Splout SQL User Guide

Splout SQL User Guide

Table of Contents

List of Tables

Chapter 1. Splout SQL installation

1.1. Download

You need to have Java >= 1.6 preinstalled. Download a release of Splout from Maven central [http://search.maven.org/#browse%7C-1223190492]:

- **-mr2** version for Hadoop $\Rightarrow$ 2.X (YARN)
- **-mr1** version for Hadoop < 2.X
- Others (cdh5, etc) for your particular Hadoop distribution

Tip

Other distros: We don't provide builds for every existing Hadoop distribution and version. If you need Splout working for your distro, you'll probably need to recompile Splout adapting a few dependencies. You can have a look to cdh5 profile at the different pom.xml to have a reference.

1.2. Environmental variables

You'll need to set properly a few environmental variables (i.e. at ~/.bashrc)

For Hadoop >= 2.X (YARN):

- SPLOUT_HADOOP_COMMON_HOME $\rightarrow$ Pointing to the folder where the hadoop-common-*.jar can be found
- SPLOUT_HADOOP_HDFS_HOME $\rightarrow$ Pointing to the folder where where the hadoop-mapreduce-client-*.jar can be found
- SPLOUT_HADOOP_MAPRED_HOME $\rightarrow$ Pointing to the folder where the hadoop-common-*.jar can be found
- SPLOUT_HADOOP_CONF_DIR $\rightarrow$ **Optional:** Optionally, specify the Hadoop configuration folder (e.g. /etc/hadoop/conf). Will default to SPLOUT_HADOOP_MAPRED_HOME/conf

Example of env variables for Cloudera CDH5 using parcels:

```
export SPLOUT_HADOOP_COMMON_HOME=/opt/cloudera/parcels/CDH/lib/hadoop
export SPLOUT_HADOOP_HDFS_HOME=/opt/cloudera/parcels/CDH/lib/hadoop-hdfs
export SPLOUT_HADOOP_MAPRED_HOME=/opt/cloudera/parcels/CDH/lib/hadoop-mapredu
export SPLOUT_HADOOP_CONF_DIR=/etc/hadoop/conf
```

For Hadoop < 2.X:

HADOOP_HOME must be properly defined pointing to your Hadoop installation folder.

1.3. Start up

Launching the server daemons is as easy as:

```
bin/splout-service.sh qnode start
bin/splout-service.sh dnode start
```

Tip

The daemons generate a .pid file from where they are launched.

Warning

By default, DNode data is written in “dnode-staging” in the same folder where it is launched. Please read carefully the Configuration section to override this default.

It is possible to have more than one QNode and more than one DNode in a single machine, but for that it is important to understand the Configuration and modify a few properties, specially if you launch the services from the same folder.

Logs are stored in logs/ folder from where daemons are launched. Commons-logging [http://commons.apache.org/logging/] is used for logging.

Splout SQL has been tested to perform correctly under both AMD64 and i386 default Amazon AMIs. Splout is compatible with the latest Elastic Map Reduce AMI. If you find any strange problem or issue please contact us or raise a bug in Github [https://github.com/datasalt/splout-db/issues].

> Back to table of contents [user_guide.html]

Chapter 2. The basics

Splout Terminology

- Table: A table in Splout can be seen as a standard database table. We will later see how tables are defined and what are their particularities.
- Tablespace: A tablespace in Splout is a logical union of one or more tables, which are co-partitioned in the same way.
- Deploy: Splout "deploys" data from a Hadoop-compatible file system such as HDFS or S3, meaning that DNodes fetch the appropriate database binary files and save them in their local filesystem. When all DNodes have been coordinated to do so, the version of the database that they are serving changes atomically to the next one that has been fetched.
- Rollback: Splout can "rollback" previous versions if they are kept in the local storage of all DNodes. DNodes may keep up to some number of versions for each tablespace, which is a configurable property (see the Configuration section for that).

2.1. Table definitions

A table schema's is defined the same way a Pangool Tuple schema [http://pangool.net/user-guide/schemas.html] is defined. However, the data types are adjusted to match those which are compatible with SQLite [http://www.sqlite.org/datatype3.html]. The following table shows the correspondence between a Pangool type and the underlying SQLite type used:

Table 2.1. Correspondence between Pangool types and SQLite types used

Pangool type	SQLite type used
INT	INTEGER
LONG	INTEGER
DOUBLE	REAL
FLOAT	REAL
STRING	TEXT
BOOLEAN	INTEGER (0 is false, 1 is true. SQLite doesn't support booleans.)

2.2. Table types and restrictions

A table is either partitioned or replicated to all partitions. However, a tablespace should have at least one partitioned table for the indexer to be able to distribute the data among partitions!

2.3. Partitioning

Partitioning is the basis of Splout and it allows it to balance data before indexing and deploying it.

The most usual case of partitioning is columnar partitioning, meaning that a table is partitioned using one or more columns of its schema.

When more than one table is partitioned in the same tablespace, they must be co-partitioned using the same kind of fields. For example, if a tablespace A contains tables A1, A2 and A3, and A1 and A2 are partitioned tables and A3 is replicated to all partitions, then if A1 is partitioned by a pair of (string, int) columns, then A2 should also be partitioned by a pair of (string, int) columns.

TIP: Note that when a table is partitioned by some columns, Splout just concatenates the value of those columns as a single string. From that point of view, partitioning is a function of a row that returns a string. Therefore, it is also possible to partition using arbitrary functions, for example a javascript function that takes the first two characters of a field. You can check this in the Advanced API section.

2.4. Partitioning and Querying

Because data is explicitly partitioned, the user must also explicitly provide a partitioning key when querying Splout. For example, if a dataset has been partitioned by "customer_id", then the user will provide the appropriated "customer_id" together with the SQL query when querying Splout through its REST interface.

> Back to table of contents [user_guide.html]

Chapter 3. Splout SQL Configuration

Splout uses a dual-configuration method, one file for the defaults and one file for overriding them. The defaults file is bundled in the JAR and loaded from the classpath. All you need to do if you want to override a default is specify it in a new “splout.properties” file. This file must be in SPLOUT_HOME or in any other place from where you will launch the daemons or in any location of the classpath.

