Big Data and Hadoop with z/VSE

Last formatted on: Monday, May 18, 2015

Ingo Franzki
ifranzki@de.ibm.com

Disclaimer

Trademarks

Comments and Questions

Comments or questions on this documentation are welcome. Please send your comments to:

zvse@de.ibm.com

Contents

1. Introduction

This document describes how to use Big Data analysis and Hadoop with z/VSE. Anything described here can be used with any supported z/VSE version (i.e. z/VSE V5.1 or V5.2).

1.1. Apache Hadoop

Apache Hadoop is an open source software project that enables distributed processing of large data sets across clusters of commodity servers. It is designed to scale up from a single server to thousands of machines, with very high degree of fault tolerance. Rather than relying on high-end hardware, the resiliency of these clusters comes from the software's ability to detect and handle failures at the application layer.

The project includes these modules:

- **Hadoop Common:** The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS):** A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN:** A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets.

1.2. MapReduce

MapReduce is the heart of Hadoop. It is this programming paradigm that allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster. The MapReduce concept is fairly simple to understand for those who are familiar with clustered scale-out data processing solutions.

For people new to this topic, it can be somewhat difficult to grasp, because it’s not typically something people have been exposed to previously. If you’re new to Hadoop’s MapReduce jobs, don’t worry: we’re going to describe it in a way that gets you up to speed quickly.

The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform. The first is the **map job**, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). The **reduce job** takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job.

An example of MapReduce

Let’s look at a simple example. Assume you have five files, and each file contains two columns (a key and a value in Hadoop terms) that represent a city and the corresponding temperature recorded in that city for the various measurement days. Of course we’ve made this example very simple so it’s easy to follow. You can imagine that a real application won’t be quite so simple, as it’s likely to contain millions or even billions of rows, and they might not be neatly formatted rows at all; in fact, no matter how big or small the amount of data you need to analyze, the key principles we’re covering here remain the same. Either way, in this example, city is the key and temperature is the value.

```
Toronto, 20
Whitby, 25
New York, 22
Rome, 32
```

```
Toronto, 4
Rome, 33
New York, 18
```

Out of all the data we have collected, we want to find the maximum temperature for each city across all of the data files (note that each file might have the same city represented multiple times). Using the MapReduce framework, we can break this down into five map tasks, where each mapper works on one of the five files and the mapper task goes through the data and returns the maximum temperature for each city. For example, the results produced from one mapper task for the data above would look like this:

```
(Toronto, 20) (Whitby, 25) (New York, 22) (Rome, 33)
```

Let’s assume the other four mapper tasks (working on the other four files not shown here) produced the following intermediate results:

```
(Toronto, 18) (Whitby, 27) (New York, 32) (Rome, 37)(Toronto, 32)
(Whitby, 20) (New York, 33) (Rome, 38)(Toronto, 22) (Whitby, 19)
(New York, 20) (Rome, 31)(Toronto, 31) (Whitby, 22) (New York, 19)
(Rome, 30)
```

All five of these output streams would be fed into the reduce tasks, which combine the input results and output a single value for each city, producing a final result set as follows:

```
(Toronto, 32) (Whitby, 27) (New York, 33) (Rome, 38)
```

As an analogy, you can think of map and reduce tasks as the way a census was conducted in Roman times, where the census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital city. There, the results from each city would be reduced to a single count (sum of all cities) to determine the overall population of the empire. This mapping of people to cities, in parallel, and then combining the results (reducing) is much more efficient than sending a single person to count every person in the empire in a serial fashion.

1.3. IBM InfoSphere BigInsights

IBM BigInsights for Apache Hadoop is capable of collecting and economically storing a very large set of highly variable data. IBM BigInsights enhances open source Hadoop with the complete set of capabilities to query, visualize, explore data and conduct distributed machine learning at scale resulting in deeper insight and better actions.

IBM BigInsights for Apache Hadoop enhances Hadoop to support the demands of your enterprise, adding administrative, workflow, provisioning and security features, along with sophisticated analytical capabilities from IBM Research. The result is a more developer- and user-friendly solution for complex, large-scale analytics.

Besides the 4 Hadoop core components (Hadoop Common, HDFS, YARN and MapReduce), IBM BigInsights for Apache Hadoop also contains the following projects and components:

- **Data access projects:**
  - **Pig:** A programming language designed to handle any type of data, helping users to focus more on analyzing large data sets and less on writing map programs and reduce programs.
  - **Hive:** A Hadoop runtime component that allows those fluent with SQL to write Hive Query Language (HQL) statements, which are similar to SQL statements. These are broken down into MapReduce jobs and executed across the cluster.
  - **Flume:** A distributed, reliable and available service for efficiently collecting, aggregating and moving large amounts of log data. Its main goal is to deliver data from applications to the HDFS.
  - **HCatalog:** A table and storage management service for Hadoop data that presents a table abstraction so the user does not need to know where or how the data is stored.
  - **Jaql:** A query language designed for JavaScript Object Notation (JSON), which is primarily used to analyze large-scale semi-structured data. Core features include user extensibility and parallelism.
  - **Avro:** An Apache open source project that provides data serialization and data exchange services for Hadoop.
  - **Spark:** An open-source cluster computing framework with in-memory analytics performance that is up to 100 times faster than MapReduce, depending on the application.
  - **Sqoop:** An ELT tool to support the transfer of data between Hadoop and structured data sources.
  - **HBase:** A column-oriented non-relational (noSQL) database that runs on top of HDFS and is often used for sparse data sets.
- **Search projects:**
  - **Solr:** An enterprise search tool from the Apache Lucene project that offers powerful search capabilities, including hit highlighting, as well as indexing capabilities, reliability and scalability, a central configuration system, and failover and recovery.
- **Administration and security projects:**
  - **Kerberos:** A network authentication protocol that works on the basis of "tickets" to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner.
  - **Zookeeper:** A centralized infrastructure and set of services that enable synchronization across a cluster.

