# flamy Documentation

*Release 0.7.0*

Furcy Pin

Jan 05, 2018

# Contents

**Flamy** is an all-in-one command line tool that helps you :

- Manage your Hive (and Spark-SQL) database on Hadoop or AWS
- Keep your SQL tables and queries clean and well-defined
- Easily design, control and execute complex data workflows
- Automatically infer and visualize dependencies between SQL queries

# CHAPTER 1

# Getting Started

## 1.1 Flamy Overview

### 1.1.1 What Flamy is

Flamy is a command line tool designed to make all people using SQL on Hadoop being more productive.

It does so by bringing multiple functionalities, that allows SQL developers to: - easily and rapidly check the integrity of their queries, even against an evolving database - better visualize and understand the workflows they created - easily deploy and execute them on multiple environments - efficiently gather metadata from the Metastore

SQL is often recognized to be a powerful language to script and automate queries, while at the same time maintaining and improving running workflow can often become quite frustrating. The fact that it is not a compiled language and cannot be easily unit tested is often cited as a main downside when compared to other approaches such as using plain Spark in java or scala.

Flamy's philosophy is to remove such downsides, and allow users to make the most out of SQL on Hadoop, without forcing them to use SQL for tasks it is not great at.

### 1.1.2 What Flamy is not

#### Flamy is not a GUI

Flamy doesn't come with fancy graphical sugar, yet. Sorry.

#### Flamy is not a scheduler

It can execute workflows of consecutive Hive-SQL queries, and we plan to add more capabilities such as running Presto queries or Spark jobs, but it has no cron-like capabilities and is not intended to have someday.

Flamy is best used in conjunction with your favorite scheduler, either by using flamy to generate a workflow and export it into the scheduler's format, or by having the scheduler directly calling flamy commands. We encourage the community to contribute by building such bridges between flamy and other great open source schedulers.

## 1.2 Setup Guide

### 1.2.1 Installing Flamy

To install flamy, you can either download a pre-packaged version or build it from source.

#### Dependencies

Flamy requires the program dot to be able to print table dependency graphs.

#### Debian-based

```
apt-get install graphviz libgraphviz-dev
```

#### Mac OS X

Install brew if not already installed

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)" < /dev/null 2> /dev/null
```

Install graphviz

```
brew install graphviz
```

#### Installation

#### Download a pre-packaged version

Download and untar the .tgz from this url:

```
wget 'https://oss.sonatype.org/service/local/artifact/maven/content?r=snapshots&g=com.
↪flaminem&a=flamy&p=tgz&v=LATEST' | gunzip | tar -x
```

*You still need to install the program 'dot' as explained above to be able to display graphs.*

#### or Build from source

Compilation requires sbt to compile.

```
git clone git@github.com:flaminem/flamy.git
cd flamy
sbt clean stage
```

The packaging directory will be found at *target/universal/stage*, with the executable at *target/universal/stage/bin/flamy* and the configuration file at *target/universal/stage/conf/flamy.properties* but bear in mind that recompiling the project will regenerate the *target/universal/stage/* folder. You can use the *–config-file* to point the configuration file to an alternate location.

#### Starting a shell

Once packaged, you can start a shell with:

```
target/universal/stage/bin/flamy shell
```

Once in the shell, the *help* command will list all the available commands and their options, and the *show conf* command will help you troubleshoot any configuration issue.

#### (Optional) Running unit tests

```
sbt test
```

#### (Optional) Granting access for Flamy to the Hive Metastore

Flamy can perform some actions such as metadata retrieval from Hive's Metastore. It can perform many useful actions such as listing all the schemas, tables, or partitions with useful associated information.

For this Flamy can use either the Thrift client (HiveMetastoreClient) provided by Hive or directly connect to the metastore database via JDBC. While the first method works out of the box, it is often much slower than the second. On the other hand, the JDBC connection requires some configuration on the Hive Metastore's side, *as explained here*.

#### (Optional) local install

#### Ubuntu

In your .basrhc, add this and set FLAMY_HOME to the correct value:

```
FLAMY_HOME=<PATH_TO_FLAMY_INSTALL_DIR>
alias flamy=$FLAMY_HOME/bin/flamy
```

#### Next steps

- Try the *demo*
- Configure flamy by editing `target/universal/stage/conf/flamy.properties` (See the *configuration guide*)
- Check out *the list of all commands available in Flamy*.

### 1.2.2 Setting up direct Metastore access

Flamy can perform some actions such as metadata retrieval from Hive's Metastore. It can perform many useful actions such as listing all the schemas, tables, or partitions with useful associated information. For this, Flamy can use either the Thrift client (HiveMetastoreClient) provided by Hive or directly connect to the metastore database via JDBC. While the first method works out of the box, it is often much slower than the second.

In order to grant direct access to the Hive Metastore for Flamy, you need to ask your favorite administrator to create a flamy user and grant it a **read-only** access to the following tables of the metastore (beware, for names are case-sensitive) :

- PARTITIONS
- TBLS
- DBS
- SDS
- COLUMNS_V2
- TABLE_PARAMS
- PARTITION_PARAMS

Currently, flamy is only compatible with Metastore's backed by PostgreSQL, MySQL or MariaDB. MySQL and MariaDB require the user to manually download the jdbc client jar, for license incompatibility reasons.

If you are not yourself an administrator, and want to perform the changes yourself on a development cluster, you may do so by following theses steps:

### 1. connect with ssh to the machine hosting the database's metastore

(replace <MY_HOSTS> with the correct name)