This table shows the property names, the explanation of each property and its default value. Properties that start with "qnode" affect the configuration of the QNode service. Properties that start with "dnode" affect the configuration of the DNode service. Properties that start with "fetcher" affect the configuration of the DNode’s fetcher that is used for deploying new data from a remote location (HDFS, S3, etc). Properties that start with "hz" affect the behavior of the coordination system used among the cluster, which is based on Hazelcast [http://hazelcast.com].

Table 3.1. Splout server configuration

qnode.port	The port this QNode will run on.	4412
qnode.port.autoincrement	Whether this QNode should find the next available port in case "dnode.port" is busy or fail otherwise.	true
qnode.host	The host this QNode will run on. Note: localhost will be automatically substituted by the first valid private IP address at runtime.	localhost
qnode.versions.per.tablespace	The number of successfully deployed versions that will be kept in the system (per tablespace).	10
qnode.deploy.seconds.to.check.then	The number of seconds to wait before checking each time if a DNode has failed or if a timeout has occurred in the middle of a deploy.	60
dnode.port	This DNode’s port.	4422
dnode.port.autoincrement	Whether this DNode should find the next available port in case "dnode.port" is busy or fail otherwise.	true
dnode.host	This DNode’s host name. Note: localhost will be automatically substituted by the first valid private IP address at runtime.	localhost
dnode.serving.threads	How many threads will be allocated for serving requests in Thrift’s ThreadPool Server.	64

dnode.data.folder	The data folder that will be used for storing deployed SQL data stores	./dnode-staging
dnode.pool.cache.seconds	The amount of seconds that the DNode will cache SQL connection pools. After that time, it will close them. Because the DNode may receive requests for different versions in the middle of a deployment, we want to expire connection pools after some time (to not cache connection pools that will not be used anymore).	3600
dnode.pool.cache.n.elements	Number of SQL connection pools that will be cached. There will be one SQL connection pool for each tablespace, version and partition that this DNode serves. So this number must not be smaller than the different numbers of tablespace + version + partitions.	128
dnode.deploy.timeout.seconds	The amount of seconds that the DNode will wait before canceling a too-long deployment. Default is 10 hours.	36000
dnode.max.results.per.query	A hard limit on the number of results per each SQL query that this DNode may send back to QNodes. If the limit is hit, an error will be returned.	50000
dnode.handle.test.commands	If set, this DNode will listen for test commands. This property is used to activating responsiveness to some commands that are useful for integration testing: making a DNode shutdown, etc.	false
dnode.max.query.time	Queries that run for more than this time will be interrupted. Must be greater than 1000.	15000
dnode.slow.query.abs.limit	In milliseconds, queries that are slower will be logged with a WARNING and registered as "slow queries" for this DNode’s stats.	2500

dnode.db.connections.per.pool	Size of the connection pool to each partition that this DNode services.	10
dnode.deploy.parallelism	Number of parallel downloads allowed when deploying partitions in a DNode	3
fetcher.s3.access.key	If using S3 fetching, specify here your AWS credentials.	(none)
fetcher.s3.secret.key	If using S3 fetching, specify here your AWS credentials.	(none)
fetcher.temp.dir	The local folder that will be used to download new deployments.	fetcher-tmp
fetcher.download.buffer	The size in bytes of the in-memory buffer used to download files from S3.	1048576
fetcher.hadoop.fs.name	If using Hadoop fetching, the address of the NameNode for being able to download data from HDFS.	(none)
hz.persistent.data.folder	Folder to be used to persist Hazelcast state information needed to persist current version information. If not present, no information is stored, and restarting a cluster will cause it to start without any active tablespace.	hz-data
hz.port	Enable this property if you want your Hazelcast service to bind to an specific port. Otherwise the default Hazelcast port is used (5701), and auto-incremented if needed.	(none)
hz.join.method	Use this property to configure Hazelcast join in one or other way. Possible values: MULTICAST, TCP, AWS	multicast
hz.multicast.group	Uncomment and use this property if method=MULTICAST and fine-tuning is needed.	(none)
hz.multicast.port	Uncomment and use this property if method=MULTICAST and fine-tuning is needed.	(none)
hz.tcp.cluster	Uncomment and use this property if method=TCP. Specify a comma-separated list of host cluster members.	(none)
hz.aws.security.group	Uncomment and use this property if method=AWS and	(none)

	only a certain security group is to be examined.	
hz.aws.key	Don't forget your AWS credentials if you use method=AWS.	(none)
hz.aws.secret	Don't forget your AWS credentials if you use method=AWS.	(none)
hz.backup.count	Modifies the standard backup count. Affects the replication factor of distributed maps.	3
hz.disable.wait.when.joining	Hazelcast waits 5 seconds before joining a member. That is good in production because it improves the possibilities of joining several members at the same time. But very bad for testing… This property allows you to disable it for testing.	false
hz.oldest.members.leading.count	Number of oldest members leading operations in the cluster. Sometimes only these members answer to events, in order to reduce coordination traffic.	3
hz.registry.max.time.to.check.registration	Max time, in minutes, to check if the member is registered. This check is used to assure eventual consistency in rare cases of network partitions where replication was not enough to ensure that no data is lost.	5

3.1. Typical distributed configurations

It is fairly easy to install Splout in a distributed environment. By default, Splout will use Hazelcast [http://hazelcast.com] in Multicast mode for finding members, but it is possible to configure Splout for explicit TCP/IP or Amazon AWS auto-discovery. Following there are some examples of distributed configurations:

Table 3.2. Distributed configurations

Multicast	"hz.join.method=multicast", Activated by default.	Optionally, "hz.multicast.group", "hz.multicast.port" can be used for fine-tuning the configuration.
TCP/IP	"hz.join.method=tcp", "hz.tcp.cluster=192.168.1.3,192.168.1.4"	Only hosts specified in the comma-separated list will be considered for membership.