IBM InfoSphere BigInsights extends open-source components with value-added capabilities that customers can choose to take advantage of without compromising on openness or adherence to standards:

Best-in-class SQL-on-Hadoop

- **Big SQL:** delivers unmatched simplicity, performance and standards compliance. Unlike other SQL-on-Hadoop implementations, Big SQL works with what you have. It runs against native Hadoop data sources and provides federated access to third-party databases, preserving your investments in tools, applications and expertise.

  Big SQL leverages IBM's strength in SQL engines to provide ANSI SQL access to data across any system from Hadoop, via JDBC or ODBC - seamlessly whether that data exists in Hadoop or a relational data base. This means that developers familiar with the SQL programming language can access data in Hadoop without having to learn new languages or skills.

  With Big SQL, all of your big data is SQL accessible. It presents a structured view of your existing data, using an optimal execution strategy, given your available resources. You can leverage MapReduce parallelism when needed for complex data sets and avoid it when it hinders, using direct access for smaller, and low-latency queries.

  Let's take a look at the different types of queries you can encounter with Big SQL:
  - **"Point queries":** These are queries that need to return very fast, like HBase queries, for example. In these types of queries, you cannot use MapReduce.
  - **Big ad-hoc queries:** In larger, more complex jobs MapReduce parallelism becomes very important to be able to break down these massive data sets.
  - **Standards-compliant via JDBC:** This is how most applications access databases and in this usage pattern, you can use the same to access your Hadoop-based data store.

Easy-to-use tooling for business users

- **BigSheets:** a web-based analysis and visualization tool with a familiar, spreadsheet-like interface and rich graphing capabilities. Non-technical users can load, filter, analyze and visualize large datasets from both in and out of Hadoop, boosting productivity and avoiding the need for programming or scripting.
- **Data exploration with Watson Explorer:** When open-source tooling is not enough, BigInsights Enterprise Edition extends the capabilities of Hadoop with IBM Watson Explorer, combining content and data from many systems throughout the enterprise and presenting it to users via a single, intuitive interface.

Built-in advanced analytics—descriptive, predictive, prescriptive

- **IBM InfoSphere BigInsights Big R:** Big R enables data scientists to use the popular R language to explore, visualize, transform and model big data right from within the R environment without the need to program using MapReduce.
- **Text analytics:** A sophisticated text analytics capability unique to BigInsights allows developers to easily build high-quality applications able to process text in multiple written languages, and derive insights from large amounts of native textual data in various formats.
- **Social Data Analytics:** BigInsights provides the capability to ingest and process large volumes of social media data from various sources. A ready-to-use Twitter data feed is included with select configurations of IBM BigInsights on Cloud to help organizations get productive with social data quickly.
- **Machine Data Analytics:** BigInsights provides the capability to ingest and process large volumes of machine data from sources such as system log files, sensor data, GPS devices and more. Data scientists can easily apply advanced machine learning algorithms to collected data seamlessly from within the R language environment.

Accelerators that speed time to value

- **Application accelerators:** Whether the data being analyzed includes text, machine data or social data, pre-written accelerators included in BigInsights help organizations realize value more quickly by leveraging pre-written application components for a variety of common big data use cases.
- **Rich development tools:** Developers can quickly develop and deploy big data applications from within the familiar Eclipse interface. Pre-built wizards and numerous implementation examples help speed development and improve application quality, enabling applications to be deployed to the BigInsights console from within Eclipse.

Performance optimized

- **Adaptive MapReduce:** Adaptive MapReduce is a drop-in replacement for Apache MapReduce that can be optionally enabled in BigInsights Enterprise Edition. It provides high-performance scheduling and flexible management of MapReduce workloads. In an independently audited report, BigInsights with Adaptive MapReduce was found to deliver on average four times the application performance compared to the open-source MapReduce.
- **Blistering-fast SQL:** Big SQL provides both strict SQL language compliance as well as exceptional performance. Users can access native Hadoop sources using their choice of tools, including Hive, or they can choose to use Big SQL on the same datasets leveraging its native high-performance MPP query engine. In an audited result, Big SQL was shown to outperform Hive on a standard set of queries by a factor of five.
- **Faster Processing of Streaming Data:** For clients needing fast and reliable processing of real-time data feeds, a limited-use license of InfoSphere Streams included in BigInsights extends open-source Hadoop delivering faster and more efficient processing of streaming data.

Enterprise-grade management

- **Management console:** A comprehensive web-based interface included in BigInsights simplifies cluster management, service management, job management and file management. Administrators and users can share the same interface, launching applications and viewing a variety of configurable reports and dashboards.
- **Security built-in:** InfoSphere BigInsights was designed with security in mind, supporting Kerberos authentication and providing data privacy, masking and granular access controls with auditing and monitoring functions to ensure that the environments stays secure.
- **Fault-tolerant POSIX file system:** GPFS FPO, included in BigInsights Enterprise Edition, provides an optionally deployable POSIX file system fully compatible with HDFS. GPFS allows both Hadoop and non-Hadoop applications to share the same file system avoiding replicated data, reducing costs, and simplifying workflows that frequently copy data in and out of Hadoop. Users can take advantage of enterprise-grade features like snapshots, off-site block replication and hierarchical storage management enabling infrequently accessed data to be transparently migrated to lower-cost storage tiers.

Seamless data integration

- **Code-less Integration of data:** IBM DataStage for BigInsights Enterprise Edition enables code-less creation of data integration logic and jobs, reusable across the enterprise. Enable data governance including data lineage, business rule and policy management and data quality.

- **A Unified view of data:** Unified view of all data-driven information, including on Hadoop, for a comprehensive, contextually-relevant view powered by Watson Explorer.
- **Probabilistic matching with Big Match:** When integrating data from multiple sources matching like data quickly emerges as a major challenge. Available as an optional add-on to BigInsights, InfoSphere Big Match for Hadoop uses statistical learning algorithms and probabilistic matching to provide fast and efficient linking of data sources for more complete and accurate information.

