verdictdoc Documentation

Release 0.4.1

Verdict Team

Oct 05, 2017

Contents:

CHAPTER 1

Quick Start

Verdict can run on top of Apache Hive, Apache Impala, and Apache Spark, and Amazon Redshift. We are adding drivers for other database systems, such as Google BigQuery, Google Dataproc, Teradata, etc.

Using Verdict is easy. Following this guide, you can finish setup in five minutes if you have any of those supported systems ready.

Downloading and Building Verdict

Downloading and building Verdict requires only a couple steps. Building Verdict does not require any `sudo` access.

1. **Download** and unzip the latest release (version 0.3.0).
1. **Type** `mvn package` in the unzipped directory. The command will download all the dependencies and compile Verdict's code. The command will create three `jar` files in the `target` directory.

This is all!

More details

Verdict is tested on Oracle JDK 1.7 or above, but it should work with open JDK, too. `mvn` is the command for the Apache Maven package manager. If you do not have the Maven, you will have to install it. The official page for the Maven installation is this.

Verdict is currently tested with the systems included in a Cloudera distribution (CDH 5.11). However, Verdict should work with other versions as well. Please let us know if Verdict is incompatible with your environment.

Using Verdict

Verdict takes a slightly different approach depending on the database system it works with. Once connected, however, you can issue the same SQL queries.

On Apache Spark

Verdict works with Spark by creating Spark's HiveContext internally. In this way, Verdict can load persisted tables through Hive Metastore. Verdict is tested with Apache Spark 1.6.0 (in the Cloudera distribution CDH 5.11). We will support Spark 2.0 shortly.

We show how to use Verdict in `spark-shell` and `pyspark`. Using Verdict in an Spark application written either in Scala or Python is the same.

Due to the seamless integration of Verdict on top of Spark (and PySpark), Verdict can be used within Apache Zeppelin notebooks and Python Jupyter notebooks. See this page for more detail about how to set up Verdict with Zeppelin or Jupyter.

Verdict-on-Spark

You can start `spark-shell` with Verdict as follows:

```
$ spark-shell --jars target/verdict-core-0.3.0-jar-with-dependencies.jar
```

After spark-shell starts, import and use Verdict as follows:

```
import edu.umich.verdict.VerdictSparkHiveContext

scala> val vc = new VerdictSparkHiveContext(sc)    // sc: SparkContext instance

scala> vc.sql("show databases").show(false)        // Simply displays the databases␣
↪(or often called schemas)

// Creates samples for the table. This step needs to be done only once for the table.
// The created tables are automatically persisted through HiveContext and can be used␣
↪in the other
// pyspark applications.
scala> vc.sql("create sample of database_name.table_name").show(false)

// Now Verdict automatically uses available samples for speeding up this query.
scala> vc.sql("select count(*) from database_name.table_name").show(false)
```

The return value of `VerdictSparkHiveContext#sql()` is a Spark's DataFrame class; thus, any methods that work on Spark's DataFrame work on Verdict's answer seamlessly.

Verdict-on-PySpark

You can start `pyspark` shell with Verdict as follows:

```
$ export PYTHONPATH=$(pwd)/python:$PYTHONPATH

$ pyspark --driver-class-path target/verdict-core-0.3.0-jar-with-dependencies.jar
```

Limitation: Note that, in order for the `--driver-class-path` option to work, the jar file (i.e., `target/verdict-core-0.3.0-jar-with-dependencies.jar`) must be placed in the Spark's driver node. Verdict will support `--jars` option shortly.

After pyspark shell starts, import and use Verdict as follows:

```
>>> from pyverdict import VerdictHiveContext
```

```
>>> vc = VerdictHiveContext(sc)         # sc: SparkContext instance

>>> vc.sql("show databases").show()     # Simply displays the databases (or often␣
↪called schemas)

# Creates samples for the table. This step needs to be done only once for the table.
# The created tables are automatically persisted through HiveContext and can be used␣
↪in the other
# pyspark applications.
>>> vc.sql("create sample of database_name.table_name").show()

# Now Verdict automatically uses available samples for speeding up this query.
>>> vc.sql("select count(*) from database_name.table_name").show()
```

The return value of `VerdictHiveContext#sql()` is a pyspark's DataFrame class; thus, any methods that work on pyspark's DataFrame work on Verdict's answer seamlessly.