AWS	"hz.join.method=aws", "hz.aws.key=KEY", "hz.aws.secret=SECRET"	Using the provided credentials, active hosts in AWS will be considered for membership. The list of hosts can be narrowed by specifying a security group in "hz.aws.security.group"

> Back to table of contents [user_guide.html]

Chapter 4. Splout-Hadoop API

The Splout-Hadoop API contains the libraries and command-line tools needed for indexing and deploying "Tablespaces" to Splout. Splout uses Pangool [http://pangool.net] jobs for balancing and creating the needed binary files for being able to serve the provided datasets afterwards. Tablespaces can be generated according to some partitioning policy specified by the user. The partitioning is then leveraged by sampling processes to equitatively distribute data among partitions.

The output of these processes is usually a set of binary SQLite ".db" files and a partition map which specifies how queries should be routed to these files. Then, a "deployer" process is used to distribute these files to the Splout cluster. The "deployer" can also make the same file replicate several times in order to have fail-over replication.

4.1. Command line tools

The command line tools have been developed to ease the most common use cases. There are two "generator" tools that are responsible for launching a Hadoop process that will balance and index the input files, and there is one "deployer" tool that is able to deploy the result of any of the generators to an alive Splout cluster.

The tools allow to process either textual files (CSV or fixed-width), Cascading binary files, Hive tables and Pangool Tuple files.

4.1.1. Simple generator

The "Simple generator" allows us to seamlessly index and deploy a single tablespace made up by a single table, which is a very common use case. By invoking the tool with no parameters we obtain an explanation of all possible parameters. We will see a few examples of how to use this tool:

The following line generates the structures needed for deploying a tablespace called "customers" containing a table named "customers" whose schema is made up by an integer "customer_id" field, a "name" string and an integer "age". The file is present in input folder "my-input", will be partitioned in 12 partitions and the binary resultant files will be saved in "my-output". The partitioning policy is a columnar partitioning based on the column "customer_id".

```
hadoop jar splout-hadoop-*-hadoop.jar simple-generate -i my-input -o my-outpu
```

Tip

The default text format, when not specified, is a tabulated file with no quotes, no escaping, no header and no other active advanced parsing option.

The following line generates the structures for the same tablespace, but specifying a custom CSV format which is comma-separated, escaped by character "\", uses strict quotes (""), has a header line and may contain a sequence of characters which has to be interpreted as null: "\N".

```
hadoop jar splout-hadoop-*-hadoop.jar simple-generate --separator , --escape
```

Warning

Notice how we needed to escape backslashes when passing them through command-line parameters.

Strict quotes means that any field which is not quoted will be considered as null. When a field can't be parsed to its expected format, it is returned as null. For example, an empty integer field will be considered null.

Tip

Splout can also use fixed-width text files. For that, you can use the argument "--fixed-widthfields". When used, you must provide a comma-separated list of numbers. These

numbers will be interpreted by pairs, as [beginning, end] inclusive position offsets. For example: "0,3,5,7" means there are two fields, the first one of 4 characters at offsets [0, 3] and the second one of 3 characters at offsets [5, 7].

4.1.2. Generator

The "generator" is a simpler command-line which only accepts a JSON file. This JSON file will contain the specification of the tablespace or tablespaces to generate. In this case, tablespace specs can be as complex as desired, containing multiple tables if needed. You can also provide more than one JSON tablespace file to generate them together. Following we will show an example tablespace JSON file:

```
{
	"name": "meteo",
	"nPartitions": 16,
	"partitionedTables": [{
		"name": "meteo",
		"schema": "station:string,date:string,metric:string,measure:in
		"partitionFields": "station",
		"tableInputs": [{
			"inputSpecs": {
				"separatorChar": ","
			},
			"paths": [
				"small.csv"
			]
		}]
	}]
}
```

Following we will show the full schema of the JSON object (JSONTablespaceDefinition) that can be passed through this file:

Table 4.1. JSONTablespaceDefinition spec

Property	Type	Explanation
name	string	The name of the tablespace.
nPartitions	integer	The number of partitions to generate.
partitionedTables	array of JSONTableDefinition	The partitioned tables of this tablespace. There must be one, at least.
replicateAllTables	array of JSONTableDefinition	The tables that are replicated to all the partitions.

This is the spec of the JSONTableDefinition object:

Table 4.2. JSONTableDefinition object

Property	Type	Explanation
name	string	The name of the table.
tableInputs	array of JSONTableInputDefinition	The input locations of this table.
schema	string	The in-line Pangool schema that defines the structure of

		this table. This property is optional when using input types other than TEXT - the schema will be automatically discovered from input files.
partitionFields	array of string	If used, the table will be partitioned by one or more columns, otherwise it will be replicated to all partitions.
indexes	array of string	List of columns that need to be indexed after the data is added to the table. You can also specify compound indexes here, comma-separated.
initialStatements	array of string	Raw SQL commands that will be performed before the CREATE TABLE, but just after some defaults. Rigth place to put your PRAGMA statements.
preInsertStatements	array of string	Raw SQL commands that will be performed before inserting all the data to the table, just after the CREATE TABLE statements. For example, that is a good place to alter the table schema at your own.
postInsertStatements	array of string	Raw SQL commands that will be performed just after inserting all the data, but just before the CREATE INDEX statements
finalStatements	array of string	Raw SQL commands that will be performed just after all the other statements, at the end of the process.
insertionOrderBy	string	In-line Pangool Order By clause (in the form "field1:asc, field2:desc, …,fieldn:asc") that will be used for sorting the data before inserting it in a SQLite table.

This is the spec of the JSONTableInputDefinition object:

Table 4.3. JSONTableInputDefinition object

Property	Type	Explanation
paths	array of string	List of paths that will be used for creating this table.
inputType	InputType	Optional property. Type of input that will be added,

		by default, TEXT. Possible values are: TUPLE, CASCADING, HIVE.
inputSpecs	TextInputSpecs	Optional property. When using inputType = TEXT, specifies how to parse the text file.
cascadingColumns	string	Optional property. When using inputType = CASCADING, specify here a comma-separated list of column names. These names will be used when parsing the Cascading binary file.
hiveTableName	string	Optional property. When using inputType = HIVE, specify here the name of the Hive table to import.
hiveDbName	string	Optional property. When using inputType = HIVE, specify here the name of the Hive database from where the table will be imported.