1.4. The z/VSE Connectors

The z/VSE Connectors are part of the z/VSE operating system (component IDs 5686-CF9-35 and 5686-CF9-38). They enable you to integrate z/VSE data and applications into a distributed environment.

The z/VSE Connectors consist of several components that support different connectivity scenarios.

Java-based Connector

Enables you to write Java applications that access z/VSE data and functionality. You can have realtime access to z/VSE resources (like VSE/VSAM, VSE/POWER, DL/I, Librarian, VSE/ICCF, console) from remote platforms.

The Java-based connector consists of two parts:

- The **z/VSE Connector Client** provides the z/VSE Java Beans class library, online documentation and programming reference (JavaDoc), and many samples including Java source code for writing Web applications like applets, servlets, Enterprise Java Beans (EJBs) etc.
- The **z/VSE Connector Server** (part of VSE/ESA 2.5 or later releases) is running on z/VSE and implements native access methods to VSE/VSAM, DL/1, Librarian, VSE/POWER, ICCF, allowing you to submit jobs, and access the z/VSE operator console.

VSAM Redirector

The VSAM Redirector allows you to redirect all accesses to a certain VSAM file into any other file system or database on any other (Java-enabled) platform.

The VSAM Redirector connector of the following parts:

- The **VSAM Redirector Client** is an exit program running on z/VSE. It is called for each VSAM request and forwards it to the VSAM Redirector Server.
- The **VSAM Redirector Server** is running on a Java enabled platform. If receives the VSAM requests and translates them into SQL queries or requests to flat files. The translation of the requests is performed by so called **Handlers**.
- The **VSAM Redirector Loaders** is running on a Java enabled platform. If allows you to perform mass data load, such as an initial load an entire VSAM cluster into a database.
- The VSAM Capture Exit captures the changes performed by applications and stores them into a so called delta file or into an MQ queue. The changes can the later be applied to a database (via the Delta Loader).

You redirect access to a VSAM file by setting up the Redirector's config table, which specifies a list of VSAM files and their new locations.

z/VSE Script Server

The z/VSE Script Server provides access to z/VSE resources using the Java based connectors, but without writing Java programs. The z/VSE Script Server is used to execute so called z/VSE scripts. z/VSE scripts are written in a simple script language. It provides statements like if, while, for to control the program flow. z/VSE scripts can use various script commands to access z/VSE resources.

A z/VSE Script can be invoked by any kind of programs (especially non-Java Programs). The z/VSE Script Server package contains samples to invoke a z/VSE script from office applications like Microsoft Excel or

Lotus 1-2-3. The samples shows how to read a VSAM record and include it into a spread sheet. You can also call a z/VSE Script from a z/VSE program (batch or CICS). This allows you to run processes on the distributed side (e.g. as a replacement for REXEC).

z/VSE Database Connector (DBCLI)

The Database Call Level Interface (DBCLI) allows z/VSE applications to access a relational database on any suitable database server. Therefore, you have the flexibility of being able to choose a database server (IBM DB2, Oracle, Microsoft SQL Server, MySQL, and so on) that runs on a platform other than z/VSE.

Because neither the DBCLI client nor the DBCLI server inspect the SQL statements being used by your application program, you can use any type of SQL statements and SQL dialect that are supported by the Vendor's JDBC Driver providing the database-provider also supplies a suitable JDBC driver.

The z/VSE DBCLI consist of 2 main parts:

- The DBCLI client that runs under z/VSE.
- The DBCLI server that runs on a Java platform.

The DBCLI client provides a programming API for your application programs:

1. The DBCLI client connects to the DBCLI server via a TCP/IP connection.
2. The DBCLI client translates the calls from your application programs into requests to the DBCLI server.
3. The DBCLI server receives the requests from the applications and passes them to a JDBC driver that is provided by the Vendor database.
4. The DBCLI server returns the result of the call to the application program running under z/VSE via the DBCLI client.

Web Services (SOAP) with z/VSE

Web services is a technology that allows applications to communicate with each other in a platform- and programming language-independent manner. A Web service is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging. It uses protocols based on the XML language to describe an operation to be executed or data to be exchanged with another Web service.

SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and HTTP Extension Framework.

The SOAP implementation is part of the z/VSE e-business Connectors and runs within CICS TS. A simple SOAP example for z/VSE is part of the z/VSE Connector Client.

The implementation allows a z/VSE system to act as a Web Service provider (server) and as a Web Service requestor (client).

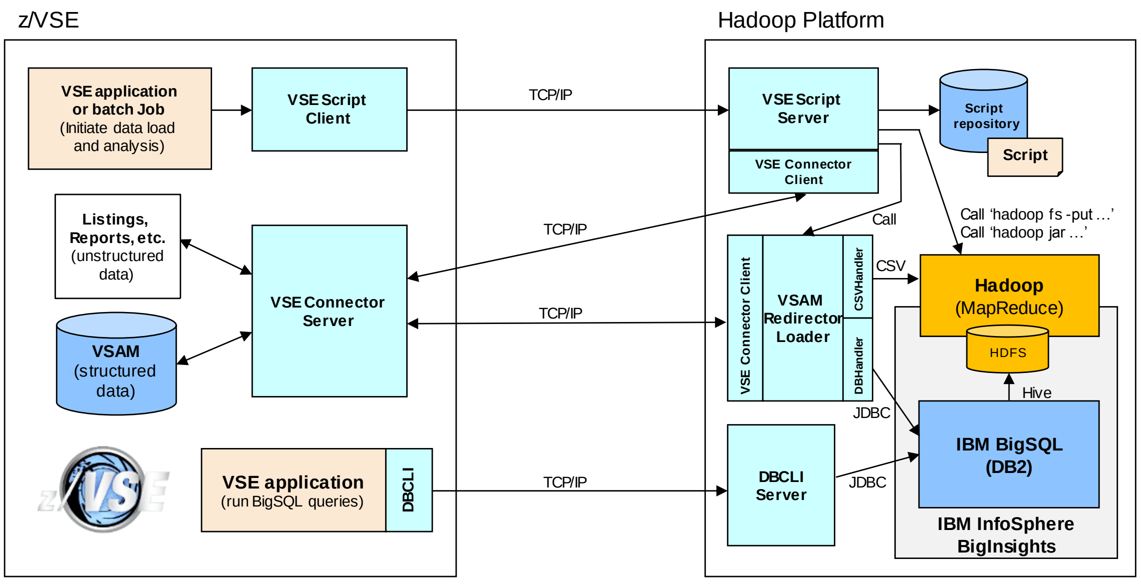
- **z/VSE as a service provider:** a CICS TS application can be provided as a Web Service that is callable from outside of z/VSE using the SOAP protocol (the program must be callable with a COMMAREA).
- **z/VSE as a service requestor:** any CICS TS application can call a Web Service that resides outside of z/VSE using a EXEC CICS LINK.