On Apache Impala, Apache Hive, Amazon Redshift

We will use our command line interface (which is called `veeline`) for connecting to those databases. You can also programmatically connect to Verdict (see *JDBC in Java/Python applications*).

Prerequsites for *veeline*

`veeline` uses the JDBC drivers stored in the `libs` folder for making a connection to the database Verdict works with. Therefore, to make `veeline` able to connect to your database, you must store the `jar` files required to make a connection to your database. By default, our code ships with the Cloudera's Impala and Hive JDBC drivers, and Redshift JDBC drivers. However, if these drivers are not compatible with your environment, you should put the compatible JDBC drivers in the *libs* folder in place of existing ones. We plan to automate this process in the future.

Verdict-on-Impala

Type the following command in terminal to launch `veeline` that connects to Impala:

```
$ veeline/bin/veeline -h "impala://hostname:port/schema;key1=value1;key2=value2;..." -
↪u username -p password
```

Note that parameters are delimited using semicolons (`;`). The connection string is quoted since the semicolons have special meaning in bash. The user name and password can be passed in the connection string as parameters, too.

Verdict supports the Kerberos connection. For this, add `principal=user/host@domain` as one of those key-values pairs.

After `veeline` launches, you can issue regular SQL queries as follows:

```
verdict:impala> show databases;

// Creates samples for the table. This step needs to be done only once for the table.
verdict:impala> create sample of database_name.table_name;

verdict:impala> select count(*) from database_name.table_name;

verdict:impala> !quit
```

Verdict-on-Hive

Type the following command in terminal to launch `veeline` that connects to Hive:

```
$ veeline/bin/veeline -h "hive2://hostname:port/schema;key1=value1;key2=value2;..." -
↪u username -p password
```

Note that parameters are delimited using semicolons (`;`). The connection string is quoted since the semicolons have special meaning in bash. The user name and password can be passed in the connection string as parameters, too.

Verdict supports the Kerberos connection. For this, add `principal=user/host@domain` as one of those key-values pairs.

After `veeline` launches, you can issue regular SQL queries as follows:

```
verdict:Apache Hive> show databases;

// Creates samples for the table. This step needs to be done only once for the table.
verdict:Apache Hive> create sample of database_name.table_name;

verdict:Apache Hive> select count(*) from database_name.table_name;

verdict:Apache Hive> !quit
```

Verdict-on-Redshift

Type the following command in terminal to launch `veeline` that connects to Amazon Redshift:

```
$ veeline/bin/veeline -h "redshift://endpoint:port/schema;key1=value1;key2=value2;...
↪" -u username -p password
```

Note that parameters are delimited using semicolons (`;`). The connection string is quoted since the semicolons have special meaning in bash. The user name and password can be passed in the connection string as parameters, too.

After `veeline` launches, you can issue regular SQL queries as follows:

```
// In Redshift, this displays the schemas in the database to which you are connected
verdict:PostgreSQL> show databases;

// Creates samples for the table. This step needs to be done only once for the table.
verdict:PostgreSQL> create sample of schema_name.table_name;

verdict:PostgreSQL> select count(*) from schema_name.table_name;

verdict:PostgreSQL> !quit
```

The search path can be set by `use schema_name;` statement. Currently, only a single schema name can be set for the search path using the `use` statement.

Notes on using `veeline`

`veeline` makes a JDBC connection to the database systems that Verdict work on top of (e.g., Impala or Hive). For this, it uses the JDBC drivers stored in the `lib` folder. Our code ships by default with the Cloudera's Impala and Hive JDBC drivers (jar files). However, if these drivers are not compatible with your environment, you can put the compatible JDBC drivers in the `lib` folder after deleting existing ones.