And this is the spec of the TextInputSpecs object:

Table 4.4. TextInputSpecs object

Property	Type	Explanation
separatorChar	character	The field separator in the file. By default, a tabulation.
quotesChar	character	The quotes character, if any. By default, none.
escapeChar	character	The character used for escaping, if any. By default, none.
skipHeader	boolean	If the CSV has a header, activate this property for not failing to parse it.
strictQuotes	boolean	If quotesChar is specified, activating this property will cause all the fields without quotes to be considered null. False by default.
nullString	string	A sequence of characters that, if found without quotes, will be considered null. None by default.
fixedWidthFields	array of integers	If present, the file will be parsed as a fixed-width file. When used, you must provide a comma-separated list of numbers. These numbers will be interpreted by pairs,

		as [beginning, end] inclusive position offsets. For example: "0,3,5,7" means there are two fields, the first one of 4 characters at offsets [0, 3] and the second one of 3 characters at offsets [5, 7].

4.1.3. Deployer

The "deployer" tool can be used for deploying any tablespace or set of tablespaces that has been generated by any of the generators. More than one tablespace may be deployed at the same time, and Splout will increment the version for all of them in an "all-or-nothing" fashion. For the common case of deploying only one tablespace, you can use straight command-line parameters:

```
hadoop jar splout-hadoop-*-hadoop.jar deploy -r 2 -root my-generated-tablespa
```

The above line will deploy binary files generated in "my-generated-tablespace" folder using replication 2. The deployed tablespace will be named "mytablespace" and it will be deployed to the alive Splout cluster using the local QNode address at port 4412. The corresponding expected file tree for this example would have been the following:

```
hdfs://.../my-generated-tablespace/
hdfs://.../my-generated-tablespace/partition-map
hdfs://.../my-generated-tablespace/sampled-input
hdfs://.../my-generated-tablespace/store
```

(This file tree corresponds to the output of a "generator" process with "my-generated-tablespace" as output folder.)

Tip

For failover, it is convenient to replicate your tablespace when deploying it. If ommitted, only one copy of each binary file will be distributed to the cluster, meaning that if one machine fails there will be a portion of your data that will not be available for serving. A replication factor of 2 will mean that there will be 2 copies of each file, so one machine can fail and all the data will still be served. When deploying to a cluster with less machines than the replication factor specified, it will be automatically downgraded to the minimum viable one.

For deploying more than one tablespace atomically and with the same replication factor, you can also use command-line parameters:

```
hadoop jar splout-hadoop-*-hadoop.jar deploy -r 2 -root my-root-folder -ts my
```

In this case we will deploy 3 tablespaces at the same time: mytablespace1, mytablespace2 and mytablespace3. The "root" parameter is a parent folder that contains the specified subfolders, and the tablespaces will be named after the folder name. So in this case the file tree structure is the following:

```
hdfs://.../my-root-folder/mytablespace1/partition-map
hdfs://.../my-root-folder/mytablespace1/sampled-input
hdfs://.../my-root-folder/mytablespace1/store/...
hdfs://.../my-root-folder/mytablespace2/partition-map
hdfs://.../my-root-folder/mytablespace2/sampled-input
hdfs://.../my-root-folder/mytablespace2/store/...
hdfs://.../my-root-folder/mytablespace3/partition-map
hdfs://.../my-root-folder/mytablespace3/sampled-input
hdfs://.../my-root-folder/mytablespace3/store/...
```

Last but not least, if we are to atomically deploy a more complex combination of tablespaces, we can also use a JSON configuration file for that. This file will contain an array of "TablespaceDepSpec" objects whose spec is the following:

Table 4.5. TablespaceDepSpec object

Property	Type	Explanation
sourcePath	string	The root folder which contains one or more tablespaces and where this tablespace can be located.
tablespace	string	The subfolder in the root folder that contains the tablespace. It will be used for its name.
replication	integer	The replication factor to be used for this tablespace.

In this case, we can just pass the configuration file like shown below, taking into account that the file must be present in the local file system:

```
hadoop jar splout-hadoop-*-hadoop.jar -c my-config.json -q http://localhost:4
```

4.2. Hadoop Java API

All the command-line tools use the underlying Java API that we have implemented for Splout. You can also use this Java API directly in your Java project and you can have access to more advanced features such as specifying custom partitioning functions, record processing functions and such.

4.2.1. Configuring your environment for developing using the "splout-hadoop" API

You can use the splout-hadoop-starter [https://github.com/datasalt/splout-hadoop-starter] project as a starting point for your project that will use the splout-hadoop API. There are a few things you have to take into account:

- Splout uses SQLite native libraries: they can be found in splout-resources maven dependency, and you can use a maven plugin for uncompressing them like in the splout-hadoop-starter project's pom [https://github.com/datasalt/splout-hadoop-starter/blob/master/pom.xml]:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-remote-resources-plugin</artifactId>
    <version>1.4</version>
    <configuration>
      <resourceBundles>
        <resourceBundle>com.splout.db:splout-resources:0.2.4</resourceBundle
      </resourceBundles>
    </configuration>
   <executions>
     <execution>
       <goals>
         <goal>process</goal>
       </goals>
```

```
      </execution>
    </executions>
  </plugin>
```

- You then need to set your java.library.path in development mode (e.g. Eclipse) to contain the folder "target/maven-shared-archive-resources". In Eclipse, you can do that by:

  ```
  Run Configurations -> ... -> JRE -> Installed JREs... -> Click -> Edit ...
  ```

- When working in your own splout-hadoop project in pseudo-distributed or distributed Hadoop mode, you will need to copy the native libraries to the DistributedCache. The default Splout command-line tools automatically do that, but in your own project you must do this for ensuring that the libraries will be loaded in your Hadoop cluster. You can do that by using the splout-hadoop API before launching the Jobs as follows:

  ```
  // Only for distributed mode: Add sqlite native libs to DistributedCache
  if(!FileSystem.getLocal(hadoopConf).equals(fS)) {
    SploutHadoopConfiguration.addSQLite4JavaNativeLibsToDC(hadoopConf);
  }
  ```

The default local folder containing the native libraries is "native" but you can use any other. You will need to copy the libraries in Maven's "target/maven-shared-archive-resources" to e.g. "native" in your destination application distribution. You can do that with Maven's assembly. You can see an example of that in the splout-hadoop-starter project [https://github.com/datasalt/splout-hadoop-starter/blob/master/src/main/assembly/assembly.xml].

4.2.2. Basic API

The basic API consists of the following classes:

- TableBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TableBuilder.html]: a builder used to obtain a Table instance. Table instances can be used in TablespaceBuilder for constructing a tablespace specification.

- TablespaceBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TablespaceBuilder.html]: a builder used to obtain a TablespaceSpec instance.

- TablespaceGenerator [apidocs/splout-hadoop/com/splout/db/hadoop/TablespaceGenerator.html]: It can be used to generate the binary files according to a TablespaceSpec instance.

- StoreDeployerTool [apidocs/splout-hadoop/com/splout/db/hadoop/StoreDeployerTool.html]: It can be used to deploy the files generated by the TablespaceGenerator [apidocs/splout-hadoop/com/splout/db/hadoop/TablespaceGenerator.html] to an alive Splout cluster. It will accept TablespaceDepSpec instances, which have been documented in the previous sections.

The javadoc of each of these classes should guide you well into using them in your custom Java project. In addition, you can check this example [https://github.com/datasalt/splout-db/blob/master/splout-hadoop/src/main/java/com/splout/db/examples/PageCountsExample.java] which uses the Wikipedia Pagecounts data [http://dom.as/2007/12/10/wikipedia-page-counters/] for seeing a practical example on how to use this programmatic API.

4.2.2.1. Custom partitioning

Aside of column-based partitioning, an arbitrary partitioning function can be provided in the form of a Javascript function. This function ca be passed to TableBuilder's partitionByJavaScript() method.

4.2.2.2. RecordProcessor

If you want to have more control on the generation process, you can implement your own RecordProcessor which will receive Tuples as they have been parsed by the input files and should emit Tuples as you want them to be indexed in your SQL tables. For example, you may choose to narrow your input Tuple and emit a subset of it, modify some field by decoding its content, and so on. The RecordProcessor may also act as a filter. If "null" is returned, the input Tuple would have been filtered out from the generation process.

The pagecounts example RecordProcessor [https://github.com/datasalt/splout-db/blob/master/splout-hadoop/src/main/java/com/splout/db/examples/PageCountsRecordProcessor.java] is a good example.

> Back to table of contents [user_guide.html]

Chapter 5. Integration with other tools

It is possible to use the splout-hadoop API to import data directly from Hive, Cascading, Pig or Pangool. We will see an overview of such functionality in this section, together with some practical examples.

5.1. Integration with Hive

Note: For using Hive with Splout, it is recommended to add Hive conf/ and lib/ folder to the HADOOP_CLASSPATH environment variable:

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:$HIVE_HOME/conf:$HIVE_HOME/lib/*
```

In this way, Splout will be able to locate the appropriate Hive metastore and sample the Schema of Hive tables implicitly.

For importing tables using the "simple-generate" tool we can use arguments "-hdb" and "-htn" for indicating the Hive database and the Hive table we want to import. We will need to specify that the input type is "HIVE" through "-it" property and add the rest of parameters needed (number of partitions, partitioning columns, output folder, table name and tablespace name). Note how we don't need to specify input paths anymore as the input is read directly from Hive.

```
hadoop jar splout-hadoop-*-hadoop.jar simple-generate -it HIVE -hdb default -
```

Or we can use the "generate" tool. For that we can create a JSON tablespace descriptor like the example one. We specify the input type and the Hive database and table names. Note how we don't need to specify input paths.

```
{
        "name": "hive_simple_example",
        "nPartitions": 2,
        "partitionedTables": [{
                "name": "mentions_of_me",
                "partitionFields": "mentioned",
                "tableInputs": [{
                        "inputType": "HIVE",
                        "hiveTableName": "mentions",
                        "hiveDbName": "default"
                }]
        }]
}
```

We can then execute the JSON descriptor tablespace generation with the "generate" tool as usual:

```
hadoop jar splout-*-hadoop.jar generate -tf file:///`pwd`/hive_simple_example
```

And finally, with either methods, we can deploy the generated database as usual:

```
hadoop jar splout-hadoop-*-hadoop.jar deploy -root out-hive-simple -ts hive_s
```

When using the Java API, we can add Hive table inputs with methods addHiveTable() from TableBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TableBuilder.html].

Keep in mind that if you are using Hive with default embedded Derby database you can't import Hive tables while you have another session opened in Hive at the same time, as the import process will try to connect to it too.

Note: When using advanced Hive features such as the OCRInputFormat, it is also needed to add the hive-exec JAR via "-libjars", so that Mappers and Reducers can understand this native format:

```
hadoop jar splout-*-hadoop.jar generate -libjars $HIVE_HOME/lib/hive-exec-0.1
```

5.2. Integration with Cascading

Splout SQL can import binary Cascading files directly without needing to convert them to text. For that we just need to specify the input path where the output of the Cascading process is, and a list of comma-separated column names. The type of each column will be automatically discovered from the binary Cascading file.