2. Analyze z/VSE data with Hadoop and Big Insights

There are various possibilities how to put z/VSE data into the Hadoop Distributed File System (HDFS) and analyze it using Hadoop's MapReduce programming paradigm.

Note: This chapter shows how to put z/VSE data into HDFS, and how to run a MapReduce program to analyze the data. However, it does **NOT** discuss how to develop a MapReduce program itself to solve a certain analysis task. This paper assumes that the appropriate MapReduce program is already available, or that the knowledge how to implement a MapReduce program exists.

The following picture shows the connector components used. The different possibilities will be discussed in detail in the following sections.



2.1. *Analyze z/VSE reports, logs or listings with Hadoop*

Most z/VSE customers want to control the data transfer as well as the execution of the analysis program from the z/VSE side. Either from within a z/VSE job (JCL) or from within an application program running on z/VSE.

The z/VSE Script connector provides functions to:

- Transfer (download) z/VSE data
- Run commands and programs on the Hadoop node

For unstructured data, such as reports, logs or job listings, the z/VSE Script Server provides built in script functions to:

- Run jobs on z/VSE that create the desired reports that are to be analyzed.
- Download the reports, logs or job listings to the Script Server system.
- Run the '`hadoop fs -put ...`' command on the Hadoop node to put the downloaded data into the Hadoop file system (HDFS)
- Run the '`hadoop jar ...`' command on the Hadoop node to run the MapReduce analysis program.

- Transfer the results of the analysis back to z/VSE.

Note: z/VSE reports, logs or job listings are typically not so large, so that they can be first transferred into a temporary file on the Script Server (local file system), and then put into the Hadoop file system (HDFS).

