Mobius User's Manual

v0.1 06/25/2012

Contents

1.0 GENERAL INFORMATION

1.1 What's Mobius?

Mobius is an integrated data analytic platform that helps simplify and streamline the large scale data analysis life cycle.

Mobius has two major components:

- The Mobius core API is a Java-based, high-level data processing framework built on top of the Apache Hadoop project.
- The Mobius Eclipse plugin supports local debugging, managing multiple analysis jobs through a high-level GUI-based configuration console and removes the burden of resource packaging, job submission, and management when using Hadoop.

1.2 Who Should Use Mobius?

Mobius is for anyone who wants to accelerate the analysis process, improve job interaction, and decrease the amount of system management. Mobius saves time by automatically breaking down high level analysis flow operations into several MapReduce steps, so users don't need to worry about dependencies between jobs, debugging the environment setup, monitoring resources, or managing jobs. Users can stay focused on implementing the analysis logic, and let the system take care of the rest.

Mobius is built on top of Hadoop, a platform that many companies use to process vast amounts of data. People use Hadoop for to perform complex analysis flow, which requires a series of MapReduce iterations, writing a number of mapper and reducer classes, and then requires accurate planning the job dependencies, which is usually a non-trivial and time consuming process.

A complex analysis can be shown in a DAG graph, with each node represented as a Mapper or a Reducer task that solves part of the problem, and edges denoting the dependencies. Together, these complete the analysis logic. Nodes without parents can be started in parallel, while nodes with parents must wait until their parents are finished successfully before they can be started. Analysts/developers not only need to spend their time implementing the business logic, but also must break down the problem into X number of MapReduce jobs and plan the job dependencies properly so that their job runs more efficiently.

After the prototype flow is complete, developers must spend some time debugging it locally. They must set up the environment correctly, download the data from the cluster, and possibly edit some code if the IO path is different for the local and server nodes.

Mobius is designed to manage this complexity and provide a quick ramp up time in a friendly environment for users to perform large-scale data processing easily.

2.0 QUICK START

This section contains a set of Mobius examples of the most common scenarios. Not all functionalities are shown in this section.

2.1 Listing Data

```
package example;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.model.Column;

@SuppressWarnings({ "deprecation", "unused" })
public class ListJobTest extends MobiusJob
{
    private static final long serialVersionUID = -8730331936213740568L;

    @Override
    public int run(String[] args) throws Exception {

        // building a dataset on HDFS
        Dataset tsv = TSVDatasetBuilder.newInstance(
                this,
                "TSV", // name of this dataset, it will be shown on jobtracker
                new String[]{"ID", "TIME", "URL", "PARAMETER"} // dataset schema
            )
            .addInputPath(new Path("$PATH_TO_MY_INPUT"))
        .build();

        // store ID and TIME to the output, the first column is ID and
        // 2nd column is TIME, one can change the ordering to make ID
        // on the 2nd column and TIME on the 1st column:
        //
        // new Column(tsv, “TIME”), new Column(tsv, “ID”)
        this.list(tsv,
                new Path("$PATH_TO_MY_OUTPUT"),
                new Column(tsv, "ID"),
                new Column(tsv, "TIME")

        );

        return 0;
    }

    public static void main(String[] args)
        throws Throwable
    {
        int exit = MobiusJobRunner.run(new ListJobTest(), args);
        System.exit(exit);
    }
}
```

2.2 Joining Two Datasets

```
package example;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.TextOutputFormat;

import com.ebay.erl.mobius.core.JoinOnConfigure.EQ;
import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.function.Avg;
import com.ebay.erl.mobius.core.function.Max;
import com.ebay.erl.mobius.core.function.Medium;
import com.ebay.erl.mobius.core.function.Min;
import com.ebay.erl.mobius.core.function.Unique;
import com.ebay.erl.mobius.core.model.Column;

@SuppressWarnings("deprecation")
public class InnerJoinTest extends MobiusJob
{
	private static final long serialVersionUID = -5045003345238678287L;

	@Override
	public int run(String[] args)
		throws Exception
	{
		String orderInputPath  = "$ORDER_PATH";// HDFS path to the order dataset
		String personInputPath = "$PERSON_PATH";// HDFS path to the persone dataset

		// building order dataset
		Dataset order = TSVDatasetBuilder.newInstance(
				this,
				"orders",
				new String[]{"O_Id", "OrderNo", "P_Id"}
			)
			.addInputPath(new Path(orderInputPath))
		.build();

		// building person dataset
		Dataset person = TSVDatasetBuilder.newInstance(
				this,
				"persons",
				new String[]{"P_Id", "LastName", "FirstName", "Address", "City"}
			)
			.addInputPath(new Path(personInputPath))
		.build();

		this
			.innerJoin(person, order)
			.on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
			.save(this,
					new Path("$PATH_TO_OUTPUT")
					,new Column(person, "P_Id")
					,new Column(person, "FirstName")
					,new Column(person, "LastName")
					,new Column(order, "OrderNo")
			);

		return 0;
	}

	public static void main(String[] args)
		throws Throwable
	{
		int exit = MobiusJobRunner.run(new InnerJoinTest(), args);
		System.exit(exit);
	}

}
```

2.3 Grouping a Dataset

```
package test;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.function.Max;
import com.ebay.erl.mobius.core.model.Column;

public class GroupByTest extends MobiusJob
{

	private static final long serialVersionUID = 5217672582768153239L;

	@Override
	public int run(String[] args) throws Exception
	{
		String orderInputPath  = "$ORDER_PATH";// HDFS path to the order dataset

		Dataset order = TSVDatasetBuilder.newInstance(
				this,
				"orders",
				new String[]{"O_Id", "OrderNo", "P_Id"}
			)
			.addInputPath(new Path(orderInputPath))
		.build();

		this
			.group(order)
			.by(new Column(order, "P_ID"))
			.save(this,
					new Path("$PATH_TO_MY_OUTPUT")
					, new Column(order, "P_ID")
					, new Max(new Column(order, "O_Id"))
			);
		return 0;
	}

	public static void main(String[] args)
		throws Throwable
	{
		int exit = MobiusJobRunner.run(new GroupByTest(), args);
		System.exit(exit);
	}

}
```

