query the via HAWQ external table feature. The flow we are going to use is as below;



As shown in the diagram the process is broken 3 steps; data producer, data ingestion, query step.

We are going to use SpringXD for data ingestion. To setup SpringXD please refer to <http://docs.spring.io/springxd/docs/1.0.0.RELEASE/reference/html/#_install_spring_xd> to setup SpringXD in single node environment and configure HADOOP FS in SpringXD.

Flow 1, usually an external source sending the files to a file store. To simulate the flow, I choose to use SpringXD to trigger the Document Generator process for a fixed interval time; which generates a sequence file containing a record with three fields name, type, content. The name and type are text type and content is a byte array representing the document. The file will be stored in a folder. The process is configured to accept the folder location and number of records would like to generate in a file.

Flow2 is another SpringXD stream which uses out of box File polling process and a custom Sink to store files to HDFS. This process polls the folder for fixed interval time and if there is any new file, moves to HDFS.

The final step is creating an external table in HAWQ to query the data in client Application.

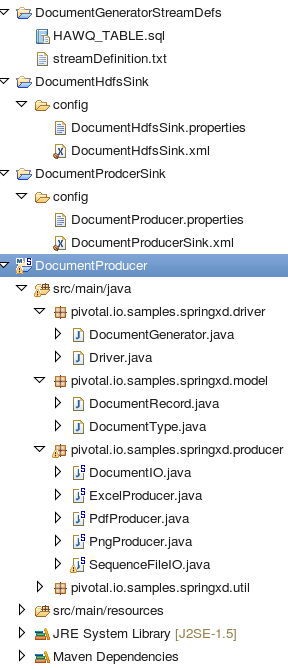
The code for this sample can be downloaded at <https://github.com/sridharpaladugu/SpringXD_Pivotal_hadoop/tree/master/SpringXD_HAWQ_SEQFILE_DEMO>

The Java code is in folder DocumentProducer,

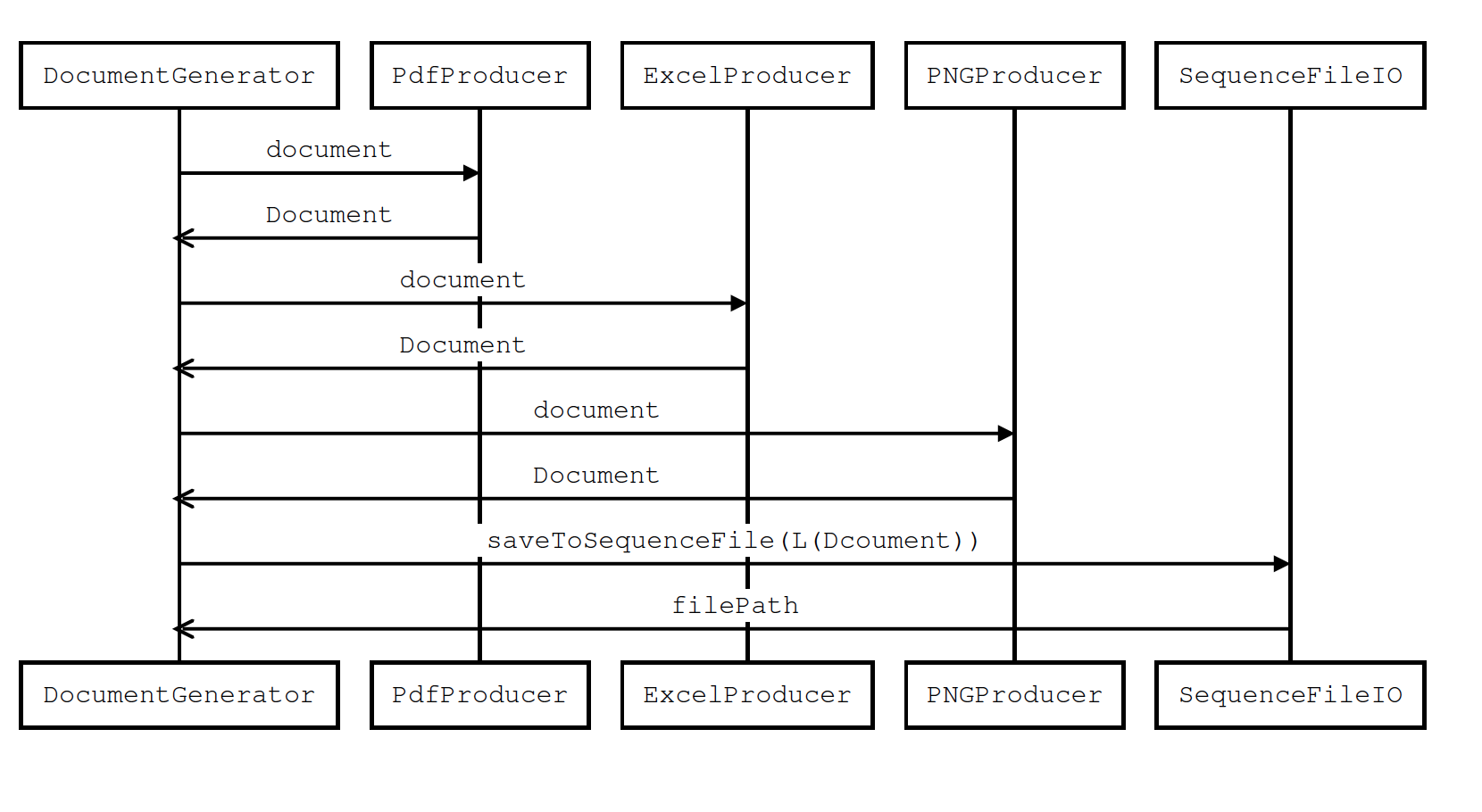
SpringXD module definitions are in DocumentProdcerSink DocumentHdfsSink folders,