For importing a Cascading table using the "simple-generate" tool we can use argument "-cc" for indicating the comma-separated column names, together with CASCADING in "-it" argument, and the rest of arguments as usual. See the following example:

```
hadoop jar splout-hadoop-*-hadoop.jar simple-generate -i out-clogs-analytics
```

Or we can use the "generate" tool. For that we can write a JSON descriptor as follows:

```
{
        "name": "cascading_simple_example",
        "nPartitions": 2,
        "partitionedTables": [{
                "name": "analytics",
                "partitionFields": "user",
                "tableInputs": [{
                        "inputType": "CASCADING",
                        "cascadingColumns": "day,month,year,user,category,cou
                        "paths": [ "out-clogs-analytics" ]
                }]
        }]
}
```

We can then execute the JSON descriptor tablespace generation with the "generate" tool as usual:

```
hadoop jar splout-*-hadoop.jar generate -tf file:///`pwd`/cascading_simple_ex
```

And finally, with either methods, we can then deploy the generated database as usual:

```
hadoop jar splout-hadoop-*-hadoop.jar deploy -root out-cascading-simple -ts c
```

When using the Java API, we can add Cascading table inputs with methods addCascadingTable() from TableBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TableBuilder.html]. Keep in mind that in order to have this functionality in your Java project you will need to explicitly import the appropriated Maven dependency:

```
<dependency>
  <groupId>cascading</groupId>
  <artifactId>cascading-hadoop</artifactId>
  <version>2.2.0-wip-15</version>
</dependency>
```

Which you can get from Conjar's Maven repo:

```
<repositories>
  <repository>
    <id>conjars.org</id>
```

```
  <url>http://conjars.org/repo</url>
</repository>
...
```

5.3. Integration with Pangool

Binary Pangool Tuple files can be imported directly using inputType "TUPLE". Both the types and the column names will be read from those files so we need to do nothing special besides indicating the inputType and the input path.

5.4. Integration with Pig

We can import data from Pig by storing it as Pangool Tuple files. For that we need to obtain the pangool-core JAR. We can download it from Maven Central [http://search.maven.org/#search%7Cga%7C1%7Cpangool] if we don’t have it somewhere else. We then register the JAR in our Pig sessiona s follows:

```
REGISTER /.../path-to-pangool-core-jar/.../pangool-core-0.60.2.jar;
```

And we can save any output as:

```
STORE cntd INTO 'pig-wordcount-result' USING com.datasalt.pangool.pig.Pangool
```

Note how the first argument to the StoreFunc is the table name, and the rest are column names. The following example executes a simple word-count task in Pig and saves the output as a binary Pangool Tuple file. We will use an input file called *mary* with the following content:

```
Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go.
```

This is the script code:

```
REGISTER /.../path-to-pangool-core-jar/.../pangool-core-0.60.2.jar;
a = LOAD 'mary' as (line);
words = FOREACH a GENERATE flatten(TOKENIZE(line)) AS word;
grpd = GROUP words BY word;
cntd  = FOREACH grpd GENERATE group, COUNT(words);
STORE cntd INTO 'pig-wordcount-result' USING com.datasalt.pangool.pig.Pangool
```

We can then import the resulting binary file as a normal Pangool Tuple file, for example with the "simple-generate" tool:

```
hadoop jar splout-hadoop-*-hadoop.jar simple-generate -i pig-wordcount-result
hadoop jar splout-hadoop-*-hadoop.jar deploy -root out-pig-simple -ts pig_sim
```

Chapter 6. REST API

You can interact with the server through the Java client [apidocs/splout-javaclient/com/splout/db/common/SploutClient.html] or directly through the REST interface. These are the basic methods of the REST interface:

Table 6.1. Rest API overview

Method type	Path	Parameters	Explanation	Example
GET	api/overview	(none)	Returns a QNodeStatus object with the overview of the cluster.	http://localhost:4412/api/overview
GET	api/dnodelist	(none)	Returns the list of DNodes in the cluster.	http://localhost:4412/api/dnodelist
GET	api/tablespaces	(none)	Returns the list of active tablespaces in the cluster.	http://localhost:4412/api/tablespaces
GET	api/tablespace/{tablespace}	(none)	Returns a Tablespace object with the info for this particular tablespace.	http://localhost:4412/api/tablespace/mytablespace
GET	api/tablespace/{tablespace}/versions	(none)	Returns a Map<Long, Tablespace> object with all the available versions for this tablespace.	http://localhost:4412/api/tablespace/mytablespace/versions
GET	api/dnode/{dnode}/status	(none)	Returns a DNodeSystemStatus object for the specified dnode.	http://localhost:4412/api/dnode/localhost:4422/status
GET	api/query/{tablespace}	key, sql, [callback]	Performs a SQL query to the specified tablespace and returns a QueryStatus object.	http://localhost:4412/api/mytablespace?key=K1&sql=SELECT%201;
POST	api/deploy	List<DeployRequest>	Performs a deploy according to the associated DeployRequest objects passed as body for the request and returns a DeployInfo object.	

POST	api/rollback	List<SwitchVersionRequest>	Requests a rollback according to the associated DeployRequest objects passed as body for the request and returns a StatusMessage object.	

Following we will see each method with a little more detail:

6.1. api/overview

Use this method for obtaining an overview on the cluster status. Returns a QNodeStatus status object. Example response:

```
{
	dNodes: {
		192.168.1.3:4422: {
			average: "NaN",
			files: [ ],
			freeSpaceInDisk: 289065160,
			upSince: 1354124108706,
			nQueries: 0,
			failedQueries: 0,
			slowQueries: 0,
			systemStatus: "UP",
			deployInProgress: false,
			occupiedSpaceInDisk: 0,
			lastExceptionTime: -1
		}
	},
	tablespaceMap: { },
	clusterSize: 2
}
```

Table 6.2. QNodeStatus object

Property	Type	Explanation
dNodes	Map<String, DNodeSystemStatus>	Alive DNodes in cluster and their associated information.
tablespaceMap	Map<String, Tablespace>	Current tablespaces being served by the cluster and their associated information.
clusterSize	integer	Number of services (QNodes + DNodes) in the cluster.

Table 6.3. DNodeSystemStatus object

Property	Type	Explanation
systemStatus	string	"UP" if everything is fine. Otherwise "Last exception" together with a short Exception message will appear indicating that some Java Ex-

		ception was thrown by the DNode.
lastExceptionTime	long	The time when the last Java Exception was thrown corresponding to the sytemStatus caption.
deployInProgress	boolean	Whether the DNode is fetching data for a deployment or not.
upSince	long	The time when this DNode was started.
nQueries	int	The number of queries that this DNode has served.
failedQueries	int	The number of queries that this DNode has failed to serve.
slowQueries	double	The number of queries considered to be "slow". Slow queries are configured by configuration property "dnode.slow.query.abs.limit".
average	double	The average query time for this DNode.
occupiedSpaceInDisk	long	The number of bytes occupied by the data that this DNode holds.
freeSpaceInDisk	long	The free disk space in the disk that this DNode is using.
files	List<String>	The list of files for this DNode, and the size of each file in parenthesis.