2.4 Sorting a Dataset

```
package example;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.model.Column;
import com.ebay.erl.mobius.core.sort.Sorter;
import com.ebay.erl.mobius.core.sort.Sorter.Ordering;

public class SortTest extends MobiusJob
{
	private static final long serialVersionUID = 7945209428566552518L;

	@Override
	public int run(String[] args)
		throws Exception
	{

		Dataset ds = TSVDatasetBuilder.newInstance(
				this,
				"people",
				new String[]{"fname","lname","age","zipcode","gender"}
			)
			.setDelimiter(",")
			.addInputPath(new Path("$INPUT"))
		.build();

		this
			.sort(ds)
			.select(
					new Column(ds, "age"),
					new Column(ds, "gender"),
					new Column(ds, "fname"),
					new Column(ds, "lname"))
			.orderBy(
					new Sorter(new Column(ds, "age"), Ordering.ASC, true),
					new Sorter(new Column(ds, "gender"), Ordering.DESC, true))
			.save(
					this,
					new Path("$OUTPUT")
			);

		return 0;
	}

	public static void main(String[] args)
		throws Throwable
	{
		int exit = MobiusJobRunner.run(new SortTest(), args);
		System.exit(exit);
	}
}
```

2.5 Grouping then Sorting a Dataset

```
package com.ebay.erl.mobius.core;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.function.Counts;
import com.ebay.erl.mobius.core.model.Column;
import com.ebay.erl.mobius.core.sort.Sorter;
import com.ebay.erl.mobius.core.sort.Sorter.Ordering;

public class GroupThenSort extends MobiusJob {

	private static final long serialVersionUID = -7216853302346293360L;

	@Override
	public int run(String[] args) throws Exception {

		Dataset people = TSVDatasetBuilder.newInstance(
					this,
					"people",
					new String[]{"fname","lname","age","zipcode","gender"}
				)
				.setDelimiter(",")
				.addInputPath(new Path("$INPUT"))
			.build();

			// group by the people dataset, and store it
			// in a temporal place
			Dataset d1 = this
				.group(people)
				.by(new Column(people, "age"))
				.build(this
					, new Column(people, "age")
					, new Counts(new Column(people,
"zipcode")).setOutputSchema("ZIP_CODE_COUNT")
				);


			// sort the intermediate result (d1) by
			// age ascending then by zip code counts
			// descending.
			this
				.sort(d1)
				.select(
					new Column(d1, "age"),
					new Column(d1, "ZIP_CODE_COUNT"))
				.orderBy(
					new Sorter(new Column(d1, "age"), Ordering.ASC, true),
					new Sorter(new Column(d1, "ZIP_CODE_COUNT"), Ordering.DESC,
true))
				.save(
					this,
					new Path("$OUTPUT")
				);

			return 0;
	}

	public static void main(String[] args) throws Throwable {
		int exit = MobiusJobRunner.run(new GroupThenSort(), args);
		System.exit(exit);
	}
}
```

3.0 MOBIUS CORE API

3.1 Dataset

The *Dataset* class is the representation of a data source that is readable via an implementation of *org.apache.hadoop.mapred.InputFormat*. It can be a set of files on HDFS or rows from a database. A dataset always has a fixed schema associated with it, and each record is represented as a *tuple*. When the data source can be represented as *Dataset* using *DatasetBuilder* to build it, then Mobius can process it through a set of high-level operators, such as joining, grouping, or sorting.

Note that, even though the schema of a dataset is fixed, the type of the value of a column can be a hash map, allowing Mobius to process semi-structured data. For more information about the supported types, check *Tuple* for more detail.

3.2 DatasetBuilder

The *DatasetBuilder* class builds a dataset by specifying its schema, input path, input format, and mapper class. Mobius provides two built-in *DatasetBuilder* methods:

- **com.ebay.erl.mobius.core.builder.TSVDatasetBuilder**
  - Reads any line-oriented, text-based files on HDFS. The columns are delimited by some delimiter; *TSVDatasetBuilder* uses tab as the default delimiter.
  - Each line is delimited by the given delimiter, and it can be changed to other kinds of delimiter.
- **com.ebay.erl.mobius.core.builder.SeqFileDatasetBuilder**
  - Reads any Hadoop sequence file. The user needs to provide a mapper that parses the writable objects into columns in a tuple.

3.2.1 Creating a Customized DatasetBuilder Class

If the data source is on HDFS and the built-in dataset builders cannot represent the data you have, extend *com.ebay.erl.mobius.builder.AbstractDatasetBuilder*. Create customized *Dataset* and *com.ebay.erl.mobius.core.mapred.AbstractMobiusMapper* methods as well. Here is an example.

```
package com.ebay.erl.mobius.edw;

import java.io.IOException;
import java.sql.Timestamp;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;

import com.ebay.erl.mobius.core.MobiusJob;
import com.ebay.erl.mobius.core.builder.AbstractDatasetBuilder;
import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.mapred.AbstractMobiusMapper;
import com.ebay.erl.mobius.core.model.Tuple;
import com.ebay.erl.mobius.util.CaseInsensitiveTreeMap;

@SuppressWarnings("deprecation")
public class MyWebLogDatasetBuilder extends AbstractDatasetBuilder<MyWebLogDatasetBuilder>{

	protected MyWebLogDatasetBuilder(MobiusJob aJob, String datasetName)
	{
		super(aJob, datasetName);
	}

	@Override
	protected Dataset newDataset(String datasetName)
	{
		Dataset newDataset = new MyWebLogDataset(this.mobiusJob, datasetName);
		return newDataset;
	}


	public static MyWebLogDatasetBuilder newInstance(MobiusJob job, String name)
		throws IOException
	{
			MyWebLogDatasetBuilder builder = new MyWebLogDatasetBuilder(job, name);

			// my web log has fixed schema, {LEVEL, TIME, PAGE, URL, PARAMETERS},
			// LEVEL is a string, like WARNING, INFO...etc
			// TIME is a string in the format of yyyy-mm-dd HH:MM:ss,
			// PAGE is a string, URL is a string, PARAMETERS is an unbounded
			// k=v pairs.
			//
			// here are some web log entries:
			// INFO 2011-12-01 13:12:11 home      /homepage      k1=v1
			// INFO 2011-12-01 13:15:11 page1     /page1/xyz     k1=v1   k2=v2

			builder.setSchema(new String[]{"LEVEL", "TIME", "PAGE", "URL",
"PARAMETERS"});
			return builder;
	}

}
```

```
@SuppressWarnings("deprecation")
class MyWebLogDataset extends Dataset
{
	private static final long serialVersionUID = 32208064940124248361;

	protected MyWebLogDataset(MobiusJob job, String name)
	{
		super(job, name);

		// fix the mapper to MyWebLogMapper
		this.setMapper(MyWebLogMapper.class);

		// fix the input format to TextInputFormat
		this.setInputFormat(TextInputFormat.class);
	}
}

@SuppressWarnings("deprecation")
class MyWebLogMapper extends AbstractMobiusMapper<Writable, Text>
{
	private String[] schema;

	@Override
	public void configure(JobConf conf)
	{
		super.configure(conf);

		// schema set in the MyWebLogDatasetBuilder#newInstance
		// can be always retrieved in this way.
		this.schema = this.conf.get(this.getDatasetID()+".schema", "").split(",");
	}

	@Override
	public Tuple parse(Writable inkey, Text invalue)
			throws IllegalArgumentException, IOException
	{
		Tuple record = new Tuple();

		// the log filed is delimited by tab
		String[] values = invalue.toString().split("\t", -1);

		int bound = Math.min(values.length, this.schema.length);

		// values for the first 4 columns always presented
		record.put("LEVEL", values[0]);
		record.put("TIME", Timestamp.valueOf(values[1]));
		record.put("PAGE", values[2]);
		record.put("URL", values[3]);

		// the PARAMETERS part is dynamic
		CaseInsensitiveTreeMap params = new CaseInsensitiveTreeMap();
		for( int i=bound;i<values.length;i++ )
		{
			String[] kv           = values[i].split("=", -1);
			String key            = kv[0];
			String value   = kv[1];

			params.put(key, value);
		}
		record.put("PARAMETERS", params);

		return record;
	}
}
```