The following example script shows how to run a PRINTLOG job on z/VSE to print recent z/VSE console messages, and then put its output into HDFS for analysis. The example script then executes the example MapReduce program “wordcount”, which is provided by Hadoop. Wordcount simply counts the number of occurrences of each word in the input text file(s). Of course, counting the words on a PRINTLOG listing does not make much sense. In real live, you would implement a MapReduce program that analyses the recent console messages and looks for unusual message patters or sequences that humans would not likely see. It might provide indications of how conditions vary from normal operations. By identification of such unusual messages surrounding an incident, you can quickly respond and implement changes in z/VSE needed to prevent problem recurrence. This is kind of what IBM zAware is doing.

```
//****************************************************************************
//*           z/VSE VSE Script Server                                        *
//****************************************************************************
//*                                                                          *
//* MODULE NAME : hadoopPrintlog.src                                         *
//*                                                                          *
//* COPYRIGHT NOTICE: Copyright (C) 2015.  IBM Corporation.                  *
//*                                                                          *
//*   You have a royalty-free right to use, modify, reproduce and            *
//*   distribute this demonstration file (including any modified             *
//*   version), provided that you agree that IBM Corporation                 *
//*   has no warranty, implied or otherwise, or liability                    *
//*   for this demonstration file or any modified version.                   *
//*                                                                          *
//****************************************************************************
//* This example demonstrates how to copy VSE data into the                  *
//* Hadoop Distributed File System (HDFS) and to run a Hadoop                *
//* program on it.                                                           *
//****************************************************************************

//******************************************************************
// General settings
//******************************************************************
// the VSE system to use
string system              = "VSE01";
string printlog_job;
string printlog_disp       = "NEW";
string printlog_out        = "PRINTLOG.out";
string hadoop_cmd          = "hadoop";
string hdfs_input_path     = "/user/biadmin/vse";
string hdfs_output_path    = "/user/biadmin/vse-out";
string hadoop_wordcount_jar = "/opt/ibm/biginsights/IHC/hadoop-example.jar";
string hadoop_wordcount_cmd = "wordcount";

//******************************************************************
// Preparation
//******************************************************************
int numargs;
arraySize(&argv,&numargs);
if (numargs > 0) do;
  printlog_disp = argv[0];
endif;

printlog_job[0] = "* $$ JOB JNM=PRINTLOG,CLASS=0,DISP=D";
printlog_job[1] = "* $$ LST DISP=D,CLASS=A";
printlog_job[2] = "// JOB PRINTLOG";
printlog_job[3] = "// EXEC PRINTLOG,PARM='"+printlog_disp+"'";
printlog_job[4] = "/*";
printlog_job[5] = "/&";
printlog_job[6] = "* $$ EOJ";
```

```
string jout;
int rc;
string errmsg;
int numlines;
int i;
string args;
string retlines;

//*****************************************************************
// Step 1: Run a PRINTLOG job on VSE to get the console messages
//*****************************************************************
println("Step 1: Executing a PRINTLOG ("+printlog_disp+") job on VSE...");
// execute the job in jin and store the joboutput in jout
executePowerJob(system,&printlog_job,&jout,&rc);
getLastErrorMsg(&errmsg);
if(rc!=0) do;
  println("Failed to execute PRINTLOG job. RC: "+rc+" Error: "+errmsg);
  exit(2);
endif;

// save the output into a local file
saveFile(printlog_out,&jout,&rc);
getLastErrorMsg(&errmsg);
if(rc!=0) do;
  println("Failed to write file "+printlog_out+" RC: "+rc+" Error: "+errmsg);
  exit(3);
endif;

//*****************************************************************
// Step 2: Create input directory in HDFS and clear output dirctory
//*****************************************************************
println("Step 2: Create input/output directories in HDFS...");

resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
args[1] = "-mkdir";
args[2] = hdfs_input_path;
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;

resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
args[1] = "-rm";
args[2] = "-r";
args[3] = "-skipTrash";
args[4] = hdfs_input_path+"/*";
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;

resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
args[1] = "-rm";
args[2] = "-r";
args[3] = "-skipTrash";
args[4] = hdfs_output_path;
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;

//*****************************************************************
// Step 3: Put the local file into the HDFS file system
//*****************************************************************
println("Step 3: Putting the PRINTLOG output into HDFS...");
resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
```

```
args[1] = "-put";
args[2] = "-f";
args[3] = printlog_out;
args[4] = hdfs_input_path;
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;
if(rc!=0) do;
  println("Failed to put "+printlog_out+" into HDFS. RC: "+rc+" Error: "+errmsg);
  exit(4);
endif;

//*************************************************************
// Step 4: Run the WordCount MapReduce example on the data
//*************************************************************
println("Step 4: Running the WordCount MapReduce example...");
resetVar(&args);
resetVar(&retlines);
args[0] = "jar";
args[1] = hadoop_wordcount_jar;
args[2] = hadoop_wordcount_cmd;
args[3] = hdfs_input_path;
args[4] = hdfs_output_path;
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;
if(rc!=0) do;
  println("Failed to execute "+hadoop_wordcount_cmd+" with hadoop. RC: "+rc+"
Error: "+errmsg);
  exit(5);
endif;

//*************************************************************
// Step 5: Pass back the output
//*************************************************************
println("Step 5: Pass back the output...");
resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
args[1] = "-ls";
args[2] = hdfs_output_path+"/part*";
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;
if(rc!=0) do;
  println("Failed to list "+hdfs_output_path+"/part*. RC: "+rc+" Error: "+errmsg);
  exit(6);
endif;

resetVar(&args);
resetVar(&retlines);
args[0] = "fs";
args[1] = "-cat";
args[2] = hdfs_output_path+"/part*";
call(hadoop_cmd, &args, &retlines, &rc);
getLastErrorMsg(&errmsg);
println("****** Output: *****");
gosub print_retlines;
println("*******************");
if(rc!=0) do;
  println("Failed to display "+hdfs_output_path+"/part*. RC: "+rc+" Error:
"+errmsg);
  exit(7);
endif;

println("Finished.");


//*************************************************************
// Soubroutine to print the retlines array
//*************************************************************
```

```
sub print_retlines;
  arraySize(&retlines,&numlines);
  if (numlines > 0) do;
    for(&i,0,numlines-1) do;
      println("  "+retlines[i]);
    endfor;
  endif;
return;
```

2.2. Analyze VSAM data with Hadoop

The previous section describes how to transfer unstructured data, such as reports, logs, or job listings into the Haddop file system (HDFS). For structured data, such as VSAM, another approach should be used. Of course, VSAM data can simply be transferred ‘as is’ using FTP and then put into HDFS. In case the VSAM data contains pure textual data (e.g. an ESDS cluster used as kind of a log file) a simple text-mode FTP would be sufficient. However, for more sophisticated VSAM data (e.g. variable length KSDS cluster), a simple binary-mode FTP transfer would typically remove record boundary information, and thus makes it hard to read and analyze the data afterwards in the MapReduce jobs.

In order to download a huge amount of VSAM data in a format that MapReduce programs can use easily, the VSAM Redirector Loader in combination with the CrlfFileHandler or CSVFileHandler can be used (part of the VSAM Redirector Server package). The VSAM Redirector Loader is designed to download lots of VSAM data in a performant way, and pass the downloaded VSAM records to a configurable Handler, which translates the data into a target format.

- **CrlfFileHandler:** The CRLF file handler translates each record from EBCDIC into ASCII and stores it into a plain text file. This handler can be used for VSAM clusters that contain plain text data.
- **CSVFileHandler:** The CSV file handler is much more sophisticated, and supports the translation of the VSAM record into fields of different data types (e.g. STRING, PACKED, ZONED, BINARY, etc.). The VSAM records are converted based on the supplied mapping rules, and are stored in a Comma Separated Values (CSV) file. Thus, the records and field structure is retained.

Hadoop provides readers that support to read plain text files as well as CSV files. CSV files can also be imported into Hadoop’s Hive to create relational database tables out of it. This makes it easy to analyze VSAM data downloaded as CSV file.

You can use the z/VSE Script Server to automate the execution of the VSAM Redirector Loader and the subsequent execution of the Hadoop commands to put the created CSV file into HDFS as well as the execution of the MapReduce job to analyze the data.