```
ssh <METASTORE_HOST>
```

### 2. connect to the database

If you have a Cloudera cluster with a **postgresql** database, you may retrieve the admin password with:

```
sudo cat /var/lib/cloudera-scm-server-db/data/generated_password.txt
```

and then connect with:

```
psql -h localhost -p 7432 hive cloudera-scm
```

Otherwise please refer to your SQL back end's documentation.

### 3. Once connected to the database, create a new user called flamy

(replace <FlamyPassword> with the strong password of your choice):

```
CREATE USER flamy WITH PASSWORD '<FlamyPassword>';
```

### 4. And grant read-only access to flamy to the following tables

```
GRANT SELECT ON TABLE "PARTITIONS", "TBLS", "DBS", "SDS", "COLUMNS_V2", "TABLE_PARAMS
↪", "PARTITION_PARAMS" TO flamy ;
```

### 5. Change Flamy's configuration to enable direct access to the Metastore

Edit conf/flamy.properties and add or change the line

```
flamy.env.<ENVIRONMENT>.hive.meta.fetcher.type = direct
```

for the specific environment you want to connect to (eg: dev, prod, etc.)

You also need to fill the following properties:

```
flamy.env.<ENVIRONMENT>.hive.metastore.jdbc.uri = ...
flamy.env.<ENVIRONMENT>.hive.metastore.jdbc.user = flamy
flamy.env.<ENVIRONMENT>.hive.metastore.jdbc.password = ...
```

For instance, if the metastore is using **postgresql**, the jdbc uri shall be of this form

**::** jdbc:postgresql://<METASTORE_HOST>:<PORT>/hive

If using Cloudera, <PORT> might be 7432, but we recommend checking on your parameters.

### 6. Try connecting to the database

by running the following command:

```
bin/flamy show schemas --on <ENVIRONMENT>
```

#### 7. (Optional) If the metastore is using *postgresql*, you might get this kind of error

```
FATAL: no pg_hba.conf entry for host "XXX.XXX.XXX.XXX", user "flamy", database "hive",
↪ SSL off
"hive", SSL off
```

In such case, you will have to:

#### 7.1. connect to the Metastore Host with a sudo user

```
ssh <METASTORE_HOST>
```

#### 7.2. edit postgresql's pg_hba.conf configuration file

If using cloudera and the hive metastore uses the same database as Cloudera Manager:

```
sudo vi /var/lib/cloudera-scm-server-db/data/pg_hba.conf
```

otherwise:

```
sudo vi /etc/postgresql/<PG_VERSION>/main/pg_hba.conf
```

#### 7.3. Add the following line

```
# Grant access for user flamy to hive database
host    hive        flamy        <IP/MASK>          md5
```

where <IP/MASK> describe the subnetwork that will require to connect to the database with flamy. Check https://www.iplocation.net/subnet-mask for more informations.

#### 7.4. Finally, restart the database (schedule a maintenance for this)

Be careful if the database is using **postgresql**, and especially if the same database is used by cloudera, since the *stop* (and the *restart*) command will block every new connection to the database and wait for all currently open connections to be closed before stopping. This means that you will have to stop the HiveMetastore, HiveServer2, and all Cloudera monitoring services first (the ActivityMonitor, cloudera-scm-server and all cloudera-scm-agents).

If you don't fear being careless, the *fast_stop* command should shut down the database immediately and drop the currently open connections.

## 1.3 Tutorial

In this tutorial, we will use a sample of server logs from the NASA (1995), and show how we can use Hive, Spark and Flamy to build a small ETL pipeline.

### 1.3.1 Part 0. Starting the demo

First, download and install flamy *as explained here*, and make sure `FLAMY_HOME` is correctly set.

```
export FLAMY_HOME=<path/to/flamy/installation/dir>
```

Once ready, checkout the flamy_demo git repository:

```
git clone git@github.com:flaminem/flamy_demo.git
cd flamy_demo
```

start the demo:

```
./demo.sh
```

and type your first flamy command:

```
show tables
```

You should obtain the following result:

Once you are ready, we will start running some offline commands (no Hive cluster required).

### 1.3.2 Part 1. Local commands

#### Folder architecture

First, let us have a look at what this repository contains: We see a `conf/` folder containing the configuration file for flamy, a `data/` folder that contains sample data, and most importantly a `model/` folder that contains your first Hive project using flamy.

If you take a look at the conf, you see that flamy is configured via the `conf/flamy.properties` file, in which the configuration line `flamy.model.dir.paths = model` indicates to flamy where the model folder is located.

```
$ tree conf
conf
├── flamy.properties
└── log4j2.properties
```

As you can see, the `model/` follows the architecture pattern used by Hive to store data on hdfs: `model/schema_name.db/table_name/...`