3.3 Tuple

Every record in a dataset is called a *tuple* and is represented as *com.ebay.erl.mobius.core.model.Tuple*. A tuple has one to many columns, and its schema is the same as the schema of the dataset it belongs to. Each column in a tuple has a name and value. The name of a column is a case-insensitive string. The user uses the name to retrieve the value of that column. The value of a column can be one of several types, listed below.

3.3.1 Supported Types

Type name	In Java
Numerical types	*byte, short, int, long, float, double*
Date types	*java.sql.Timestamp, java.sql.Time, java.sql.Date*
String type	*java.lang.String*
Boolean type	*boolean*
Null type	*null*
Tuple type	*com.ebay.erl.mobius.core.model.Tuple*
Byte array type	*byte[]*
String map type	*java.util.Map<String, String>*
Writable type	*org.apache.hadoop.io.Writable*
Null writable type	*org.apache.hadoop.io.NullWritable*
Serializable type	*java.io.Serializable*

3.3.2 Setting or Retrieving the Values in Tuple

The name of the column is in the format of the following Java regular expression: *([\p{Graph}&&[^\.]]+)(\.([\p{Graph}&&[^\.]]+))?*. Normally, the red part of this expression is sufficient to provide the value of the column. If the green part is included, Mobius checks whether the value type is a *String map type*. If it is a String map type, then Mobius first uses the red part to locate the map, then uses the green part as the key to the hash map to retrieve the corresponding value.

Users can use the name and the *getters* provided by the Tuple class to retrieve the value, and use the *setters* to set the value of a column.

3.4 MobiusJob

com.ebay.erl.mobius.core.MobiusJob is the main class users can extend to compose a data analysis job. By extending this class, the user builds datasets, manipulating them through a series of operations like filtering, joining, or grouping iteration to obtain the final result. Depending on analysis flow, *MobiusJob* usually translates into one to many MapReduce iterations. As mentioned in session one, the dependencies of those MapReduce tasks are automatically managed by the Mobius engine; tasks without dependencies are submitted in parallel, and those with dependencies are blocked until all the dependencies are finished successfully.

Note that if a *MobiusJob* class is translated into more than one MapReduce tasks, and one of the MapReduce task fails during processing, then the sub tasks that depend on it also fail. The current

implementation of the Mobius engine does not resume from the failure point, so the user has to rerun the job from the beginning if a failure occurs.

3.5 MobiusJobRunner

com.ebay.erl.mobius.core.MobiusJobRunner is submits Mobius jobs. The *MobiusJobRunner* class uses *org.apache.hadoop.mapred.jobcontrol.JobControl* to submit the translated Hadoop jobs in a Mobius job. Here is an example of the correct way to submit a Mobius job.

```
package example;

import org.apache.hadoop.fs.Path;

public class MyAnalysis extends MobiusJob {

	private static final long serialVersionUID = -7216853302346293360L;

	@Override
	public int run(String[] args) throws Exception {

		//
		// bulid your analysis flow here
		//

		return 0;
	}

	public static void main(String[] args) throws Throwable {
		int exit = MobiusJobRunner.run(new MyAnalysis(), args);
		System.exit(exit);
	}
}
```

Always extend *com.ebay.erl.mobius.core.MobiusJob* and always use *com.ebay.erl.mobius.core.MobiusJobRunner* to submit the job, as shown in the example for the *main* method.

4.0 GETTING STARTED

4.1 Extending MobiusJob

To create a Mobius job, the first step is to simply extend the *com.ebay.erl.mobius.core.MobiusJob* class and write a main function, as shown below:

```
package example;

import org.apache.hadoop.fs.Path;

public class MyAnalysis extends MobiusJob {

	private static final long serialVersionUID = -7216853023462933 60L;

	@Override
	public int run(String[] args) throws Exception {

		//
		// bulid your analysis flow here
		//

		return 0;
	}

	public static void main(String[] args) throws Throwable {
		int exit = MobiusJobRunner.run(new MyAnalysis(), args);
		System.exit(exit);
	}
}
```

It is important to use *MobiusJobRunner* to execute *MobiusJob*, as specified in the main method of the example above. Otherwise, *MobiusJob* cannot be executed properly.

Next, build datasets to analyze the Mobius job. If the built-in *DatasetBuilder* methods fit your data format, use one of those. Otherwise, refer to 3.2.1 Create a Customized DatasetBuilder to create your own *DatasetBuilder* class.