The following z/VSE script snippet shows how to run the VSAM Redirector Loader from within a z/VSE script. It assumes that the record mapping has been provided in file sample-mapping.csv. The mapping can be created using the MapperConfigGUI tool coming with the VSAM Redirector Server package.

```
//******************************************************************
// General settings
//******************************************************************
// the VSE system to use
string system              = "9.152.131.164";
string user                = "SYSA";
string password            = "u89NXLKyKiJ/In9+JWC0pQ==";
string vsam_catalog        = "VSESP.USER.CATALOG";
string vsam_cluster        = "VSAM.CONN.SAMPLE.DATA";
string redir_handler       = "com.ibm.vse.csvfilehandler.CSVFileHandler";
string redir_options       = "\"configfile=sample-mapping.csv\"";
string redir_runloader     = "/opt/IBM/VSAMRedirectorServer/runloader.sh";
string cfg_file_name        = "REDIRLOADER.cfg";
string cfg_file;

cfg_file[0] = "VSEHOST="+system;
cfg_file[1] = "VSEUSER="+user;
cfg_file[2] = "encVSEPASSWORD="+password;
cfg_file[3] = "VSAMCATALOG="+vsam_catalog;
cfg_file[4] = "VSAMCLUSTER="+vsam_cluster;
cfg_file[5] = "HANDLER="+redir_handler;
cfg_file[6] = "OPTIONS="+redir_options;
cfg_file[7] = "# Optional parameters";
cfg_file[8] = "REUSE=YES";
cfg_file[9] = "TRACE=NO";
cfg_file[10] = "SHOWPROGRESS=NO";
cfg_file[11] = "MAX-PREFETCH-COUNT=100";
cfg_file[12] = "STOP-ON-DUPLICATE-RECORD-ERROR=YES";
cfg_file[13] = "STOP-ON-OTHER-ERROR=YES";
cfg_file[14] = "CLOSE=YES";
cfg_file[15] = "CONVERTERDIRECTORY=converters";

int rc;
string errmsg;
int numlines;
int i;
string args;
string retlines;

//******************************************************************
// Step 1: Create a config file for the Redirector Loader
//******************************************************************
println("Step 1: Create a config file for the Redirector Loader...");
saveFile(cfg_file_name,&cfg_file,&rc);
getLastErrorMsg(&errmsg);
if(rc!=0) do;
  println("Failed create the Redirector Loader config file. RC: "+rc+" Error: 
"+errmsg);
  exit(2);
endif;

//******************************************************************
// Step 2: Run the Redirector Loader
//******************************************************************
println("Step 2: Running the Redirector Loader...");
// Run the loader and wait for completition
args[0]=cfg_file_name;
call(redir_runloader,&args,&retlines,&rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;
if(rc!=0) do;
  println("Failed to run the Redirector Loader. RC: "+rc+" Error: "+errmsg);
  exit(4);
endif;

//******************************************************************
// Step 3: Put the local file into the HDFS file system
//******************************************************************
...

//******************************************************************
// Step 4: Run the MapReduce job on the data
//******************************************************************
...

//******************************************************************
// Step 5: Pass back the output
//******************************************************************
...

//******************************************************************
// Soubroutine to print the retlines array
//******************************************************************
```

```
sub print_retlines;
  arraySize(&retlines,&numlines);
  if (numlines > 0) do;
    for(&i,0,numlines-1) do;
      println("  "+retlines[i]);
    endfor;
  endif;
return;
```

2.3. Stream VSAM data into Big SQL for analysis

The previous section showed how to download VSAM data as a CSV file, and then put it into Hadoop's file system (HDFS) for analysis. If the downloaded VSAM data is rather large, then the intermediate step of downloading the data as CSV file and then putting the possibly large CSV file into HDFS may be quite time and disk space consuming.

Instead, the VSAM data can also be **streamed** directly into IBM Big SQL, without the need of storing the data in an intermediate file. By loading the data into Big SQL you can easily analyze the data using SQL queries. Big SQL will automatically break complex queries into parts and parallelizes them using Hadoop's MapReduce framework.

To load VSAM data into Big SQL, the VSAM Redirector Loader in combination with the DBHandler can be used. The DBHandler is provided as part of the VSAM Redirector Server.

Similar to the previous section, you can use the z/VSE Script Server to automate the execution of the VSAM Redirector Loader and the subsequent execution of the analysis program.

Note: The old Big SQL Version 1 is not supported, since it uses a special JDBC driver that is not capable of working with the DBHandler. The current Big SQL version uses the DB2 Universal Driver, which is supported by the DBHandler.