HAWQ table definition and SpringXD streams are in folder DocumentGeneratorStreamDefs.

Clone and import the DocumentProducer in to Eclipse with import existing maven projects option. The code is organized as follows.



The Document generation process flow generates preconfigured number pdf, excel, and image records in a sequence file. To keep things simple I have a Driver class which instantiate the 3 beans and generate number of specified document records. Each of the service call result in a Document POJO with text attributes ‘name, type, and a byte[] representing the document. The high level flow is as below;



The service accepts two parameters dropBoxLocation to store the output file, and the size of the records to produce in each file.

Run maven build for DocumentProducer and deploy the jar in SpringXD. Deploying to springXD is, copy the jar to either a configured EXT folder or simply dropping the jar in the folder ${SPRINGXD\_HOME }/xd/lib/. Also copy the dependent jars itext, jxl from maven path to xd lib.

We need to generate xml configuration for the Document generation sink module. For this purpose we use spring integration service activator.

Copy the folders DocumentProdcerSink and DocumentHdfsSink. to ${SPRINGXD\_HOME}/xd/modules/sink.

Once we have all the pieces in place, please restart spring xd and verify we have the modules deployed. For example;

**XD>module info sink: DocumentHdfsSink**

Information about sink module DocumentHdfsSink:

Option Name Description Default Type

-------------- ----------------------------------------------------- ---------------------- ---------------------------------------------------------------------------

fsUri Hadoop FS URL ${spring.hadoop.fsUri} String

hdfsDirectory hdfs file location /user/gpadmin String

localDirectory location of the file in local file system /tmp String

inputType how this module should interpret messages it consumes <none> MimeType

**xd:>module info sink: DocumentProdcerSink**

Information about sink module DocumentProdcerSink:

Option Name Description Default Type

--------------- -------------------------------------------------------- ------- -------------------------------------------------------------------------

recordSize number of records in sequence file 200 Integer

dropboxLocation location of the folder where the sequence file is stored /tmp String

inputType how this module should interpret messages it consumes <none> MimeType

Now let us create a stream and deploy to generate the files. The following command create a stream module in SpringXD, which runs the Document generation process with fixed interval 0f 20 seconds.