4.2 Building a Dataset

Here is an example of how to build comma-separated datasets using *TSVDatasetBuilder*. The data source sits on HDFS and is in text format.

```
package example;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.MobiusJob;
import com.ebay.erl.mobius.core.MobiusJobRunner;
import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;

public class MyAnalysis extends MobiusJob {

	private static final long serialVersionUID = -7216853023462933 60L;

	@Override
	public int run(String[] args) throws Exception {

		////////////////////////////////////////
		// build acomma separated dataset using
		// the TSVDatasetBuilder
		////////////////////////////////////////
		Dataset myds = TSVDatasetBuilder.newInstance(
				this,
				// give a name to this dataset
				"my_dataset",
				// specify the schema of the dataset
				new String[]{"ID", "PRICE", "UNIT"})

			// the underline data is delimited by comma,
			// set the delimiter to override the default
			// one, which is tab.
			.setDelimiter(",")

			// specify the path on HDFS where stores the
			// my_dataset data.
			.addInputPath(new Path("$INPUT_PATH_ON_HDFS"))

		// finish the building, call build to build the dataset,
		// it will perform some validation here, ex: throws
		// IOException when the specified $INPUT_PATH_ON_HDFS
		// doesn't exist.
		.build();

		return 0;
	}

	public static void main(String[] args) throws Throwable
	{
		// execute MyAnalysis using MobiusJobRunner
		int exit = MobiusJobRunner.run(new MyAnalysis(), args);
		System.exit(exit);
	}
}
```

This example is of building one dataset. For some analysis cases, users can build more than one dataset, and then work with them using joining and sorting.

4.3 Dynamically Computed Columns

It is often useful to compute values by reading a record from the source. This type of operation is quite common in the mapper phase. Mobius provides the *addComputedColumn(ComputedColumns)* method in each *DatasetBuilder* class to allow for this kind of computation.

An instance of the *ComputedColumns* class is always executed in the map phase. When the user declares a *ComputedColumns* class during the building phase, every record includes those computed columns during runtime, except in certain conditions, such as if the user doesn't call the *output* function within

ComputedColumns. In this case, the records include only the original columns. The computed columns become part of the dataset and are selectable for projection.

4.3.1 Adding One Computed Column

The example below computes the total price for each record, using the formula *PRICE * UNIT*.

```
..............
..............

Dataset myds = TSVDatasetBuilder.newInstance(
        /*
         * building part in secion 4.2
         * ..............
         */
        .addComuptedColumn(new ComputedColumns("TOTAL_PRICE"){
                // create an instance of ComputedColumns to compute
                // total price.
                // ComputedColumns will be serialized into Base64
                // string, make sure all the fields are serializable,
                // or make them transient.
                private static final long serialVersionUID = -4603749995059372757L;

                // make it transient as we don't need to serialize this object,
                // create it when it is required.
                private transient DecimalFormat price_format;

                @Override
                public void consume(Tuple newRow) {
                        if( price_format==null ) {
                                price_format = new DecimalFormat("###,###.##");
                        }
                        float price    = newRow.getFloat("PRICE", 0.0F);
                        int unit       = newRow.getInt("UNIT", 0);
                        float total    = price*(float)unit;

                        // create a tuple which contains the exact
                        // same schema as we create this ComputedColumns,
                        // (specified in constructor), in this case, it
                        // has only one column: TOTAL_PRICE
                        Tuple t = new Tuple();
                        t.put("TOTAL_PRICE", price_format.format(total));

                        // output this tuple, so each record will also has
                        // one additional column named "TOTAL_PRICE"
                        this.output(t);
                }
        })
.build();

..............
..............
```

4.3.2 Adding More than One Computed Column

ComputedColumns can have multiple columns, if the user provides more than one column name in the constructor of a *ComputedColumns* method, and uses the same schema as the constructor in the outputted tuple. Using the example above, modify it as follows to compute another column that indicates if the unit is greater than 1,000.

```
Dataset myds = TSVDatasetBuilder.newInstance(
        /*
         * building part in secion 4.2
         * ..............
         */
        .addComuptedColumn(new ComputedColumns("TOTAL_PRICE", "IS_BIG_ORDER"){

                // create an instance of ComputedColumns to compute
                // total price.
                //
                // ComputedColumns will be serialized into Base64
                // string, make sure all the fields are serializable,
                // or make them transient.

                private static final long serialVersionUID = -4603749995059372757L;

                // make it transient as we don't need to serialize this object,
                // create it when it is required.
                private transient DecimalFormat price_format;

                @Override
                public void consume(Tuple newRow){
                        if( price_format==null ){
                                price_format = new DecimalFormat("###,###.##");
                        }

                        float price    = newRow.getFloat("PRICE", 0.0F);
                        int unit       = newRow.getInt("UNIT", 0);
                        float total    = price*(float)unit;

                        // create a tuple which contains the exact
                        // same schema as we create this ComputedColumns,
                        // (specified in constructor), in this case, it
                        // has only one column: TOTAL_PRICE
                        Tuple t = new Tuple();
                        t.put("TOTAL_PRICE", price_format.format(total));
                        t.put("IS_BIG_ORDER", unit>=1000);

                        // output this tuple, so each record will also has
                        // one additional column named "TOTAL_PRICE"
                        this.output(t);
                }
        })
.build();
```

4.3.3 Outputting Multiple Rows per Record

Some use cases require generating multiple outputs per input record, such as computing the most frequently used words in movie titles from a movie inventory, as shown in the example below.

```
////////////////////////////////////////
// build a comma separated dataset using
// the TSVDatasetBuilder
////////////////////////////////////////
Dataset movies = TSVDatasetBuilder.newInstance(
                this,
                // give a name to this dataset
                "movies",
                // specify the schema of the dataset
                new String[]{"ID", "MOVIE_TITLE"})
        .addInputPath(new Path("$INPUT_PATH_ON_HDFS"))
        .addComuptedColumn(new ComputedColumns("TITLE_TOKEN"){
                private static final long serialVersionUID = -4603749995059372757L;
                @Override
                public void consume(Tuple newRow)
                {
                        String[] tokens = newRow.getString("MOVIE_TITLE").split("\\p{Space}+");
                        for( String aToken:tokens )
                        {
                                Tuple t = new Tuple();
                                t.put("TITLE_TOKEN", aToken);
                                this.output(t);
                        }
                }
        })
.build();
```

When a *ComputedColumns* method emits multiple rows (the *output* method is called multiple times per row), each record in the original dataset performs cross product to the result of *ComputedColumns*. In the example above, one entry uses *100000* as the movie ID and *Mission Impossible Ghost Protocol* as the movie title, and this row becomes four rows, as shown below.