The following z/VSE script snippet shows how to run the VSAM Redirector Loader from within a z/VSE script. It assumes that the record mapping has been provided in DB2 table LOADER_CONFIG. The mapping can be created using the MapperConfigGUI tool coming with the VSAM Redirector Server package. The data table (USEDCARS) must be created in Big SQL by modifying the CREATE TABLE statement generated by the MapperConfigGUI to "CREATE **HADOOP** TABLE ...". That way the data inserted into the data table is stored in HDFS (via HBase or Hive), rather than into a regular DB2 table. You may have to adjust the data types of some columns, since HADOOP tables do not support all data types. For example, change data type DECIMAL to BIGINT. The mapping table (LOADER_CONFIG) does not need to be a HADOOP table, it can be a regular DB2 table, since it does not hold any data to be analyzed later on.

```
//***************************************************************************
//*           z/VSE VSE Script Server                                      *
//***************************************************************************
//*                                                                         *
//* MODULE NAME : hadoopBigSQL.src                                          *
//*                                                                         *
//* COPYRIGHT NOTICE: Copyright (C) 2015.  IBM Corporation.                 *
//*                                                                         *
//*   You have a royalty-free right to use, modify, reproduce and           *
//*   distribute this demonstration file (including any modified            *
//*   version), provided that you agree that IBM Corporation                *
//*   has no warranty, implied or otherwise, or liability                   *
//*   for this demonstration file or any modified version.                  *
//*                                                                         *
//***************************************************************************
```

```
//* This example demonstrates how to copy VSAM data into the                *
//* Hadoop Distributed File System (HDFS) using IBM's Big SQL.               *
//*                                                                         *
//* Preparation required:                                                   *
//* You need to create the tables and the mapping in the database before    *
//* running this example. You do this with the MapperConfigGUI tool that comes *
//* with the VSAM Redirector Package. Import the usedcars.xml file into the *
//* MapperConfigGUI and create a mapping. Then create the config table, as  *
//* well as the data table(s) and save the mapping configuration into the   *
//* config table.                                                           *
//***************************************************************************

//************************************************************
// General settings
//************************************************************
// the VSE system to use
string system               = "9.152.131.164";
string user                 = "SYSA";
string password             = "u89NXLKyKiJ/In9+JWC0pQ==";
string vsam_catalog         = "VSESP.USER.CATALOG";
string vsam_cluster         = "VSAM.CONN.SAMPLE.DATA";
string redir_handler        = "com.ibm.vse.dbhandler.DBHandler";
string redir_options        =
"\"dburl=jdbc:db2://bivm.ibm.com:51000/bigsql;dbuser=biadmin;dbpassword=biadmin;co
nfigtable=LOADER_CONFIG;configname=USEDCARS;allowreuse=yes\"";
string redir_runloader      = "/opt/IBM/VSAMRedirectorServer/runloader.sh";
string cfg_file_name        = "REDIRLOADER.cfg";
string cfg_file;

cfg_file[0] = "VSEHOST="+system;
cfg_file[1] = "VSEUSER="+user;
cfg_file[2] = "encVSEPASSWORD="+password;
cfg_file[3] = "VSAMCATALOG="+vsam_catalog;
cfg_file[4] = "VSAMCLUSTER="+vsam_cluster;
cfg_file[5] = "HANDLER="+redir_handler;
cfg_file[6] = "OPTIONS="+redir_options;
cfg_file[7] = "# Optional parameters";
cfg_file[8] = "REUSE=YES";
cfg_file[9] = "TRACE=NO";
cfg_file[10] = "SHOWPROGRESS=NO";
cfg_file[11] = "MAX-PREFETCH-COUNT=100";
cfg_file[12] = "STOP-ON-DUPLICATE-RECORD-ERROR=YES";
cfg_file[13] = "STOP-ON-OTHER-ERROR=YES";
cfg_file[14] = "CLOSE=YES";
cfg_file[15] = "CONVERTERDIRECTORY=converters";

int rc;
string errmsg;
int numlines;
int i;
string args;
string retlines;
string cfg_file_name;

//************************************************************
// Step 1: Create a config file for the Redirector Loader
//************************************************************
println("Step 1: Create a config file for the Redirector Loader...");
saveFile(cfg_file_name,&cfg_file,&rc);
getLastErrorMsg(&errmsg);
if(rc!=0) do;
  println("Failed create the Redirector Loader config file. RC: "+rc+" Error:
"+errmsg);
  exit(2);
endif;

//************************************************************
// Step 2: Run the Redirector Loader
//************************************************************
println("Step 2: Running the Redirector Loader...");
// Run the loader and wait for completition
```

```
args[0]=cfg_file_name;
call(redir_runloader,&args,&retlines,&rc);
getLastErrorMsg(&errmsg);
gosub print_retlines;
if(rc!=0) do;
  println("Failed to run the Redirector Loader. RC: "+rc+" Error: "+errmsg);
  exit(4);
endif;

//*************************************************************
// Step 3: Run an SQL query using Big SQL against the data
//*************************************************************
// You can now run SQL queries against the just loaded data.
// BigSQL will try to break the query into smaller parts that are
// then executed parallel on the Hadoop cluster nodes.
...


//*************************************************************
// Soubroutine to print the retlines array
//*************************************************************
sub print_retlines;
  arraySize(&retlines,&numlines);
  if (numlines > 0) do;
    for(&i,0,numlines-1) do;
      println("  "+retlines[i]);
    endfor;
  endif;
return;
```

2.4. *Running Big SQL queries through the z/VSE Database Connector*

Once VSAM data has been loaded into Big SQL tables, you can use various ways to execute SQL queries against the data.

By using the z/VSE Database Connector you can implement z/VSE applications (batch or CICS) that use the DBCLI programming interface to connect to a Big SQL database instance, and run (complex) queries against the data previously loaded into the Big SQL tables. Big SQL will execute the query by breaking complex queries into parts and parallelizes them using Hadoop's MapReduce framework. The z/VSE application will then retrieve the query result by fetching the result rows through DBCLI.

You configure the DBCliServer to access the Big SQL database just like any other DB2 database.

Note 1: The old Big SQL Version 1 is not supported, since it uses a special JDBC driver that is not capable of working with the z/VSE Database Connector. The current Big SQL version uses the DB2 universal driver, which is supported by the z/VSE Database Connector.

Note 2: Not all type of SQL queries can be issued through the z/VSE Database Connector, because the Big SQL JDBC driver does not allow to prepare certain SQL statements. However, SELECT statements are working, which is what you need to run queries.

2.5. Using the Java-based Connector for special purpose scenarios

For special cases where neither the z/VSE Script Server, not the VSAM Redirector Loader provides sufficient support, you can also develop own Java programs that loads/streams z/VSE data in an appropriate way into Hadoop's file system (HDFS). This task requires general Java knowledge as well as knowledge about Hadoop's Java programming interfaces.

By utilizing the z/VSE Connector Client Java classes together with the Hadoop programming interface in the same Java program, you can create special purpose Java programs that load data from z/VSE and store or stream it into HDFS as needed. The z/VSE Connector Client & Server provide methods to access various kinds of z/VSE data, such as VSAM, DL/1, LIBR, POWER, Console, etc.

Hadoop provides various programming interfaces that can be used to create custom solutions:

Hadoop File System (HDFS) API

Hadoop provides an easy to use Java programming interface to create files inside HDFS and write or read files inside HDFS. Class org.apache.hadoop.fs.FileSystem provides the base for HDFS file system access. It provides methods like create, append, close, delete, etc. to access files stored in HDFS. HDFS allows to either sequentially write to or sequentially read from a file stored in HDFS.