Table 6.4. Tablespace object

Property	Type	Explanation
partitionMap	PartitionMap	The partition map that is being used to route queries for this tablespace.
replicationMap	ReplicationMap	The replication map that is being used for failover for this tablespace.
version	long	The version number of this tablespace.
creationDate	long	The time when this tablespace was deployed.

Table 6.5. PartitionMap object

Property	Type	Explanation
partitionEntries	List<PartitionEntry>	The list of partition entries with (min, max) ranges for this tablespace.

Table 6.6. PartitionEntry object

Property	Type	Explanation
min	string	If a key falls between [min, max) it will be routed to this shard. Min is inclusive.
max	string	If a key falls between [min, max) it will be routed to this shard. Max is not inclusive.
shard	int	If a key falls between [min, max) it will be routed to this shard.

Table 6.7. ReplicationMap object

Property	Type	Explanation
replicationEntries	List<ReplicationEntry>	The list of replication entries with list of dnodes for this tablespace.

Table 6.8. ReplicationEntry object

Property	Type	Explanation
shard	int	This shard can be served by any dnode in the "nodes" list.
nodes	list<string>	This shard can be served by any dnode in the "nodes" list.

6.2. api/dnodelist

Use this method for getting the list of alive DNodes. Returns a list of strings. Example response:

```
[
	"192.168.1.3:4422",
	"192.168.1.4:4422",
]
```

6.3. api/tablespaces

Use this method for getting the list of tablespaces being served in the cluster. Returns a list of strings. Example reponse:

```
[
	"pagecounts"
]
```

6.4. api/tablespace/{tablespace}

Use this method for getting the associated information of a tablespace being served by the cluster. Returns a Tablespace object. Example response:

```
{
	partitionMap: {
		partitionEntries: [
			{
```

```
                                shard: 0,
                                max: "Sp",
                                min: null
                        },
                        {
                                shard: 1,
                                max: null,
                                min: "Sp"
                        }
                ]
        },
        version: 5649092059,
        replicationMap: {
                replicationEntries: [
                        {
                                shard: 0,
                                nodes: [
                                        "192.168.1.3:4422"
                                ]
                        },
                        {
                                shard: 1,
                                nodes: [
                                        "192.168.1.3:4422"
                                ]
                        }
                ]
        },
        creationDate: 1354124763853
}
```

6.5. api/tablespace/{tablespace}/versions

Returns all available versions for the specified tablespace, including the one which may be being served at the moment by the cluster. Returns a Map<Long, Table 6.4, “Tablespace object” [23]>. For every version in the key of the map, returns the associated Tablespace object information. Example response:

```
{
        5649092059:
                partitionMap: {
                        partitionEntries: [
                                {
                                        shard: 0,
                                        max: "Sp",
                                        min: null
                                },
                                {
                                        shard: 1,
                                        max: null,
                                        min: "Sp"
                                }
                        ]
                },
                version: 5649092059,
                replicationMap: {
                        replicationEntries: [
```

```
                                {
                                        shard: 0,
                                        nodes: [
                                                "192.168.1.3:4422"
                                        ]
                                },
                                {
                                        shard: 1,
                                        nodes: [
                                                "192.168.1.3:4422"
                                        ]
                                }
                        ]
                },
                creationDate: 1354124763853
        }
}
```

6.6. api/dnode/{dnode}/status

Returns a DNodeSystemStatus object filled with the detailed information of the specified DNode. Example response:

```
{
        average: "NaN",
        files: [
                "/var/opt/splout/./dnode-staging (13512 bytes)",
                "/var/opt/splout/./dnode-staging/pagecounts (13512 bytes)",
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059 (13512
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/0 (3072
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/0.meta
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/0/0.db
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/1 (1024
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/1.meta
                "/var/opt/splout/./dnode-staging/pagecounts/5649092059/1/1.db
        ],
        freeSpaceInDisk: 289059948,
        upSince: 1354124108706,
        nQueries: 0,
        failedQueries: 0,
        slowQueries: 0,
        systemStatus: "UP",
        deployInProgress: false,
        occupiedSpaceInDisk: 13512,
        lastExceptionTime: -1
}
```

6.7. api/query/{tablespace}

Perform a SQL query and get a JSON result back. Returns a QueryStatus object with some metadata about the query execution. Example response for key=Up, sql=SELECT * FROM pagecounts LIMIT 1 and tablespace pagecounts (http://localhost:4412/api/query/pagecounts?key=Up&sql=SELECT%20*%20FROM%20pagecounts%20LIMIT%201;):

```
{
        "millis": 135,
        "error":null,
```

```
        "result": [
                {
                        "pagename": "zh:####",
                        "hour": "23",
                        "pageviews": 1,
                        "date": "20090430"
                }
        ],
        "shard": 1
}
```

Table 6.9. QueryStatus object

Property	Type	Explanation
result	list<Object>	The query result from the database, which has been JSONified. If there was an error, it will be empty.
shard	int	The shard that the query was routed to.
millis	long	The time the query took to be executed.
error	string	The error message if there was any.

6.8. api/deploy

Warning

You won't usually need to perform a deploy manually using this REST method. However, you can refer to this documentation in case you need to do something very specific. For doing deploys you will usually use the "deployer" command-line tool or the StoreDeployerTool [apidocs/splout-hadoop/com/splout/db/hadoop/StoreDeployerTool.html] Java class.

By providing a list of DeployRequest objects we can perform a deploy through this POST method. We need to specify a tablespace name, an URI from where binary objects must be fetched, a PartitionMap and a ReplicationMap and optionally a list of SQL statements that will be executed each time a connection is made to the database.