ID	MOVIE_TITLE
10000	Mission Impossible Ghost Protocol

→

ID	MOVIE_TITLE	TITLE_TOKEN
100000	Mission Impossible Ghost Protocol	Mission
100000	Mission Impossible Ghost Protocol	Impossible
100000	Mission Impossible Ghost Protocol	Ghost
100000	Mission Impossible Ghost Protocol	Protocol

Users can then select *TITLE_TOKEN* only from the cross-product result to compute the frequency. Note that only the selected columns are transferred into the reducer.

4.4 Filtering Records in a Dataset

A common operation during an analysis task is removing the records that are not required in the analysis flow. During the dataset building phase, users can specify filters to remove certain records from the dataset, and those filters are applied during the map phase. Here is a filtering example.

```
Dataset myds = TSVDatasetBuilder.newInstance(
                this,
                // give a name to this dataset
                "my_dataset",
                // specify the schema of the dataset
                new String[]{"ID", "PRICE", "UNIT"})

                // the underline data is delimited by comma,
                // set the delimiter to override the default
                // one, which is tab.
        .setDelimiter(",")

        // specify the path on HDFS where stores the
        // my_dataset data.
        .addInputPath(new Path("$INPUT_PATH_ON_HDFS"))

        // only select records with price > 20.0
        .constraint(TupleRestrictions.gt("PRICE", 20.0D))
        // and unit > 10
        .constraint(TupleRestrictions.gt("UNIT", 10))
.build();
```

This example only selects records using the criteria of *PRICE>20.0 AND UNIT>10*. Users can build a more complex composited filter using *com.ebay.erl.mobius.core.criterion.LogicalExpression*. The example below shows how to build a filter using the criteria of *(PRICE>20.0 AND UNIT>10)* or *(PRICE>100.0 AND UNIT<=10)*.

```
// build PRICE>20.0 AND UNIT>10
TupleCriterion c1 =
                TupleRestrictions.gt("PRICE", 20.0D)
                        .and(TupleRestrictions.gt("UNIT", 10));

// build PRICE>100.0 AND UNIT<=10
TupleCriterion c2 =
                TupleRestrictions.gt("PRICE", 100.0D)
                        .and(TupleRestrictions.le("UNIT", 10));

// build (PRICE>20.0 AND UNIT>10) OR (PRICE>100.0 AND UNIT<=10)
TupleCriterion criteria = c1.or(c2);

////////////////////////////////////////
// build acomma separated dataset using
// the TSVDatasetBuilder
////////////////////////////////////////
Dataset myds = TSVDatasetBuilder.newInstance(
                this,
                // give a name to this dataset
                "my_dataset",
                // specify the schema of the dataset
                new String[]{"ID", "PRICE", "UNIT"})

        // the underline data is delimited by comma,
        // set the delimiter to override the default
        // one, which is tab.
        .setDelimiter(",")

        // specify the path on HDFS where stores the
        // my_dataset data.
        .addInputPath(new Path("$INPUT_PATH_ON_HDFS"))
        .constraint(criteria)
.build();
```

A *TupleCriterion* method can also be applied to columns defined in *ComputedColumns*.

4.5 Running an Analysis

When a dataset is built, users can start the analysis. Mobius supports common operations like listing, joining, grouping, and sorting, which are described in the sections below.

4.5.1 Using Listing

Listing is an operation that saves the user-specified columns from a dataset to a destination. It's a *map*-only job, with no *reduce* operation. *Listing* is often used with *ComputedColumns* when an analysis only requires one dataset. Here is an example.

```
package example;

import .....

public class MyAnalysis extends MobiusJob {

	private static final long serialVersionUID = -7216853302346293360L;

	@Override
	public int run(String[] args) throws Exception {


		// build PRICE>20.0 AND UNIT>10
		TupleCriterion c1 = .....;

		// build PRICE>100.0 AND UNIT<=10
		TupleCriterion c2 = .....;

		// build (PRICE>20.0 AND UNIT>10) OR (PRICE>100.0 AND UNIT<=10)
		TupleCriterion criteria = c1.or(c2);


		Dataset myds = TSVDatasetBuilder.newInstance(
				this,
				// give a name to this dataset
				"my_dataset",
				// specify the schema of the dataset
				new String[]{"ID", "PRICE", "UNIT"})
			.setDelimiter(",")

			.addComuptedColumn(new ComputedColumns("TOTAL_PRICE", "IS_BIG_ORDER"){


				// same as section 4.3.2
			})
			// specify the path on HDFS where stores the
			// my_dataset data.
			.addInputPath(new Path("$INPUT_PATH_ON_HDFS"))
			.constraint(criteria)
		.build();


		// each operation will return a new dataset with
		// the specified columns as its schema.  The
		// returned dataset can be used for further
		// analysis if necessary.
		Dataset d1 = this.list(myds,
				new Path("$PATH_TO_OUTPUT_ON_HDFS"),
				new Column(myds, "ID"),
				new Column(myds, "UNIT"),
				new Column(myds, "IS_BIG_ORDER"),
				new Column(myds, "TOTAL_PRICE"));

		return 0;
	}

	public static void main(String[] args) throws Throwable
	{
		// execute MyAnalysis using MobiusJobRunner
		int exit = MobiusJobRunner.run(new MyAnalysis(), args);
		System.exit(exit);
	}
}
```

4.5.2 Joining Jobs

A *joining* job involves both *map* and *reduce* operations. Joining requires at least two datasets, and the participating datasets must have one or more columns to be used as the join key. Mobius currently supports inner join, left outer join, and right outer join, and only equi-join is supported by Mobius.

Join or group-by jobs (discussed later in this document) all involve one complete MapReduce phase. Join jobs require at least two datasets, and group-by jobs can only work on one dataset. Different kinds of aggregation operations, such as MAX, MIN, or MEDIUM can be applied in join/group-by jobs. These operations are executed in the reducer side. Mobius allows users to provide customized projections (such as aggregation operations) for customized *reduce* operations. See the customized projections section for more information.