**Xd>stream create --name generateSeqFiles --definition "trigger --fixedDelay=20 | DocumentProdcerSink --dropboxLocation=/home/gpadmin/TEST\_STAGE --recordSize=10"**

Deploy the stream to SpringXD.

**xd>stream deploy generateSeqFiles**

After this step we expect to see documents getting generated in the folder “/home/gpadmin/TEST\_STAGE”.

Since our source is up and streaming files, let us create a stream to store files to HDFS. Please create the destination folder in HDFS before this step.

**xd:>stream create --name storeDocumentsToHdfs --definition "file --dir=/home/gpadmin/TEST\_STAGE/ --pattern=\*\*/\*.bin --fixedDelay=10 --ref=true | DocumentHdfsSink --localDirectory=/home/gpadmin/TEST\_STAGE/ --hdfsDirectory=/user/hawqtest/hdfssink"**

Deploy the stream;

**xd>stream deploy DocumentHdfsSink**

Once the stream starts up we should see files in hdfs in the folder “/user/hawqtest/hdfssink”.

[pivhdsne:sink]$ hdfs dfs -ls /user/hawqtest/hdfssink

Found 81 items

drwxr-xr-x - gpadmin hadoop 0 2014-08-11 23:07 /user/hawqtest/hdfssink/\*.bin

-rw-r--r-- 3 gpadmin hadoop 36994 2014-08-11 23:07 /user/hawqtest/hdfssink/SPXDocuments\_Aug-11-2014T23-00-17.bin

-rw-r--r-- 3 gpadmin hadoop 37320 2014-08-11 23:07 /user/hawqtest/hdfssink/SPXDocuments\_Aug-11-2014T23-00-30.bin

-rw-r--r-- 3 gpadmin hadoop 36917 2014-08-11 23:07 /user/hawqtest/hdfssink/SPXDocuments\_Aug-11-2014T23-00-43.bin

-rw-r--r-- 3 gpadmin hadoop 37176 2014-08-11 23:07 /user/hawqtest/hdfssink/SPXDocuments\_Aug-11-2014T23-00-56.bin

…………

Let us look at the specifics of creating the sequence file. From the sequence diagram from we see that DocumentGenerator calls document method on the individual generators and assemble a collection of DocumentRecord POJOs. This list is passed to SeqeunceFileIO to persist the file. SequenceFileIO uses DocumentRecordPOJO as the value field while persisting the record in the file. For example;

SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path, Text.**class**, DocumentRecord.**class**);

**for** (DocumentRecord record : records) {

Text key = **new** Text(UUID.*randomUUID*().toString());

writer.append(key, record);

}

Similarly while retrieving the data we use same Document POJO for reading value out of stream.

SequenceFile.Reader reader = **new** SequenceFile.Reader (fs, p, conf);

Text key = (Text) ReflectionUtils.*newInstance*(reader.getKeyClass(), conf);

DocumentRecord documentRecord = (DocumentRecord) ReflectionUtils.*newInstance*(reader.getValueClass(), conf);

**try** {

**while** (reader.next(key, documentRecord)) {

returnData.add(documentRecord);

}

} **catch** (EOFException e) {***…***}

HAWQ PXF uses the same Document POJO while reading the data form external table. PXF uses reflection to set the fields on the POJO and require that we declare the fields as public. Also it supports only primitive types and arrays of primitive types. While declaring arrays we should hint the size of the field using the constructor and also need to use the size of the array while reading and the data out/in from the Stream.

For example, in our DocumentRecord looks as below;

public void readFields(DataInput in) throws IOException {

name = in.readUTF();

type = in.readUTF();

size = in.readInt();

content = new byte[size];

in.readFully(content,0,size);

size = content.length;

}

public void write(DataOutput out) throws IOException {

out.writeUTF(name);

out.writeUTF(type);

out.writeInt(size);

out.write(content, 0, size);

}

By now we should have accumulated enough data in HDFS folder. So let us go ahead and create an external table in HAWQ. In order to create the table with the schema DocumentRecord, we should add the jar file to class path. So let us add the jar file that we built earlier to HAWQ class path. Copy the jar to /usr/lib/gphd/public folder.