Your Java program could retrieve data from z/VSE by using the z/VSE Connector Client classes, and stream the data right into HDFS by using Hadoop's file system API. Any required data translation or conversion can be performed by the Java program, so that the MapReduce jobs can then read the data easily and analyze it.

The following Java code snippet shows how to use the VSE Connector Client classes to download (i.e. stream) a z/VSE library member directly into a file in HDFS:

```
// stream a VSE library member into HDFS
VSEConnectionSpec spec = new VSEConnectionSpec(InetAddress.getByName("9.152.131.164"),
                                              2893,"SYSA","password");
VSESystem system = new VSESystem(spec);
VSELibrarian libr = system.getVSELibrarian();
VSELibraryMember member = libr.getVSELibraryMember("PRD2", "CONFIG", "IPINIT00", "L");
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
Path p = new Path("hdfs://bivm.ibm.com:9000/user/biadmin/ipinit00.l");
DataOutputStream out = fs.create(p, true);
member.setBinary(false);
member.download(out); //  download it directly into the HDFS file
out.close();
```

Above code can be added into a Java program that you use to run the MapReduce job (i.e. the program that you start with the '`hadoop jar …`' command), or it can be used in a standalone program running independent of the MapReduce job.

For special purposes you can also implement your own custom Handler for the VSAM Redirector Loader. For example, take the CrlfFileHandler.java as an example (provided in source in the VSAM Redirector Package), and adapt it so that it streams the data right into HDFS.

Custom Data Types

Hadoop MapReduce uses typed data at all times when it interacts with user-provided Mappers and Reducers: data read from files into Mappers, emitted by mappers to reducers, and emitted by reducers into output files is all stored in Java objects.

Objects which can be marshaled to or from files and across the network must obey a particular interface, called Writable, which allows Hadoop to read and write the data in a serialized form for transmission. Hadoop provides several stock classes which implement Writable: Text (which stores String data), IntWritable, LongWritable, FloatWritable, BooleanWritable, and several others. The entire list is in the org.apache.hadoop.io package.

In addition to these types, you are free to define your own classes which implement Writable. You can organize a structure of virtually any layout to fit your data and be transmitted by Hadoop.

You could, for example, create your own type that represents a particular VSAM record. It would know what fields such a VSAM record type contains and would know how to read (de-serialize) and write (serialize) such a record from/to the underlying file.

Custom Input Formats

The InputFormat defines how to read data from a file into the Mapper instances. Hadoop comes with several implementations of InputFormat; some work with text files and describe different ways in which the text files can be interpreted. Others, like SequenceFileInputFormat, are purpose-built for reading particular binary file formats.

More powerfully, you can define your own InputFormat implementations to format the input to your programs however you want. For example, the default TextInputFormat reads lines of text files. The key it emits for each record is the byte offset of the line read (as a LongWritable), and the value is the contents of the line up to the terminating '\n' character (as a Text object). If you have multi-line records each separated by a $ character, you could write your own InputFormat that parses files into records split on this character instead.

Another important job of the InputFormat is to divide the input data sources (e.g., input files) into fragments that make up the inputs to individual map tasks. These fragments are called "splits" and are encapsulated in instances of the InputSplit interface. Most files, for example, are split up on the boundaries of the underlying blocks in HDFS, and are represented by instances of the FileInputSplit class. Other files may be unsplittable, depending on application-specific data. Dividing up other data sources (e.g., tables from a database) into splits would be performed in a different, application-specific fashion.

A z/VSE specific InputFormat implementation could access the z/VSE system and read it directly from the VSAM cluster. However, this would possibly interfere with Hadoop's parallel execution on many Hadoop nodes, and would result in slow performance. Instead, you would possibly create an InputFormat implementation that is able to construct the fields from a binary file stored in HDFS that, for example, was downloaded in binary using FTP (with MODE B).

Custom Output Formats

An OutputFormat dictate how to write the results of a job back to the underlying permanent storage. Several useful OutputFormat implementations are available in Hadoop. The default format (TextOutputFormat) will write (key, value) pairs as strings to individual lines of an output file. The SequenceFileOutputFormat will keep the data in binary, so it can be later read quickly by the SequenceFileInputFormat.

You can define your own OutputFormat implementation that will write data to an underlying medium in the format that you control. You could, for example, create your own OutputFormat implementation that writes the data in a binary form so that it then can be easily transferred back to z/VSE into a VSAM cluster. Again, accessing z/VSE directly from within an OutputFormat implementation is possible, but may possibly interfere with Hadoop's parallel execution on many Hadoop nodes, and would result in slow performance.

Custom Hive Serializer/Deserializer (SerDe)

SerDe is short for Serializer/Deserializer. Hive uses the SerDe interface for IO. The interface handles both serialization and deserialization and also interpreting the results of serialization as individual fields for processing.

A SerDe allows Hive to read in data from a table, and write it back out to HDFS in any custom format. Anyone can write their own SerDe for their own data formats.

You can implement a z/VSE specific SerDe implementation, that for example understands the record layout of a VSAM record. The SerDe implementation may also access z/VSE data directly by using the z/VSE Connector Client classes.

Appendix A: References

z/VSE V5R1 e-business Connectors User's Guide, SC34-2629-02
http://publibz.boulder.ibm.com/cgi-bin/bookmgr/download/IESCUE72.pdf

z/VSE download web page:
http://www.ibm.com/systems/z/os/zvse/downloads/

IBM InfoSphere BigInsights
http://www.ibm.com/software/data/infosphere/hadoop/enterprise.html
http://www.ibm.com/software/data/infosphere/hadoop/mapreduce/
http://www.ibm.com/software/data/infosphere/hadoop/big-sql.html

IBM InfoSphere BigInsights Quick Start Edition
http://www.ibm.com/software/data/infosphere/biginsights/quick-start/

Apache Hadoop web page
http://hadoop.apache.org/