4.5.2.1 Inner Join

The example below shows an inner join. A best practice to improve performance is to put the larger dataset (measured by the number of values within a join key, NOT the total number of values for a dataset) on the right-hand side, as shown in the highlighted section. In this example, each *person* can have more than one *order*, but each *order* belongs to one *person* only, so the *order* is on the right-hand side and *person* is on the left-hand side.

```
package example;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.TextOutputFormat;

import com.ebay.erl.mobius.core.JoinOnConfigure.EQ;
import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.model.Column;

@SuppressWarnings("deprecation")
public class InnerJoinTest extends MobiusJob
{
	private static final long serialVersionUID = -5045003345238678287L;

	@Override
	public int run(String[] args)
		throws Exception
	{
		String orderInputPath  = "$ORDER_PATH";// HDFS path to the order dataset
		String personInputPath = "$PERSON_PATH";// HDFS path to the persone dataset

		// building order dataset
		Dataset order = TSVDatasetBuilder.newInstance(
				this,
				"orders",
				new String[]{"O_Id", "OrderNo", "P_Id"}
			)
			.addInputPath(new Path(orderInputPath))
		.build();

		// building person dataset
		Dataset person = TSVDatasetBuilder.newInstance(
				this,
				"persons",
				new String[]{"P_Id", "LastName", "FirstName", "Address", "City"}
			)
			.addInputPath(new Path(personInputPath))
		.build();

		this
			.innerJoin(person, order)
			.on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
			.save(this,
					new Path("$PATH_TO_OUTPUT")
					,new Column(person, "P_Id")
					,new Column(person, "FirstName")
					,new Column(person, "LastName")
					,new Column(order, "OrderNo")
					,new Max(new Column(order, "OrderNo"))
			);

		return 0;
	}

	public static void main(String[] args)
		throws Throwable
	{
		int exit = MobiusJobRunner.run(new InnerJoinTest(), args);
		System.exit(exit);
	}

}
```

As mentioned previously, Mobius only supports equi-join and uses *com.ebay.erl.mobius.core.JoinOnConfigure.EQ* to specify the relationship. For each type of join, there must be at least one *EQ* object specified in the *on* statement. Users can specify more than one *EQ* by appending them within the *on* statement.

When joining datasets with more than one *EQ*, the *EQ*s are concatenated together with *AND* to form the join condition. Mobius only supports *AND* in current implementation. For example, the following codes are equal to the join relation of *DS1.COL1=DS2.COL1 AND DS1.COL2=DS2.COL3*:

```
on(
        new EQ( new Column(DS1, "COL1"), new Column(DS2, "COL1") ),
        new EQ( new Column(DS1, "COL2"), new Column(DS2, "COL3") )
)
```

4.5.2.2 Left Outer Join

A left outer join joins two datasets, taking all the records from the left dataset even if there is no record matched to the right dataset.

The left outer join is only slightly different from an inner join.

```
package example;

import …;

@SuppressWarnings("deprecation")
public class InnerJoinTest extends MobiusJob
{
        private static final long serialVersionUID = -5045003345238678287L;

        @Override
        public int run(String[] args)
                throws Exception
        {
                // building order dataset
                Dataset order = …;

                // building person dataset
                Dataset person = …;

                this
                        .leftOuterJoin(person, order)
                        .on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
                        .save(this,
                                        new Path("$PATH_TO_OUTPUT")
                                        ,new Column(person, "P_Id")
                                        ,new Column(person, "FirstName")
                                        ,new Column(person, "LastName")
                                        ,new Column(order, "OrderNo")
                        );

                return 0;
        }

        public static void main(String[] args)
                throws Throwable
        {
                int exit = MobiusJobRunner.run(new InnerJoinTest(), args);
                System.exit(exit);
        }

}
```

When there is no record from the *order* dataset for a *P_Id*, *OrderNo* is an empty string when using *TextOutputFormat* as the output format (the default for *save* operations), or null when using *SequenceFileOutputFormat* (the default for *build* operations).

To change the default replacement for no matched records, provide a third argument in the *innerJoin* method:

```
this
        .leftOuterJoin(person, order, “N/A”)
        .on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
```

4.5.2.3 Right Outer Join

A right outer join is almost the same as a left outer join, except that the method call is different. The right outer join takes all the records from the right dataset even if there is no record from the left dataset for a given join key.

In this example, all the records from the *order* dataset are outputted, even if there is no record from the *person* dataset in a *P_Id*.

```
package example;

import …;

@SuppressWarnings("deprecation")
public class InnerJoinTest extends MobiusJob
{
        private static final long serialVersionUID = -5045003345238678287L;

        @Override
        public int run(String[] args)
                throws Exception
        {
                // building order dataset
                Dataset order = …;

                // building person dataset
                Dataset person = …;

                this
                        .rightOuterJoin(person, order)
                        .on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
                        .save(this,
                                        new Path("$PATH_TO_OUTPUT")
                                        ,new Column(person, "P_Id")
                                        ,new Column(person, "FirstName")
                                        ,new Column(person, "LastName")
                                        ,new Column(order, "OrderNo")
                        );

                return 0;
        }

        public static void main(String[] args)
                throws Throwable
        {
                int exit = MobiusJobRunner.run(new InnerJoinTest(), args);
                System.exit(exit);
        }

}
```

4.5.3 Group-By

A group-by job works on a single dataset, and groups the records by one to many columns in the dataset.

```
package example;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.function.Max;
import com.ebay.erl.mobius.core.model.Column;

public class GroupByTest extends MobiusJob {
	private static final long serialVersionUID = 5217672582768153239L;

	@Override
	public int run(String[] args) throws Exception {
		Dataset order = TSVDatasetBuilder.newInstance(this, "orders",
				new String[]{"O_Id", "OrderNo", "P_Id"})
			.addInputPath(new Path("$INPUT_PATH"))
		.build();

		Dataset grouping_result = this
			.group(order).by("P_ID")
			.save(this,
					new Path("$OUTPUT_PATH"),
					new Column(order, "P_ID"),
					new Counts(new Column(order, "O_Id"))
			);
		return 0;
	}

	public static void main(String[] args)
		throws Throwable {
		int exit = MobiusJobRunner.run(new GroupByTest(), args);
		System.exit(exit);
	}
}
```

As with join jobs (inner or outer joins), users can apply aggregation operations in the *save* operation. The example above asks how many orders (by *O_Id*) were ordered by a person for a given person ID (*P_Id*).