In my setup this looks as;

[pivhdsne:sink]$ ls -l /usr/lib/gphd/publicstage/

total 36

-rw-r--r-- 1 gpadmin gpadmin 10096 Aug 8 07:13 CustomResolver-0.0.1-SNAPSHOT.jar

-rw-r--r-- 1 gpadmin gpadmin 22902 Aug 10 10:36 **DocumentProducer-1.0.0-SNAPSHOT.jar**

Also we should append this to hadoop-env.sh. In my setup it looks as;

# Required for PXF with HDFS

HADOOP\_CLASSPATH=$HADOOP\_CLASSPATH:\

…………….

**$GPHD\_HOME/publicstage/DataProducer-1.0.0-SNAPSHOT.jar:\**

…………….

Once the jar is added to class path, restart the server. The process is documented in HAWQ PXF document. The link is provided in the references section.

Now, let us launch psql and type the below command.

gpadmin=# create external table ext\_hdfssink(name text, type text, content bytea)

location('pxf://pivhdsne:50070/user/hawqtest/hdfssink/\*.bin?fragmenter=com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter&accessor=com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor&resolver=com.pivotal.pxf.plugins.hdfs.WritableResolver&data-schema=pivotal.io.samples.springxd.model.DocumentRecord')

format 'custom' (formatter='pxfwritable\_import');

Verify the table;

gpadmin=# \d+ ext\_hdfssink;

External table "public.ext\_hdfssink"

Column | Type | Modifiers | Storage | Description

---------+-------+-----------+----------+-------------

name | text | | extended |

type | text | | extended |

content | bytea | | extended |

Type: readable

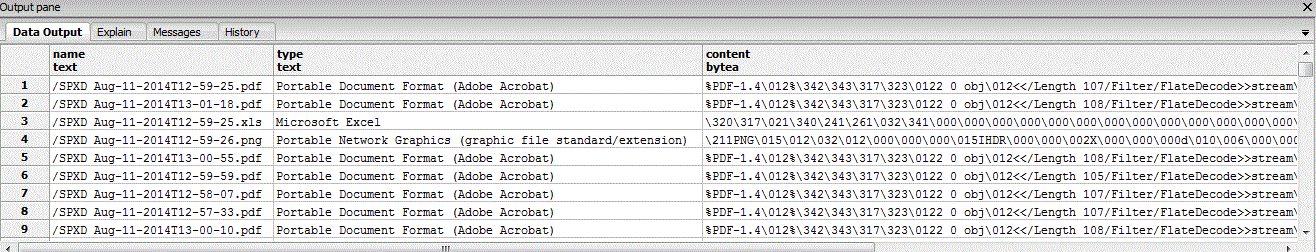
Encoding: UTF8

Format type: custom

Format options: formatter 'pxfwritable\_import'

External location: pxf://pivhdsne:50070/user/sri/hdfssink/\*.bin?fragmenter=com.pivotal.pxf.plugins.hdfs.HdfsDataFragmenter&accessor=com.pivotal.pxf.plugins.hdfs.SequenceFileAccessor&resolver=com.pivotal.pxf.plugins.hdfs.WritableResolver&data-schema=pivotal.io.samples.springxd.model.DocumentRecord

Select \* from ext\_hdfssink;



References:

<http://pivotalhd.docs.pivotal.io/doc/2010/PXFExternalTableandAPIReference.html#PXFExternalTableandAPIReference-ExternalTableExamples>

<http://pivotalhd.docs.pivotal.io/doc/2010/PXFInstallationandAdministration.html#PXFInstallationandAdministration-Recordkeyinkey-valuefileformats>

<http://blog.pivotal.io/pivotal/products/json-on-hadoop-example-for-extending-hawq-data-formats-using-pivotal-extension-framework-pxf>