What's Next

See what types of queries are supported by Verdict in this page, and enjoy the speedup provided Verdict for those queries.

If you have use cases that are not supported by Verdict, please contact us at `verdict-user@umich.edu`, or create an issue in our Github repository. We will answer your questions or requests shortly (at most in a few days).

CHAPTER 2

Using Verdict

In terminal

Please see our *Quick Start*. for the instructions on connecting to Verdict in terminal. The page includes starting Verdict on top of Apache Hive, Apache Impala, Amazon Redshift, and Apache Spark (and PySpark) in terminal.

JDBC in Java/Python applications

Verdict ships with a JDBC driver. Using the driver, you can use Verdict on top of your existing JDBC-supported database systems. Currently, Verdict supports the JDBC connections to Apache Hive, Apache Impala, and Amazon Redshift. Contact us if you want to use Verdict on other database systems. We need to add a small driver (due to non-standard SQL features used by different databases).

1. Class name for Verdict's JDBC driver: `edu.umich.verdict.jdbc.Driver`

1. Connection string for Verdict's JDBC driver:

    - Hive: `jdbc:verdict:hive2://host:port/default_database;key1=value1;key2=value2;...`
    - Impala: `jdbc:verdict:impala://host:port/default_database;key1=value1;key2=value2;...`
    - Redshift: `jdbc:verdict:redshift://endpoint:port/database;key1=value1;key2=value2;...`

To enable the Kerberos authentication, add `principal=user/host@domain` pair in the key-value pairs of the JDBC connection string. You can also pass Verdict configuration options in key-value pairs. For example, to change Verdict's log level to `DEBUG`, add `verdict.loglevel=debug` in the key-value pairs. See *Configurations* for more configuration options for Verdict.

Details

The rule for Verdict's connection string is to include the `verdict:` after `jdbc:`. Internally, Verdict communicates with the target database (e.g., Hive, Impala, etc.) using a JDBC connection as well. For this, Verdict uses the passed

JDBC connection string after removing (1) the `verdict:` keyword and (2) configuration options for Verdict. This means you can pass any existing JDBC options to Verdict as they are.

Java example

Below example connects to Verdict-on-Impala and runs a count query:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class VerdictJdbcExample {
  public static void main(String[] args) throws ClassNotFoundException, SQLException {
    Class.forName("edu.umich.verdict.jdbc.Driver");
    String url = "jdbc:verdict:impala://host:port/default";
    Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
    String sql2 = "select count(*) from orders";
    ResultSet rs = stmt.executeQuery(sql2);
  }
}
```

Of course, the Verdict's JDBC driver (`target/verdict-core-version-with-dependencies.jar`) must be included in the Java class path when the above code is compiled and run.

Python example

Below example connects to Verdict-on-Impala and runs a simple count query:

```
import jaydebeapi
import jpype
import pandas as pd
from jpype import *

class_name = 'edu.umich.verdict.jdbc.Driver'
url = 'jdbc:verdict:impala://host:port/default'
jvm_path = jpype.getDefaultJVMPath()
jpype.startJVM(jvm_path, "-Djava.class.path=%s" % classpath)
conn = jaydebeapi.connect(jclassname = class_name, url = url)
curs = conn.cursor()
curs.execute('select count(*) from orders')
columns = [desc[0] for desc in curs.description]       # getting column headers
pd.DataFrame(curs.fetchall(), columns = columns)
```

Note that the Python libraries needed for creating a JVM instance (`jpype`) and making a JDBC connection (`jaydebeapi`) must be installed.

In Apache Zeppelin

Apache Zeppelin is a notebook-based application that runs in a web browser. It can connect to Apache Spark by default and also to other database systems through the JDBC interface. Verdict can also easily integrate with Apache Zeppelin.

On Spark

Zeppelin includes the Spark interpreter by default. For Verdict to work in Spark interpreter, it is enough to include Verdict's core jar file as a dependency. First, go to Zeppelin's interpreter setting, and find the interpreter for Spark. Click the edit button and add the path to Verdict's core jar file (which is created in the `target` directory when Verdict's source code is built) as a dependency.

For Spark 1.6.0, you can import Verdict and run queries as follows:

```
import edu.umich.verdict.VerdictSparkHiveContext

val vc = new VerdictSparkHiveContext(sc)                // sc: SparkContext
vc.sql("show databases").show(false)
df = vc.sql("select count(*) from instacart.orders")    // returns a Spark DataFrame
df.show(false)
```

For Spark 2.0, it is slightly different in importing Verdict and running queries:

```
import edu.umich.verdict.VerdictSpark2Context

val vc = new VerdictSpark2Context(sc)                // sc: SparkContext
vc.sql("show databases").show(false)
df = vc.sql("select count(*) from instacart.orders")    // returns a Spark DataFrame
df.show(false)
```

On Hive, Impala

Zeppelin can connect to Verdict on any database (including Hive, Impala) using the JDBC interface. For this, go to Zeppelin's interpreter setting, and click the **create** button. Enter "verdict-impala" (or any name you want) in "Interpreter Name", choose "jdbc" in "Interpreter Group", enter `edu.umich.verdict.jdbc.Driver` for `default.driver`, and enter `jdbc:verdict:impala://host:port/schema` for `default.url` (change the database name appropriately according to *JDBC in Java/Python applications*).

You may need to set `default.user` or other authentication fields as needed for your existing database connection (Verdict will pass those parameters as it makes another connection internally to your existing database).

Under **dependencies**, add the (preferably absolute) paths to all JDBC drivers for your existing databases and `verdict-core-version-with-dependencies.jar` (which is created in the `target` directory when Verdict is built).

In Jupyter

On PySpark

You can use Verdict in Jupyter (that connects to PySpark) by following the similar approach as in *Verdict-on-PySpark*. In other words, simply include the path to the Verdict's core jar file as a Driver's Java class path when starting a Jupyter notebook server:

```
$ export PYSPARK_DRIVER_PYTHO = "path-to-jupyter"
$ export PYSPARK_DRIVER_PYTHON_OPTS = "notebook"
$ export PYTHONPATH = $(pwd)/python:$PYTHONPATH
$ pyspark --driver-class-path $(pwd)/target/{{ site.verdict_core_jar }}
```

The above command will start the Jupyter server in which you can import PySpark modules.

On Hive, Impala, Amazon Redshift

You can connect to Verdict on any database that support JDBC connections (including Hive, Impala) as in *Python example*.

In Hue

Hue supports custom JDBC connections. Please see this page for instructions.

In Google Cloud Dataproc

You can use Verdict in Spark running on Google Cloud Dataproc by following a similar approach as just in Spark. First use gcloud command to SSH into Cloud Dataproc cluster master node, in which you should have Verdict's core jar file saved. Now you can launch `spark-shell` with core jar file specified `spark-shell --jars target/verdict-core-0.3.0-snapshot-jar-with-dependencies.jar`. Then you can import Verdict and run queries as usual:

```
import edu.umich.verdict.VerdictSparkHiveContext

val vc = new VerdictSparkHiveContext(sc)
vc.sql("show databases").show(false)
df = vc.sql("select count(*) from instacart.orders")
df.show(false)
```

One thing to notice is that in order for Spark 1.6 to work on Cloud Dataproc, you need to select image version to be 1.0 when creating the cluster (default is 1.2, which supports Spark 2.2).

CHAPTER 3

Supported queries

Overview

Verdict brings significant speedups for many important analytic queries. Before providing a detailed presentation on the queries Verdict can bring speedups, the following summary provides a quick overview.

1. Verdict brings speedups for the queries including **aggregate functions**.
   (a) Verdict can speedup most common aggregate functions
   (b) The only known exceptions are extreme statistics: `min` and `max`.
2. Verdict can speedup the queries including joins of multiple tables and subqueries.
   (a) For relatively simple queries (whose depth is no more than three), Verdict mostly brings speedups.
   (b) For deeper, complex queries (such as aggregations over aggregations), speedups more depend on Verdict's sample planner.
   (c) The cost of the worst cases will simply be equivalent to running the original queries.

Querying database metadata

show databases;

Displays database names. When Verdict runs on Spark, it displays the tables accessible through `HiveContext`.

use database_name;

Set the default database to `database-name`.

show tables [in database_name];

Displays the tables in `database-name`. If the database name is not specified, the tables in the default database are listed. If `database-name` is not specified and the default database is not chosen, Verdict throws an error.

describe [database_name].table_name;

Displays the column names and their types.

show samples [for database_name];

Displays the samples created for the tables in the `database-name`. If `database-name` is not specified and the database name is not specified, the samples created for the default database are specified. If the default database is not chosen, Verdict throws an error.

Creating samples

create [XX%] sample of [database_name.]table_name;

Creates a set of samples for the specified table. **This is the recommended way of creating sample tables.** Verdict analyzes the statistics of the table and automatically creates desired samples. If the sampling probability is omitted, 1% samples are created by default.

Currently, Verdict creates three types of samples using the following rule:

1. A uniform random sample
2. Stratified samples for low-cardinality columns (distinct-count of a column <= 1% of the total number of tuples).
3. Universe samples for high-cardinality columns (distinct-count of a column > 1% of the total number of tuples).

Find more details on each sample type below.

create [XX%] uniform sample of [database_name.]table_name;

Creates XX% (1% by default) uniform random sample of the specified table.

Uniform random samples are useful for estimating some basic statistics, such as the number of tuples in a table, average of some expressions, etc., especially when there are no `group-by` clause.

create [XX%] stratified sample of [database_name.]table_name on column_name;

Creates XX% (1% by default) stratified sample of the specified table.

Stratified samples ensures that no attribute values are dropped for the column(s) on which the stratified samples are built on. This implies that, in the result set of `select group-name, count(*) from t group by group-name`, every `group-name` exists even if Verdict runs the query on a sample. Stratified samples can be very useful when users have some knowledge on what columns will appear frequently in the `group-by` clause or in the `where` clause. For example, stratified samples can be very useful for streaming data where each record contains a timestamp. Building a stratified sample on the timestamp will ensure rare events are never dropped in the sampling process. Note that the sampling probabilities for tuples are not uniform in stratified sampling. However, Verdict automatically considers them and produce correct answers.

create [XX%] universe sample of [database_name.]table_name on column_name;

Creates XX% (1% by default) universe sample of the specified table.

Universe samples are hashing-based sampling. Verdict uses hash functions available in a database system it works with. Universe samples are used for estimating *distinct-count* of high-cardinality columns. The theory behind using universe samples for *distinct-count* is closely related to the HyperLogLog algorithm. Different from HyperLogLog, however, Verdict's approach uses a sample; thus, it is significantly faster than HyperLogLog or any similar approaches that scan the entire data. Also, universe samples are useful when a query includes joins. The equi-joins of two universe samples (of different tables) built on the join key preserves the cardinality very well; thus, it can produce accurate answers compared to joining two uniform or stratified samples.

Analyzing data: query structure

Verdict processes the standard SQL query in the following form:

```
select expr1, expr2, ...
from table_source1, table_source2, ...
[where conditions]
[group by expr1, ...]
[order by expr1]
[limit n;]
```

Find more details on the supported statements below.

Analyzing data: aggregate functions

Supported aggregate functions

Verdict brings speedups for the following aggregate functions or combinations of them:

Aggregate function	Description
count(*)	Counts the number of tuples that satisfy selection conditions in the where clause (if any)
sum(col-name)	Computes the summation of the *non-null* attribute values in the "col-name" column.
avg(col-name)	Computes the avreage of the *non-null* attribute values in the "col-name" column.
count(distinct col-name)	Computes the number of distinct attributes in the "col-name" column; only one column can be specified.

Future supported aggregate functions

Verdict will be extended to support the following aggregate functions in the future:

Aggregate function	Description
var_pop(col-name)	population variance
var_samp(col-name)	sample variance
stddev_pop(col-name)	population standard deviation
stddev_samp(col-name)	sample standard deviation
covar_pop(col1, col2)	population covariance
covar_samp(col1, col2)	sample covariance
corr(col1, col2)	Pearson correlation coefficient
percentile(col1, p)	p should be within 0.01 and 0.99 for reliable results

No-speedup aggregate functions

Verdict does not bring speedups (even in the future) for the following extreme statistics:

Aggregate function	Description
min(col-name)	Min of the attribute values in the "col-name" column
max(col-name)	Max of the attribute values in the "col-name" column

If a query includes these no-speedup aggregate function(s), Verdict uses the original tables (instead of the sample tables) for processing those queries.

Analyzing data: other functions

In general, every (non-aggregate) function that is provided by existing database systems can be processed by Verdict (since Verdict will simply pass those functions to those databases). Please inform us if you want certain functions to be included. We will quickly add them.

Mathematical functions

Function	Description
round(double a)	
floor(double a)	
ceil(double a)	
exp(double a)	
ln(double a)	a natural logarithm
log10(double a)	log with base 10
log2(double a)	log with base 2
sin(double a)	
cos(double a)	
tan(double a)	
sign(double a)	Returns the sign of a as '1.0' (if a is positive) or '-1.0' (if a is negative), '0.0' otherwise
pmod(int a, int b)	a mod b; supported for Hive and Spark; See this for more information.
a % b	a mod b
rand(int seed)	random number between 0 and 1
abs(double a), abs(int a)	an absolute value
sqrt(double a)	

String operators

Function	Description
conv(int num, int from_base, int to_base), conv(string num, int from_base, int to_base)	Converts a number from a given base to another; supported for Hive and Spark
substr(string a, int start, int len)	Returns the portion of the string starting at a specified point with a specified maximum length.

Other functions

Function	Description
fnv_hash(expr)	Returns a consistent 64-bit value derived from the input argument; supported for Impala; See this page for more information.
md5(expr)	Calculates an MD5 128-bit checksum for the string or binary; supported for Hive and Spark
crc32(expr)	Computes a cyclic redundancy check value for string or binary argument and returns bigint value; supported for Hive and Spark

Analyzing data: table sources, filtering predicates, etc.

Table sources

You can use a single base table, equi-joined tables, or derived tables in the from clause. Verdict's sample planner automatically finds the best set of sample tables to process your queries. However, if samples must not be used for processing your queries (due to unguaranteed accuracy), Verdict will use the original tables.

Verdict's sample planner is rather involved, so we will make a separate document for its description.

Note: Verdict's query parser currently processes only inner joins, but it will be extended to process left outer and right outer joins.

Filtering predicates (i.e., in the where clause)

Predicate	Description
p1 AND p2	logical and of two predicates, p1 and p2
p1 OR p2	logical or of two predicates, p1 and p2
expr1 COMP expr2	comparison of two expressions, expr1 and expr2, using the comparison operator, COMP. The available comparison operators are =, >, <, <=, >=, <>, !=, !>, !<, <=, >=, <, >, !>, !<
expr COMP (subquery)	comparison of the value of expr and the value of subquery. The subquery must return a single row and a single column
expr1 NOT? BETWEEN expr2 AND expr3	returns true if the value of expr1 is between the value of expr2 and the value of expr3.
expr1 NOT? LIKE expr2	text pattern search using wild cards. See this page for more information.
expr IS NOT? NULL	test if the value of the expression is null.

Dropping samples