Example post body:

```
[
        {
                "tablespace": "pagecounts",
                "data_uri": "file:/opt/splout-db/splout-hadoop/out-pagecounts/
                "partitionMap": [
                        {
                                "min": null,
                                "max": "Sp",
                                "shard":0
                        },
                        {
                                "min": "Sp",
                                "max": null,
                                "shard":1
                        }
```

```
                ],
                "replicationMap": [
                        {
                                "shard": 0,
                                "nodes": [
                                        "192.168.1.3:4422"
                                ]
                        },
                        {
                                "shard": 1,
                                "nodes": ["192.168.1.3:4422"]
                        }
                ],
                "initStatements": [
                        "pragma case_sensitive_like=true;"
                ]
        }
]
```

Table 6.10. DeployRequest object

Property	Type	Explanation
tablespace	string	The tablespace name. It will be used as identifier after being deployed if it doesn't exist. If it exists, a new version will be promoted for it.
data_uri	string	The absolute URI where the binary files can be found.
initStatements	list<string>	A list of SQL statements that will be executed everytime a connection is made to the database files. Useful for using custom SQLite PRAGMAs.
partitionMap	list<Table 6.6, "PartitionEntry object" [24]>	The Partition map to be used for routing queries from this tablespace.
replicationMap	list<Table 6.8, "ReplicationEntry object" [24]>	The Replication map to be used for failover for this tablespace.

Tip

Java users can use SploutClient [apidocs/splout-javaclient/com/splout/db/common/SploutClient.html] or higher-level StoreDeployerTool [apidocs/splout-hadoop/com/splout/db/hadoop/StoreDeployerTool.html] (which uses apidocs/splout-javaclient/com/splout/db/common/SploutClient.html[SploutClient] underneath) instead of the raw REST API for doing deploys.

Tip

Partition maps are automatically generated by sampling methods in the generator tools and saved in the output folder that was used for the tool. The tools use the TablespaceGenerator [apidocs/splout-hadoop/com/splout/db/hadoop/TablespaceGenerator.html] Java class underneath.

Tip

Typical replication maps can be built easily with the Java API methods ReplicationMap.roundRobinMap() [apidocs/splout-commons/com/splout/db/common/ReplicationMap.html] and ReplicationMap.oneToOneMap() [apidocs/splout-commons/com/splout/db/common/ReplicationMap.html].

The deploy is an asynchronous operation coordinated by the cluster members. A deploy request returns inmediately with a Table 6.11, “DeployInfo object” [29] object indicating whether it could be started or whether there was some error trying to start it:

Table 6.11. DeployInfo object

Property	Type	Explanation
error	string	If there is any error starting the deploy.
string	startedAt	A timestamp indicating the time when this deploy started.
version	long	The version number that the tablespace will have when the deploy is promoted.

6.9. api/rollback

By providing a list of SwitchVersionRequest objects we can perform a rollback through this POST method. We just need to specify the tablespace name (which must be being served by the cluster at the moment of the rollback) and the version we want to set it to.

Table 6.12. SwitchVersionRequest object

Property	Type	Explanation
tablespace	string	The tablespace name, used as identifier.
version	long	The version to rollback to. It must be available in the cluster.

Tip

Java users can use SploutClient [apidocs/splout-javaclient/com/splout/db/common/SploutClient.html] instead of the raw REST API for doing rollbacks.

Rollback is a synchronous operation and the return type is Table 6.13, “StatusMessage object” [29]:

Table 6.13. StatusMessage object

Property	Type	Explanation
status	string	"Done" if it could be done, otherwise an error will be printed here.

> Back to table of contents [user_guide.html]

Chapter 7. Tips / Troubleshooting

In this section we will take a look to the common problems or tips that one has to take into account when using Splout:

7.1. Query speed

There are a wide variety of reasons why queries may or may not perform well. It is important to understand that each query is executed in a SQLite connection of a binary SQLite file. When troubleshooting query performance, it is advisable to take a look to the SQLite documentation [http://www.sqlite.org/] and in addition check things like:

- PRAGMA index_list(table) - to check if we have created the appropriated indexes or not.
- EXPLAIN QUERY PLAN (query) - to show if SQLite is using the expected indexes or not.
- PRAGMA case_sensitive_like - when doing LIKE queries, SQLite will use an index only if this pragma is set to true and the index was created using standard collation, or if this pragma is set to false and the index was created using COLLATE NOCASE.
- ANALYZE - You can run ANALYZE as finalStatement on partitions generation. It could help when planning query execution.

Remember that you can fine-tune your data indexing process with initialStatements, preInsertStatements, postInsertStatements and finalStatements (see Table 4.2, “JSONTableDefinition object” [11]) and initStatements (see Table 6.10, “DeployRequest object” [28]). All these features are available as part of the standard Java API of TableBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TableBuilder.html] and TablespaceBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TablespaceBuilder.html]. Use them for adding custom PRAGMA or custom CREATE INDEX commands if needed.

It is also very important to keep in mind that colocating data in disk is crucial for query speed. For example, if your query impacts 1000 records and these records need to be loaded from the main table, if they are not colocated in disk and the database doesn’t fit in memory, it would mean that the server has to peform 1000 potential seeks. You can control the data colocation policy with "insertionOrderBy" as explained in Table 4.2, “JSONTableDefinition object” [11], which is also available in TableBuilder [apidocs/splout-hadoop/com/splout/db/hadoop/TableBuilder.html].

As an example, we have used data colocation techniques within Splout SQL to obtain < 50ms average query time with 10 threads on dynamic GROUP BY’s that hit an average of 2000 records each in a multi-gigabyte database that exceeded available RAM in orders of magnitude in a m1.small EC2 machine.

7.2. Deploys failing or taking too long

Because deployments are asynchronous operations, it can be tricky to know if they have succeeded or not. You can monitor a deployment using the information returned by the API (deployInProgress flags).

A deploy may fail if some of the DNodes fail, if a timeout is reached (see Configuration) or if the leader QNode dies. The default timeout is 10 hours, so it shouldn’t be hit under normal circumstances.

7.3. The cluster is inconsistent: there are less DNodes than expected

In big networks in can take some time for Hazelcast to negotiate the membership. It can be a matter of minutes.

7.4. DNode fails all queries with "Too many open files" exceptions

You might have to tune your open files limit as Splout is opening one connection to each partition file it has for every serving thread. So if you have 64 serving threads and 20 partitions to serve, this means up to 1280 opened files which is more than the default in some machines (1024). You can check and change your limits: http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files/