```
$ tree model
model
├── model_PRESETS.hql
├── nasa
│   ├── facts.db
│   │   └── http_status
│   │       └── CREATE.hql
│   ├── nasa_access.db
│   │   ├── daily_logs
│   │   │   ├── CREATE.hql
│   │   │   └── POPULATE.hql
│   │   ├── daily_url_error_rates
│   │   │   ├── CREATE.hql
│   │   │   └── POPULATE.hql
│   │   ├── daily_urls
│   │   │   ├── CREATE.hql
│   │   │   └── POPULATE.hql
│   │   └── daily_urls_with_error
│   │       ├── CREATE.hql
│   │       └── POPULATE.hql
│   └── nasa_access_import.db
│       ├── daily_logs
│       │   ├── CREATE.hql
│       │   └── POPULATE.hql
│       └── raw_data
│           └── CREATE.hql
└── VARIABLES.properties
```

Each folder contains one or two .hql files called `CREATE.hql` and `POPULATE.hql`. If you open them, you will see that the CREATEs contain the definition of each table (CREATE TABLE . . . ) and the POPULATEs contain INSERT statements to populate the tables with data.

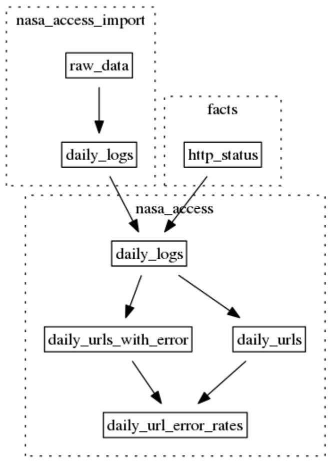
As you can see, we have already written all the Hive queries that need, and we will demonstrate how flamy can help us leverage that code efficiently and easily.

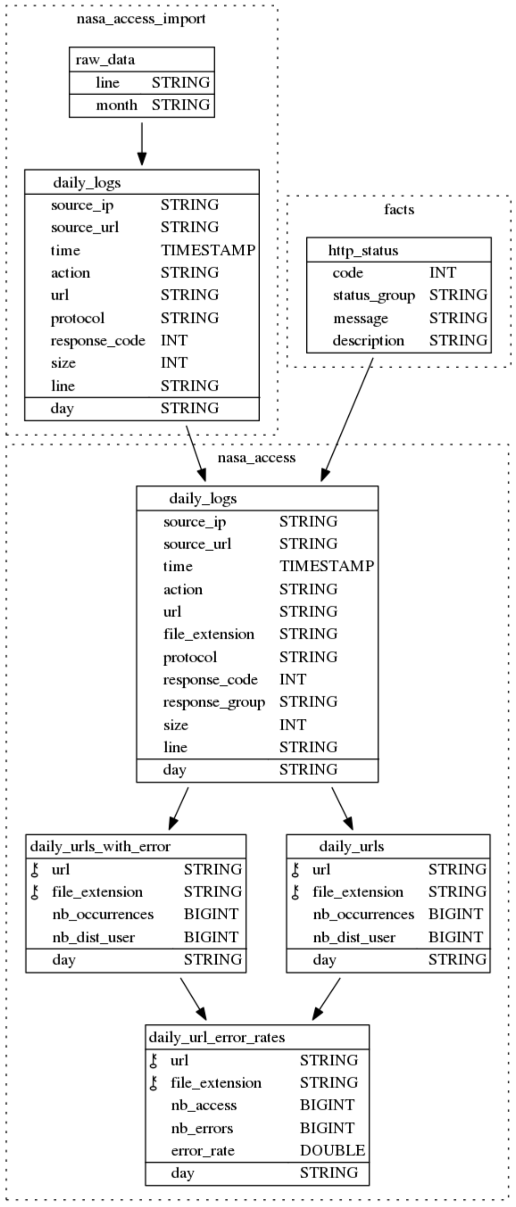
It's now time to try flamy's first feature: `show graph` !

### show graph

```
flamy> show graph
INFO: Files analyzed: 18    Failures: 0
graphs printed at :
   file:///tmp/flamy-user/result/graph%20.png
   file:///tmp/flamy-user/result/graph%20_light.png
```

Normally, a new window should automatically open and show you these images (You should be able to see the second one by pressing the right arrow):





If not, you can also try right-clicking or ctrl-clicking on the url (`file:///.../graph%20.png`) displayed by the shell to open the file.

This is one of flamy's main feature: flamy just parsed the whole `model/` folder, found and parsed all the `CREATE.hql` and `POPULATE.hql` files, and build the dependency graph of the tables. Simply put, we have an arrow going from table A to table B if we insert data coming from table A into table B. This is quite different from ORM design diagrams, where arrows symbolize foreign keys relationship, which do not exist in Hive.

As you can see, flamy always proposes two graphs, one 'light graph' to see the relationship between the tables, and another heavier graph that also displays the columns and partitions of each table. The big dotted boxes each correspond to a schema.

Of course, for big projects, displaying a whole graph with hundred of tables is not practical, which is why the `show graph` command take schema or table names as argument, to be able to concentrate on single tables. For instance, let us type:

```
flamy> show graph nasa_access.daily_logs
INFO: Files analyzed: 18    Failures: 0
graphs printed at :
   file:///tmp/flamy-user/result/graph%20nasa_access.png
   file:///tmp/flamy-user/result/graph%20nasa_access_light.png
```

This opens a new window with two new graphs, where only the tables inside the schema nasa_access are displayed, and where the dependencies that do not belong to nasa_access are still summarized in blue.

This `show graph` feature gives you a whole new way to navigate through your Hive table and better understand the architecture you or your fellow colleagues have created. This is very useful at every stage of the workflow's life: from early design and code review, to troubleshooting and production.

This command has a few options that are very useful to learn to master, you can get the list by typing `help show graph` or `show graph --help`. The most worth to be mentioned are:

- `-o --out` allows you to see the downstream dependencies of a table, very useful when editing or removing a table to see what will be impacted
- `-I --in-defs` to see the complete definitions of the upstream dependencies
- `-v --skip-views` to see the tables behind the views

We will now see the next useful feature: `check`

### check

When designing or updating a Hive workflow, making sure everything works can be complicated. Sometimes, even renaming a column is prohibited as it would cause too much hassle. However, keeping a data pipeline clean without ever renaming anything is like keeping a java project without ever refactoring: near to impossible.

The first reflex to acquire once you start using flamy is to always use the check command. In a way, it is like compiling your SQL code.

Let us try it:

```
check quick
```

As you can see, we voluntarily left a typo in one of the POPULATE.hql files. This one is easy to solve, it is simply a comma that should not be there before the FROM. You can open the file by right-clicking (MacOS) or ctrl-clicking (Linux) of the file where the typo is, and fix it.

Once the typo is fixed, rerun the `check quick` command to validate that all the queries are ok. The `check quick` command is able to detect several kind of errors, including

- Syntax errors
- Wrong number of columns
- Unknown column references
- Table name not corresponding to the file path

The validation was made to be as close as possible as Hive's, but there might be some discrepancies, for instance, flamy will complain if you insert two columns with the same alias into a table while Hive will not.

Now that the quick check is valid, let us try the long check:

```
check long
```

As you can see, the long check takes longer and found another error. Basically, what the long check does is performing a quick check first, then creating all the tables in an empty build-in environment, and finally performing an EXPLAIN statement on every POPULATE.hql with Hive to ensure that they are all compiled by Hive.

The error we found this time is a Semantic error: we forgot to add the column file_extension to the GROUP BY statement. Once you have fixed this error, run `check long` again to make sure all your Hive queries are correct.

This feature is extremely useful, and it is quite easy to use with a Continuous Integration server to make sure that whenever someone commits a change to the code base, nothing is broken.

Once all our queries are valid, there is only one thing left to do: run them!

### run

Let us run the following command:

```
run --dry --from nasa_access.daily_logs --to nasa_access.daily_url_error_rates
```

Since we haven't configured flamy for accessing a remote Hive database, the only thing we can do for now is performing a local dry-run. The dry-run mode is activated with the `--dry` option, and allows you to check what the command you type is going to do without actually running it. Any query-compile-time error will be spotted, which means that if the dry-run works, the only type of error you might encounter when running the command for real will be runtime errors.

The next arguments are a `--from` and a `--to` which allows you to ask flamy to run every POPULATE.hql related to the tables that are located in the graph between the `--from` tables and the `--to` tables.

You can also simply write the name of all the tables you want to run, or the name of a schema if you want to run the queries for all the tables in that schema. For instance, if you run the command `run --dry nasa_access` you will see it has exactly the same behavior.

Flamy runs the query by following the dependency ordering of the tables, and runs them in parallel whenever possible (even if it too fast to see in this example). The maximal number of queries run by flamy simultaneously is set by `flamy.exec.parallelism` which equals 5 by default.

Finally, as you might have noticed, next to each successful run is a `{DAY -> "1995-08-08"}`, this inform you that in the corresponding POPULATE, the variable `${DAY}` has been substituted with the value `"1995-08-08"`. This value is set inside the file `model/VARIABLES.properties`, which is itself given to flamy via the `flamy.variables.path` configuration parameter.

You can try changing this value either by editing the `VARIABLES.properties` file, or simply with the `--variables` option, as in the following command:

```
run --dry --from nasa_access.daily_logs --to nasa_access.daily_url_error_rates
```

Beware that this option should be given before the main `run` command. Every variable used in a POPULATE should have a default value in the VARIABLES file, to allow flamy performing checks.

Now that you successfully ran your first dry-run, it is time to reach the interesting part: running commands against a remote cluster!

### Summary

We have seen how flamy helps us visualizing the dependency graph of the Hive queries that we wrote, and validating all our queries like a compiler would do. Thanks to this, refactoring a Hive workflow, by renaming a table or a column for instance, has never been so easy. Finally, we saw how flamy is able to execute the query workflow in dry-run mode: this will help us in the next part of this tutorial, where we will run queries against a real (sandbox) Hive environment.

### 1.3.3 Part 2. Remote commands

#### Installing a sandbox Hive environment

**With docker**

If you have docker installed, simply run this command from inside this project's directory (this might require sudo rights):

```
docker run --rm -e USER=`id -u -n` -e USER_ID=`id -u` -it -v `pwd`/data:/data/hive -p␣
↪127.0.0.1:9083:9083 -p 127.0.0.1:4040:4040 -p 127.0.0.1:10000:10000 fpin/docker-
↪hive-spark
```

It will start a docker that will automatically:

- expose a running Metastore on the port 9083
- expose a running Spark ThriftServer (similar to HiveServer2) on port 10000
- expose the Spark GUI to follow job progress on port 4040
- start a beeline terminal

In case the docker won't start correctly, please make sure the specified ports are not already used by a service on your machine.

**Without docker**

If you don't have docker, you can either install it, or if you can't (e.g. you don't have sudo rights on your workstation) or don't want to install docker, you can try installing and configuring Hive and Spark by following the instruction from the Dockerfile in this repository, and running the file `start.sh`.

You might also need to update your `model/VARIABLES.properties` file to have `EXTERNAL_DATA_LOCATION` point to the absolute location of the `warehouse/` folder.

#### diff/push

We now have a remote environment, which we will be able to query using the `--on` option:

```
show tables --on local
```

As you can see, this new environment is empty for now.

We can use the `push` command to create all the schemas on the local environment:

```
show schemas --on local
push schemas --on local
```

`push` can be used to create schemas and tables on the remote environment. It will only create schemas and tables that do not already exists. You can also use the `diff` commands to see the difference between your model and the remote environment. Of course, you can specify the name of the schemas or tables that you want to interact with.

Let us push the tables on the local environment:

```
diff tables --on local
push tables --on local nasa_access_import facts.http_status
diff tables --on local
push tables --on local
diff tables --on local
```

As you can see, the first `push` command uses arguments and will only push the table `facts.http_status` and all the tables in `nasa_access_import`. The second `push`, without argument, will push all the remaining tables.

### describe/repair

Now that the tables are created, we can fetch some information about them:

```
describe tables --on local
```

As you can see, the two input tables `facts.http_status` and `nasa_access_import.raw_data` are TEXTFILE, but the latter doesn't seem to contain any data yet.

If you are familiar with Hive, you know that this is because the partition metadata have to be created first. We can do this with the `repair` command, that will run the command `MSCK REPAIR TABLE` on the required table. We can also use the command `describe partitions` to check that the partitions are now created:

```
describe partitions --on local nasa_access_import.raw_data
repair tables --on local nasa_access_import.raw_data
describe partitions --on local nasa_access_import.raw_data
```

As you can see, once the `repair tables` command has run, two new partitions have been added to the table `nasa_access_import.raw_data`

### run (again!)

Let us now run our first real Hive queries with flamy:

```
run --on local nasa_access_import.daily_logs
```

This query shall take a few minutes, while this run we will take a quick look at our pipeline.

The goal here was to create a simple data pipeline, taking 2 months of gzip raw web server logs, and turning them into SQL tables where we could monitor the error rates per url and per day for this server.

Let us look at the graph again:

As you can see, the first table `nasa_access_import.raw_data` is partitioned by month to ingest the two gzipped raw files, while the rest of the tables are partitioned by day. For this reason the POPULATE action for `nasa_access_import.daily_logs` take a while because we must parse the whole 2 months of data. The next queries that we will run will apply to one day at a time, so they won't take as long.

Once the previous query is done, we can now run our queries for the rest of the pipeline, and check that the new partitions have been created:

```
run --on local nasa_access
describe partitions --on local nasa_access
--variables "DAY='1995-08-09'" run --on local nasa_access
describe partitions --on local nasa_access
```

Once we have done that, in a real use case, the only thing to do left would be to schedule a job to run the command `flamy --variables "DAY='$(date +%F)'" run nasa_access` every day (sort of).

The `--dry` option is really useful for testing everything quickly when deploying a new workflow.

Here at Flaminem, we use our own home-made python scheduler, for which we developed a few plugins to have a good integration with flamy, and we encourage the community to try using it with their own schedulers.

Thanks to Flamy, we managed to develop, deploy and maintain tens of Hive+Spark workflows, running in production hundreds of Hive queries every day, with a very small team of 3 developers dedicated to it.

We also developed a command called `regen`, that allows flamy to determine by itself when some partitions are outdated compared to their upstream dependencies. You can learn more about it here.

### 1.3.4 What next ?

Congratulations, you have completed this tutorial and are now free to try flamy for yourself. From here, you can :

- Try writing new queries on this example, run and validate them with flamy
- Try to plug flamy to your own Hive cluster (see the *Configuration page*)
- Go to the *FAQ + Did You Know?* section to learn a few more tricks
- Check out *Flamy's regen* section to learn about Flamy's most amazing feature
- Spread the word and star flamy's repo :-)

# CHAPTER 2

# User Guide

## 2.1 Naming Conventions

Flamy uses the following terms and conventions:

### 2.1.1 Schema

A **schema** represents a set of tables.

It is uniquely identified by its name (eg: `my_schema`)

### 2.1.2 Table

A **table** represents a Hive table.

Although Hive does not, Flamy requires users to always refer to a table with its fully qualified name (eg: `my_schema.my_table`)

### 2.1.3 Item

An **item** may represent either a Schema or a Table. When a command requires ITEMS as arguments, the user can specify any space-separated list of table and/or schema names. Giving a schema name is equivalent to giving each table names inside this schema.

### 2.1.4 Partition

A **partition** represents one partition of a Hive Table. If you are not familiar with partitioning in Hive, checkout this tutorial and try to use them, partitions are great! Just don't try to have too many for one table… As a rule of thumb, tables with more than a few thousands partitions may start causing issues.

A partition is identified by a string of the form: `schema.table/part1=val1[/part2=val2...]` (eg: `stats.daily_visitors/day=2014-10-12/campaign=shoes`) However, the ordering of the columns in the string do not matter for Flamy (even if it does for Hive and HDFS). (eg: `stats.daily_visitors/campaign=shoes/day=2014-10-12` works too)

### 2.1.5 Special Characters

The characters `.`, `=`, and `/` being used as delimiters in the partitions names, they should not be used as schema name, table name, partition key or value.

## 2.2 Configuring Flamy

Flamy looks for its configuration file at `$FLAMY_HOME/conf/flamy.properties`. This location can be overridden when starting `flamy` with the `-config-file` option.

To get started configuring flamy, we recommend to rename the file `$FLAMY_HOME/conf/flamy.properties.template` to `$FLAMY_HOME/conf/flamy.properties` and start editing it.

Despite the file extension, flamy's configuration uses the HOCON syntax, which is a superset of both the JSON syntax and the Java properties syntax (almost). It means you can write the properties either like a regular java.properties file, or choose any level of nesting for your parameters.

For troubleshooting, you can check your configuration with the command `flamy config [--on ENV]` This will display all the values that are active in your configuration for the specified environment.

### 2.2.1 Global properties

`flamy.model.dir.paths` List[String] *Space-separated list of folder paths where flamy will look for the SQL files of your model.*

`flamy.variables.path` Option[String] *Path to the file where the variables are defined.*

`flamy.udf.classpath` Option[String] *List of jar paths (separated with ':') where flamy will look for the custom Hive UDFs. Don't forget to also add them as CREATE TEMPORARY FUNCTION in the model's presets file.*

`flamy.exec.parallelism` Int (default: 5) *Controls the maximum number of jobs that flamy is allowed to run simultaneously.*

### 2.2.2 Environment properties

These properties can be set for each environment you want to configure. Just replace `<ENV>` by the name of the correct environment

`flamy.env.<ENV>.hive.server.uri` String *URI of the Hive Server 2.*

`flamy.env.<ENV>.hive.server.login` String (default: "user") *Login used to connect to the Hive Server 2.*

`flamy.env.<ENV>.hive.presets.path` Option[String] *Path to the .hql presets file for this environment. These presets will be executed before every query run against this environment.*

`flamy.env.<ENV>.hive.meta.fetcher.type` "direct" | "client" | "default" (default: "default") *The implementation used to retrieve metadata from Hive ('client' or 'direct').*

`flamy.env.<ENV>.hive.metastore.uri` String *Thrift URI of the Hive Metastore. Required in client mode of the meta.fetcher.*

`flamy.env.<ENV>.hive.metastore.jdbc.uri` String *JDBC URI of the Hive Metastore database. Required in direct mode of the meta.fetcher.*

`flamy.env.<ENV>.hive.metastore.jdbc.user` String (default: "flamy") *JDBC user to use when connecting to the Hive Metastore database. Required in direct mode of the meta.fetcher.*

`flamy.env.<ENV>.hive.metastore.jdbc.password` String (default: "flamyPassword") *JDBC password to use when connecting to the Hive Metastore database. Required in direct mode of the meta.fetcher.*

### 2.2.3 Other properties

These are additional, less used, properties.

`flamy.run.dir.path` String (default: "/tmp/flamy-user") *Set the directory in which all the temporary outputs will be written. By default this is a temporary directory created in /tmp/flamy-$USER.*

`flamy.run.dir.cleaning.delay` Int (default: 24) *Set the number of hours for which all the run directories older than this time laps will be automatically removed. Automatic removal occurs during each flamy command startup.*

`flamy.regen.use.legacy` Boolean (default: false) *Use the old version of the regen.*

`flamy.io.dynamic.output` Boolean (default: true) *The run and regen commands will use a dynamic output, instead of a static output. Only work with terminals supporting ANSI escape codes.*

`flamy.io.use.hyperlinks` Boolean (default: true) *Every file path that flamy prints will be formatted as a url. In some shells, this allows CTRL+clicking the link to open the file.*

`flamy.auto.open.command` String (default: "xdg-open" on Linux, "open" on Mac OSX) *Some commands like 'show graph' generate a file and automatically open it. Use this option to specify the command to use when opening the file,or set it to an empty string to disable the automatic opening of the files.*

`flamy.auto.open.multi` Boolean (default: false) *In addition with auto.open.command, this boolean flag indicates if multiple files should be open simultaneously.*

`flamy.verbosity.level` "DEBUG" | "INFO" | "WARN" | "ERROR" | "SILENT" (default: "INFO") *Controls the verbosity level of flamy.*

## 2.3 List of Flamy commands

**Most flamy commands have the following form:**