(delete | drop) [XX%] samples of [database-name.]table-name;

Drop all the samples created for the specified table. The sampling ratio is 1% is not specified explicitly.

(delete | drop) [XX%] uniform samples of [database-name.]table-name;

Drop the uniform random sample created for the specified table. The sampling ratio is 1% is not specified explicitly.

(delete | drop) [XX%] stratified samples of [database-name.]table-name on column-name;

Drop the stratified sample created for the specified table. The sampling ratio is 1% is not specified explicitly.

(delete | drop) [XX%] universe samples of [database-name.]table-name on column-name;

Drop the universe sample created for the specified table. The sampling ratio is 1% is not specified explicitly.

CHAPTER 4

Configurations

Changing Verdict configuration

1. **Creating a configuration file** (i.e., `verdict.properties`) in the `config` directory. In order for the options in this file to be effective, Verdict should be build again using a `mvn package` command.
2. **Setting environment variables**: When Verdict's Java instance is created, it scans all environment variables and stores them in Verdict's configuration if they start with `verdict`.
3. **Passing parameters when making a connection**
   - **Spark**: You can pass parameters using the `--conf` argument. For example:

     ```
     $ spark-shell --jars target/{{ site.verdict_core_jar }} --conf verdict.
     ↪loglevel=warn
     ```

     will set the Verdict's log level to `warn`.
   - **Hive, Impala**: You can pass the parameters in the JDBC connection string. For example:

     ```
     $ veeline/bin/veeline -h "impala://host:port/default;verdict.loglevel=warn"
     ```

     will set the Verdict's log level to `warn`.
4. **Using a Verdict query**: You can issue a `set key=value` query after a Verdict instance is created.

The higher-numbered items take higher precedences.

Verdict configuration options

Basic options

key	Default values	Description
verdict.bypass	false	If set true, Verdict simply passes all the query to the underlying database and returns the result to the user without any modifications. The only exception is *set bypass=false*.
verdict.bypass	false	If set true, Verdict simply passes all the query to the underlying database and returns the result to the user without any modifications. The only exception is *set bypass=false*.
verdict.loglevel	info	Determines the log level; should be one of "debug", "info", "warn", "error". *loglevel* is a synonym for *verdict.loglevel*.
verdict.relative_target_cost	10%	Keeps the cost of Verdict's query processing under a certain level. Verdict chooses a set of the largest samples under this limit.

JDBC options

Key	Default values	Description
verdict.jdbc.user		If set, pass as a "user" value when making a JDBC connection to an underlying database.
verdict.jdbc.password		If set, pass as a "password" value when making a JDBC connection to an underlying database.
verdict.jdbc.kerberos_principal	n/a	If set, makes an appropriate Kerberos authentication when making a JDBC connection to an underlying database. `principal` is a synonym for `verdict.jdbc.kerberos_principal`.
verdict.jdbc.ignore_user_credentials	false	If set to true, "user" and "password" values are not passed even if they are set in `verdict.jdbc.user` and `verdict.jdbc.password`.
verdict.jdbc.dbname		One of "impala", "hive2". The database name specified in the JDBC connection string is set.
verdict.jdbc.host	127.0.0.1	The host that will be used for a JDBC connection.
verdict.jdbc.port		The port number that will be used for a JDBC connection.
verdict.jdbc.schema	default	The default schema (or database) that a JDBC connection will connect to.
verdict.jdbc.hive2.default_port	10000	The default port number that will be used for a hive2 connection if the port number is not explicitly set in the passed JDBC connection string.
verdict.jdbc.hive2.class_name	com.cloudera.hive.jdbc4.HS2Driver	The class name of the JDBC driver that will be used for a hive2 connection.
verdict.jdbc.impala.default_port	21050	The default port number that will be used for an impala connection if the port number is not explicitly set in the passed JDBC connection string.
verdict.jdbc.impala.class_name	com.cloudera.impala.jdbc4.Driver	The class name of the JDBC driver that will be used for an impala connection.

Spark options

Key	Default values	Description
verdict.spark.cache_samples	true	(Lazily) cache the sample tables created by Verdict using Spark's *DataFrame#cache()* function. The sample tables are typically significantly smaller than the original tables. Caching sample tables will lower query response times.

CHAPTER 5

API Documentation

See Javadoc.

This Javadoc pages are created in `target/apidocs` if you run `mvn javadocs:aggregate`.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search