After grouping a dataset users often sort it based on certain columns. Mobius can easily chain the operation as follows :

```
package example;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.function.Counts;
import com.ebay.erl.mobius.core.model.Column;
import com.ebay.erl.mobius.core.sort.Sorter;
import com.ebay.erl.mobius.core.sort.Sorter.Ordering;

public class GroupByTest extends MobiusJob {

	private static final long serialVersionUID = 52176725827681532 39L;

	@Override
	public int run(String[] args) throws Exception {
		Dataset order = ...;

		Dataset result =
			this
			.group(order)
			.by("P_ID")
			// use build instead of save as we don't need
			// to save the intermediate result.
			.build(this,
					new Column(order, "P_ID"),
					new Counts(new Column(order, "O_Id"))
			)
			// after the grouping done, sort the grouping
			// result by number of orders in descending order.
			.orderBy(
				// save the sorting result to $OUTPUT on HDFS
					new Path("$OUTPUT"),

				// the name of new Counts(new Column(order, "O_Id"))
				// is Counts_O_Id by default (simple class name plus
				// underscore plus the column name). So, when try
				// to sort the result based on the number of
				// orders, we need to refer to the name of that
				// column as "Counts_O_Id".
					new Sorter("Counts_O_Id", Ordering.DESC));
		return 0;
	}

	public static void main(String[] args) throws Throwable {
		int exit = MobiusJobRunner.run(new GroupByTest(), args);
		System.exit(exit);
	}
}
```

4.5.4 Sorting

Mobius uses the total sort implementation described in Tom White's "Hadoop: The Definitive Guide." This type of sorting frees the task so that it is not bound to single reducer. This process can be extremely slow especially when the dataset is large.

As long as the data source can be transformed into a Mobius Dataset, either from a dataset builder or from the intermediate results of a joining job, for example, it can be sorted in the following manner:

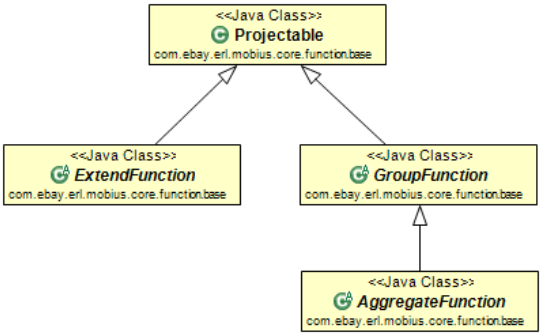
```
package example;

import org.apache.hadoop.fs.Path;

import com.ebay.erl.mobius.core.builder.Dataset;
import com.ebay.erl.mobius.core.builder.TSVDatasetBuilder;
import com.ebay.erl.mobius.core.sort.Sorter;
import com.ebay.erl.mobius.core.sort.Sorter.Ordering;

public class SortTest extends MobiusJob {
	private static final long serialVersionUID = 7945209428566552518L;

	@Override
	public int run(String[] args) throws Exception {

		Dataset people = TSVDatasetBuilder.newInstance(
				this,
				"people",
				new String[]{"fname","lname","age","zipcode","gender"}
			)
			.setDelimiter(",")
			.addInputPath(new Path("$INPUT"))
		.build();

		this
			// specify which dataset we want to sort
			.sort(people)
			// select the columns from the dataset
			// we would like to save in the output
			.select("age", "gender", "fname", "lname")
			// specify the sorting columns, note that,
			// only the columns within the "select"
			// list can be used in Sorters.
			.orderBy(
					new Sorter("age", Ordering.ASC, true),
					new Sorter("gender", Ordering.DESC))
			.save(
					this,
					// save the result to $OUTPUT
					new Path("$OUTPUT")
			);
		return 0;
	}

	public static void main(String[] args) throws Throwable {
		int exit = MobiusJobRunner.run(new SortTest(), args);
		System.exit(exit);
	}
}
```

Users can chain sorting operations after grouping or joining jobs. See the second example in the Group-By section for an illustration.

4.6 Customized Projections

Aggregation operations are a type of projection operation, which take all the records within a group and then produce one output. Previous examples in this document have illustrated joining or grouping jobs invoking aggregation operations, such as MAX or COUNTS.

Users might also want to use customized projections, such as those in their reducer classes. Here's an in-depth example of customized projections for the *com.ebay.erl.mobius.function.base* package.



Note that only joining and grouping jobs allow projections other than *com.ebay.erl.mobius.core.model.Column*.

4.6.1 Projectable

com.ebay.erl.mobius.core.function.base.Projectable is the base class for all projection operations. Users can't extend this class, but can extend ExtendFunction, GroupFunction, and their sub-classes.

A projectable takes one to many columns from one to many datasets as inputs, then performs certain calculations to generate X number of rows as the output, where X can be zero to many. The schema of each outputted row is defined in the *setOutputSchema* method. When providing a customized implementation of a projection, make sure the output's schema is consistent with the ones specified in the *setOutputSchema*.

4.6.2 ExtendFunction

An extend function (*com.ebay.erl.mobius.core.function.base.ExtendFunction*) takes one row at a time within a group as its input, and produces one row as the output. In other words, each output of an *ExtendFunction* method can be decided by just one input row, and the output is not related to other records in a group. As a result, an *ExtendFunction* object can't access all the records in a group.

The output of an extend function can be one to many columns. An example of extend function would be a general function to perform A+B where the value of both A and B columns are numeric and the output of the function is the result of A+B.

The computation logic is implemented in the method *getResult(Tuple)* in the *ExtendFunction* class.

4.6.3 GroupFunction

A group function (com.ebay.erl.mobius.core.function.base.GroupFunction) takes all the records within a group first, and then based on the inputs, produces X number of rows as the output, where X can be zero to many.

When a new record is added, the *consume(Tuple)* method is called, then after all the records within the group are iterated through, the output of the group function is retrieved by the Mobius engine via the *getResult()* method.

After the *getResult()* method is called, *reset()* is called by the Mobius engine. If any customized intermediate states need to be cleaned up, override this method. When overriding this method in a sub-class, it is important to call *super.reset()* as failure to do so produces an incorrect result.

An implementation of a group function might not need all the records within a group before it can calculate the final result. In this case, Mobius still feeds all the records to the *consume* method, and the implementer can choose to ignore certain records if the result can be computed without them.

An example group function is the *com.ebay.erl.mobius.core.function.Top* function, which takes all the records in a group, and returns only the top X rows in the group. X is specified by the user and the user can also provide a customized comparator to override the default natural ordering.