```
flamy [GLOBAL_OPTIONS] COMMAND [SUB_COMMAND] [COMMAND_OPTIONS] [ITEMS]
```

If you are already inside a `flamy shell`, beginning the command with `flamy` is not necessary.

All the commands available in the shell can also be run as standalone commands. For instance, `flamy help` and `flamy show conf` both work. This enables the user to easily write and execute scripts that calls sequence of flamy commands. The flamy shell handle quotes the same way as bash does, so there should be no worry when copy-pasting commands from the shell to standalone commands.

**The arguments `ITEMS` denotes a list of tables or schemas. Tables should always be referred to using their fully-qualified names (eg `my_database.my_table` ). Giving a schema name is equivalent to giving the list of all tables in that schema.**

### 2.3.1 Global options

When specified before the `shell` command, the options will apply to all commands subsequently run inside the shell. The following options are available:

`--help` *Display the help.*

`--version` *Show version information about the software.*

`--config-file PATH` *Make flamy use another configuration file than the default.*

`--conf KEY1=VALUE1 [KEY2=VALUE2 ...]` *Specify configuration parameters from the command line.*

`--variables NAME1=VALUE1 [NAME2=VALUE2 ...]` *Declare variables in the command line. These declaration will override the ones made in the variable declaration file. Be careful when using this option, as bash automatically removes quotes. For instance if you want to declare a variable* `DAY` *whose value is* `"2015-01-03"` *(quotes included), you should write* `--variables DAY='"2015-01-03"'`.

### 2.3.2 shell

`flamy shell` *Starts an interactive shell with autocomplete and faster response time. Once in the shell, commands are run in the same way, without the first* `flamy` *keyword. The shell handles quotes the same way bash does, which means commands run in flamy's shell can be copy-pasted, prefixed by the path of the flamy executable and directly run in a bash script.*

### 2.3.3 help

`flamy -h`

`flamy help [COMMAND]`

### 2.3.4 show

`flamy show conf [--on ENV]` *List all configuration properties applicable to the specified environment.*

`flamy show schemas [--on ENV]` *List all schemas in the specified environment.*

`flamy show tables [--on ENV] [SCHEMA1 SCHEMA2 ...]` *List all tables inside one or several schemas.*

`flamy show partitions --on ENV TABLE` *List all partitions in a table.*

`flamy show graph [--from FROM_ITEMS --to TO_ITEMS | ITEMS]` *Print the dependency graph of the specified items.*

`flamy show select` *Print a SELECT statement for the given table.*

### 2.3.5 describe

*Similar to show commands, but display more information, and take more time to run. We recommend to enable direct metastore access* as explained here.

`flamy describe schemas --on ENV` *List all schemas with their properties (size, last modification time).*

`flamy describe tables --on ENV [ITEMS]` *List all tables inside one or several schemas with their properties (size, last modification time).*

`flamy describe partitions [--bytes] --on ENV TABLE` *List all partitions in a table with their properties (size, last modification time).*

### 2.3.6 diff

*Helpful to compare the differences between environments.*

`flamy diff schemas --on ENV` *Show the schemas differences between the specified environment and the modeling environment.*

`flamy diff tables --on ENV [ITEMS]` *Show the table differences between the specified environment and the modeling environment.*

`flamy diff columns --on ENV [ITEMS]` *Show the column differences between the specified environment and the modeling environment.*

### 2.3.7 push commands

*Helpful to propagate changes on from your model to another environment.*

`flamy push schemas --on ENV [--dry] [SCHEMA1 SCHEMA2 ...]` *Create on the specified environment the schemas that are present in the model and missing in the environment.*

`flamy push tables --on ENV [--dry] [ITEMS]` *Create on the specified environment the tables that are present in the model and missing in the environment.*

### 2.3.8 check

`flamy check quick ITEMS` *Perform a quick check on the specified items.*

`flamy check long ITEMS` *Perform a long check on the specified items. This take more time than the quick check, but is more thorough.*

`flamy check partitions ITEMS --on ENV [--from FROM_ITEMS --to TO_ITEMS | ITEMS]` *Check the partitions dependencies on the specified items. This command will be much faster if run from the cluster rather than from remote.*

### 2.3.9 run

`flamy run [--dry] [--on ENV] [--from FROM_ITEMS --to TO_ITEMS | ITEMS]` *Execute the POPULATE workflow on the specified environment for the specified items. If –dry flag is used, the queries will be checked by Hive but not run. If no environment is specified, run in standalone mode on empty tables (this requires to have *‘‘SET hadoop.bin.path=… ‘‘* in your local presets.*

### 2.3.10 Other commands

`flamy wait-for-partitions --on ENV [OPTIONS] PARTITIONS` *Wait for the specified partitions to be created if they don’t already exist. Options are:* `--after TIME` *will wait for the partitions to be created or refreshed after the specified TIMESTAMP* `--timeout DURATION` *will make flamy return a failure after DURATION seconds* `--retry-interval INTERVAL` *will make flamy wait for INTERVAL seconds between every check*

`flamy gather-info --on ENV [ITEMS]` *Gather all partitioning information on specified items (everything if no argument is given) and output this as csv on stdout.*

## 2.4 FAQ + Did You Know?

In this section are gathered a few 'Frequently Asked Questions' and 'Did You Know' tricks that are useful to learn when using flamy.

Don't hesitate to come back to this page often to refresh you memory or learn new things.

### 2.4.1 Frequently Asked Questions

Be the first to ask a question!

https://groups.google.com/forum/#!forum/flamy

*(But before you do, please make sure this section doesn't already answer it)*

### 2.4.2 Did you know?

#### Folder architecture

When flamy scans the model folder, it looks recursively for folders ending in `.db` This means that you can regroup your schemas in subfolders if you want. The only constraint are that the folders corresponding to the tables must be directly inside the schema (`.db`) folder. The table folder may then contain `CREATE.hql`, `POPULATE.hql`, `VIEW.hql` and `META.properties` files. You can safely add other type of files in theses directories, they will be ignored by flamy, but we plan to extend the set of files recognized by flamy in the future. For instance, this folder structure is allowed:

```
model
├── schema0.db
│   └── table0
│       ├── CREATE.hql
│       ├── comments.txt
│       └── work_in_progress.hql
├── project_A
│   ├── schemaA1.db
│   │   └── tableA1a
│   │       └── CREATE.hql
└── project_B
    └── schemaB1.db
        └── tableA21
            └── CREATE.hql
```

The configuration `flamy.model.dir.paths` allows you to specify multiple folders, if you want to separate your projects even more.

#### Schema properties

If you want to create a schema with specific properties (location, comment), you can add a `CREATE_SCHEMA.hql` inside the schema (.db) folder, which will contain the CREATE statement of your schema. Flamy will safely ignore the location when dry-running locally.

### What about views?

Flamy supports views. To create a view with flamy, all you have to do is to write a `VIEW.hql` statement with the CREATE VIEW statement instead of the `CREATE.hql`.

Views are treated as table when possible, which means that the `show tables`, `describe tables` command will correctly list them, and the `push tables` command will correctly push them, in the right order.

### Multiple POPULATEs on the same table?

Flamy allows you to write multiple queries separated by semicolons `;` in the same `POPULATE.hql`, in such case, the queries will always be run together and sequentially. But you can also have multiple POPULATE files, by using a suffix of the form `_suffix`. In such case, when possible flamy will execute all the POPULATE files of a given table in parallel.

For instance a common pattern in Hive for a table aggregating data from two sources, is to partition it by source and to have one Hive query per source. In such case you could write a `POPULATE_sourceA.hql` and a `POPULATE_sourceB.hql` file to keep the two logics separated and be able to execute both queries in parallel.

### Hidden files

Files and folder prefixed with a dot `.` or an underscore _ will be ignore by flamy. This follows the same convention as HDFS, and is especially useful when you are developing something that is not fully ready yet, but you still want flamy to validate everything.

### Presets files

For any environment `myEnv` you have configured (including the 'model' environment), you can set the configuration parameter `flamy.env.<ENV>.hive.presets.path` to make it point to a `.hql` file that may contains several commands that will be performed before every query session on this environment. For instance, if your cluster prevents dynamic partitioning by default, you can add this line in your presets file to enable it for all your queries.

```
SET hive.exec.dynamic.partition.mode = nonstrict ;
```

This file is also required to handle custom UDFs, as explained in the next paragraph.

### Custom UDFs

One of Hive's main advantages is that it is quite easy to create and use custom UDFs. If you have custom UDFs, when using the `check long` or the `run --dry` command locally, you have to make sure that flamy has access to the custom UDF jar and that the functions are correctly defined in the model presets.

This is how to proceed:

- Set the `flamy.udf.classpath` configuration parameter to point to the jar(s) containing your custom UDFs.
- Create a PRESETS_model.hql file and set `flamy.env.model.hive.presets.path` to point to it.
- In the presets file, add one line to create each function you want to use `CREATE TEMPORARY FUNCTION my_function AS "com.example.hive.udf.GenericUDFMyFunction" ;`

### What about non-Hive (e.g. Spark) jobs?

We all agree that SQL is great at performing some tasks, and very poor at others, which is why our most complex jobs in our workflow are done with pure-Scala Spark jobs. To handle these Spark dependencies between two tables, add a file called `META.properties` in the destination table folder and indicate the name of the source tables of your spark job like this:

```
dependencies = schema.source_table_1, schema.source_table_2
```

When displaying the dependency graph with `show graph`, flamy will now add blue arrows in the graph to represent these external dependencies.

Unfortunately, for now, flamy is not capable of handling Spark job, and we usually used a regular scheduler to populate all the tables required by the spark job with one `flamy run` command, then started the spark job, and finally populated all the tables downstream with another `flamy run` command.

Better handling for Spark jobs is part of the new features we would like to develop, although we know that since Spark is much more permissive than the SQL syntax, some features, like the automatic dependency discovery or the dry-run will be difficult to extend to Spark.

For jobs at the interface between the Hive cluster and other services, we used our regular scheduler, and flamy was no help here. However some of its feature like the graph and the dry-run could be a source of inspiration for designing similar features in a scheduler.

## 2.5 Regen

In this section, we present Flamy's most powerful feature: the `regen` This feature is not part of the open-source edition of Flamy, so please contact us if you would like to try it out.

After two years developing and using flamy, we obtained very high increase in productivity when using Hive, but one of the most time-consuming issue we still had to deal with was making sure that our constantly-evolving data pipelines were always correct and up to date.

Imagine that you start with a simple data pipeline such as the one we used in the *tutorial*:

### 2.5.1 Motivation

Let us take as an example the simple data pipeline used in [flamy's tutorial](Demo). The input table of your workflow (here nasa_access.daily_logs) receives data from an external source and you execute various transformations to obtain other tables. Your tables are partitioned by day, and every day, you execute the workflow with the `flamy run` command for the new batch of data that arrived.

But at some point, you realize that because of some error somewhere, the last two months of data that arrived in the table nasa_access.daily_logs contains some error that should be fixed (for instance, a field has a wrong format, and this generates null values down your pipeline). What should you do?

Generally, the solution consists in writing a custom query to fix the issue over the last two months, and of course update the table's Populate to prevent this problem from occurring again.

But once you have applied your fix and regenerated two months of data in the table nasa_access.daily_logs, you should perhaps propagate the changes downstream and recompute two months of data for the other tables below. And this is where we hit a complex issue: how to check which partitions are up-to-date with their upstream, and which one should be regenerated? In our simple example we have only 4 tables, but quite frequently companies can build very complex pipelines that can have tens of intermediary steps, branches, and so on. Imagine the kind of conundrums it can create!

This is why we added to flamy the `regen` feature:

### 2.5.2 regen

By analyzing the Hive queries, flamy is actually able to infer which partitions are outdated or missing regarding their upstream data. Flamy is then capable of running the right POPULATE queries, with the correct variable replacements, to (re-)generated the outdated or missing partitions in your workflow.

Here is an example of the regen running on the tutorial's example:

With flamy's regen, you can now rest assured that your data pipeline is entirely coherent.

This is also extremely useful for failure recovery: if you run a workflow with flamy's regen, and one of the query fails for some reason, then by relaunching the command, flamy will automatically skip the queries that were successfully completed the first time.

**All this, without having to write any workflow description: just your SQL queries and one flamy command.**

This feature is not part of the open-source edition of Flamy, please contact us if you are interested and want to know more.