4.6.4 AggregateFunction

An aggregate function is a specific type of group function that takes all the records in a group, and outputs only one row as the output. The output columns of an aggregate function can be still one to many columns.

To implement an aggregate function, override both the *consume(Tuple)* and *getComputedResult()* methods and implement the logic in these two methods. The *consume* method is called by Mobius every time there is a new record in a group. Users can store some partially computed results during this stage. The *getComputedResult()* method returns one tuple and users should implemented the logic of providing the final result based on these partial results. Note that the schema of the returned tuple is the same as the schema that Mobius retrieved from *getOutputSchema*.

4.6.4.1 SingleInputAggregateFunction

Most of the built-in aggregate functions in Mobius do not directly extend *com.ebay.erl.mobius.core.function.base.AggregateFunction*, but instead extend *com.ebay.erl.mobius.core.function.base.SingleInputAggregateFunction*.

A single input aggregate function is a special case of aggregate function. It only takes a single column from a dataset as its input and consumes all the records in a group to generate one single row as its output. This class enforces that the user cannot provide more than one column to the constructor.

Here is an example of an implementation of *SingleInputAggregateFunction*, the built in *Counts* class provided by Mobius

```java
package com.ebay.erl.mobius.core.function;

import com.ebay.erl.mobius.core.function.base.SingleInputAggregateFunction;
import com.ebay.erl.mobius.core.model.Column;
import com.ebay.erl.mobius.core.model.Tuple;

/**
 * To count the number of records of the given <code>inputColumn</code>
 * within a group, count only if the value of the given <code>inputColumn</code>
 * is not null.
 */
public class Counts extends SingleInputAggregateFunction {

	private static final long serialVersionUID = -8406996639392091020L;

	private long counts = 0L;

	public Counts(Column inputColumn)
	{
		super(inputColumn);
	}

	@Override
	public void consume(Tuple tuple)
	{
		Object value = tuple.get(this.inputColumnName);
		if( value!=null )
		{
			counts++;
		}
	}

	@Override
	protected Tuple getComputedResult()
	{
		this.aggregateResult = new Long(this.counts);
		return super.getComputedResult();
	}

	@Override
	public void reset()
	{
		super.reset();
		this.counts = 0L;
	}
}
```

4.7 Chaining Jobs

All the built-in data process operations in Mobius, such as listing, joining, grouping, and sorting, return a dataset that represents the result of the executed operations. The returned dataset can be used directly in different kinds of operations, such as first run a join job, and then the returned dataset sorted by a certain column.

The second example in the group-by section shows one way to sort the result of a group-by job, and this same paradigm can be used for all datasets.

Another way to chain jobs is illustrated below.

```
package example;

import ...;

public class InnerJoinTest extends MobiusJob
{
	private static final long serialVersionUID = -5045003345238678287L;

	@Override
	public int run(String[] args)
		throws Exception
	{
		Dataset order = ...;

		Dataset person = ...;

		Dataset intermediate_result = this
			.innerJoin(order, person)
			.on(new EQ( new Column(order, "P_Id"), new Column(person, "P_Id")))
			.build(this
					,new Column(person, "P_Id")
					,new Column(person, "FirstName")
					,new Column(person, "LastName")
					,new Column(order, "OrderNo")
					,new Max( new Column(order, "OrderNo"))
					,new Min( new Column(order, "OrderNo"))
			);

		this
			.sort(intermediate_result)
			.select("P_Id", "FirstName", "LastName", "OrderNo", "Max_OrderNo",
"Min_OrderNo")
			.orderBy(new Sorter("OrderNo", Ordering.ASC, true))
			.save(this, new Path("$OUTPUT"));

		return 0;
	}

	public static void main(String[] args)
		throws Throwable
	{
		int exit = MobiusJobRunner.run(new InnerJoinTest(), args);
		System.exit(exit);
	}

}
```

In this example, the *sort* operation is blocked until the *innerJoin* operation has completed successfully. The *intermediate_result* can be used in more than one operation, so it can be sorted as well as joined with another dataset to produce a certain result. In this case, the operation results in three MapReduce jobs: the first job generates the intermediate result, the second job sorts the result, and the third job joins the result with another dataset. The second and third jobs are blocked by the first job. Then after the first job has completed successfully, the second and third jobs are submitted in parallel to the Hadoop job queue.

4.8 Miscellaneous

4.8.1 Difference Between save and build Operations

There are two different kinds of storing operation in Mobius, *save* and *build*.

By default, a *build* operation stores the result in binary format sequence files (via *org.apache.hadoop.mapred. SequenceFileOutputFormat*), preserving the native type of values in tuples, such as map, date, or byte array. In contrast, a *save* operation converts the tuple values into their string representations and saves them in text format (via *org.apache.hadoop.mapred.TextOutputFormat*). This

means that when a user chooses the *save* operation, and wants to process the result further, the result must be deserialized back from string representations and cannot be converted back to its original type. For example, if a user saves a byte array column, the string representation would be something similar to "[B@408812F8" so it cannot be converted back from its original value. Users can override the default output format by calling the *build* or *save* operation with *org.apache.hadoop.mapred.FileOutputFormat* as one of the arguments.

Another difference between *build* and *save* is how the results are stored. The *build* operation stores the results in randomly generated temporary folders under *hadoop.tmp.dir*, and the results are deleted after the whole analysis flow is completed (either successfully, killed, or failed). The *save* operation saves the results in the folders specified by user and the results are not deleted.

4.8.2 TextOutputFormat or SequenceFileOutputFormat

As discussed in previous section, by default, a *save* operation uses *TextOutputFormat* and a *build* operation uses *SequenceFileOutputFormat* to store the results. Note that when storing the results in *SequenceFileOutputFormat*, Mobius always uses *org.apache.hadoop.io.NullWritable* as the key and Tuple as the value in the sequence files. The columns within the tuple are the ones given by users in the *save* or *build* operation.

Users can process the sequence files generated by Mobius by another framework, but it is important to understand that the ordering of the columns in the tuple is **NOT** in the same ordering as the one specified *save* or *build* operation. The columns are stored according to their name's natural order. The reason for this is that Mobius stores the schema in Hadoop configuration, but users can create tuples with the same schema using different ordering (by inserting the values in a different order than the defined schema). To resolve this problem, Mobius always serializes the values in a tuple according to their schema name order, so they can be deserialized back to the correct schema.

5.0 MOBIUS ECLIPSE PLUGIN

TBA