Verdict Documentation

Release 0.6.0

Verdict Team

Nov 07, 2019

Contents

This documentation introduces Verdict's core concepts and operations. Verdict is developed to provide fast and resource-efficient analytics, working **on top of** existing engines (thus, no modifications to the engines).

Unlike other systems, Verdict **predicts** the answers (instead of simply aggregating individual values) by exploiting statistical properties of analytical workloads. These predictions are performed almost instantly and come with tight accuracy guarantees. In many cases, you get more than 99% accurate answers in less than a second.

CHAPTER 1

Introduction

Verdict is a lightweight system that can accurately answer your analytical queries. Instead of aggregating every value in your data, Verdict **predicts** the answers using its concise data summary, or equivalently *samples*. To maximize the efficiency, verdict incorporates advanced types of samples.

Verdict is designed to operate on top of existing analytics engines (e.g., Facebook Presto, Google BigQuery, Amazon Redshift). Thus, no data migration is required.

Verdict has two modes of operations:

1. **stream**: Verdict gives you a series of answers as continuously processing data. Each answer is statistically unbiased. Internally, Verdict performs optimization to maximize the accuracy of the answers. Typically, even the first answer (available in less than a second) is almost identical to the exact one, however large your data is.
2. **traditional**: Verdict gives you a single answer that satisfies a requested accuracy level (1% relative error by default). Again, Verdict's latency doesn't increase however large you data is.

Note: We are refactoring our old code; thus, some of the features mentioned in our website may not be available yet.

CHAPTER 2

Quickstart

In this quickstart, we will install Verdict using its public docker image. Using included scripts, we will connect to 100GB test data stored in our Amazon S3 bucket. Our demo will use Facebook Presto as an underlying analytics engine, which we will also install using a docker image.

2.1 Install

2.1.1 Requirements

1. A machine with 64GB or more memory
2. `docker` (get here) and `docker-compose` (get here).
3. AWS access key and its secret credential, which can be obtained freely from the official page.

Note: Most computational resource is for running Presto. Verdict itself is quite light-weight.

2.1.2 Steps

First, set your AWS access key and is credential to environment variables:

```
export AWS_ACCESS_KEY_ID=<your access key>
export AWS_SECRET_ACCESS_KEY=<your secret credential>
```

Second, pull and run docker containers:

```
curl -s https://raw.githubusercontent.com/verdict-project/verdict/master/docker-
↪compose-64gb.yaml \
    | docker-compose -f - up
```

Running `docker ps` will show two containers named `docker-verdict` and `docker-presto`.

Third, create external tables that connect to 100GB dataset:

```
docker exec docker-verdict define_presto_tables_and_samples.sh
```

2.2 Connect

First, open the python shell in docker:

```
docker exec -it docker-verdict python
```

Then, make a connection to Presto with verdict.

```
import verdict
v = verdict.presto(presto_host='presto')
```

2.2.1 Info method

To see the tables indexed by verdict, use the `info()` method.

```
v.info()
# {   'Registered Tables': [   'hive.tpch_sf100.orders',
#                              'hive.tpch_sf1.orders',
#                              'hive.tpch_sf100.lineitem',
#                              'hive.tpch_sf1.lineitem']}
```

You can pass an argument to `info()` to see more information about it.

```
v.info('hive.tpch_sf100.lineitem')
# {   'Column Names and Types': {   'l_comment': 'varchar',
#                                   'l_commitdate': 'date',
#                                   'l_discount': 'double',
#                                   ...
#                                   's_nationkey': 'bigint',
#                                   's_phone': 'varchar'},
#     'Samples': [   's9487fcfadd71477ead92b02cf587e525_rowid',
#                    's63d739590a784d959b3d1e8694ef5e3al_orderkey'],
#     'Row Count': 600037902}
```

From the above output, you can see two samples have been created for `hive.tpch_sf100.lineitem`. Verdict uses these samples (automatically) to speed up its query processing. You may be curious how they are created, but before describing how to create them, let's first run some queries.

2.3 Run Queries

2.3.1 Traditional Mode

To issue queries in the *traditional* mode, we use the `sql()` method. For example,

```
v.sql("select count(*) from hive.tpch_sf100.lineitem")
#           c1
# 0  597536768

v.sql("select count(*) from hive.tpch_sf100.lineitem where l_linestatus = 'F'")
#           c1
# 0  299372544
```

The above queries will return answers almost instantly.

2.3.2 Stream Mode

To run queries in the stream mode, use `sql_stream()` method. This method returns an iterator from which you can retrieve a series of answers. For example,

```
itr = v.sql_stream("select count(*) from hive.tpch_sf100.lineitem where l_linestatus␣
↪= 'F'")

for result in itr:
    print(result)
```

To see more example queries for both traditional and stream modes, see *Example Queries*.

2.3.3 Bypass Queries

Finally, you can issue any queries directly to the backend engine (Presto here) by sending a query to `sql()` method with the prefix `bypass`. For example,

```
# this will return an exact answer, but will take longer
v.sql("bypass select count(*) from hive.tpch_sf100.lineitem where l_linestatus = 'F'")
#        _col0
# 0  299979732
```

The answers are almost identical, but it takes longer this time since the query is directly processed by the backend engine.

You can also issue metadata queries or DDL queries by prefixing `bypass`.

```
# regular presto query
v.sql("bypass show catalogs")

# The above method will return this:
#
#   Catalog
# 0    hive
# 1     jmx
# 2  memory
# 3  system
# 4    tpch

v.sql("bypass show schemas in hive")
#                 Schema
# 0              default
# 1  information_schema
# 2              verdict
```

(continues on next page)

(continued from previous page)

```
# 3            tpch_sf1
# 4          tpch_sf100

v.sql("bypass show tables in hive.tpch_sf100")
#       Table
# 0  lineitem
# 1    orders
# 2  partsupp
```

CHAPTER 3

Example Queries

We show several example queries here. To run the following examples, first import verdict and create its instance as follows:

```
import verdict
v = verdict.presto(presto_host='presto')
```

3.1 Traditional Mode

Use the `sql(query_string)` method to obtain a single accuracy-guaranteed answer. The error level can be specified at query time (1% relative error, by default) .

Note: To see details of the class of SQL queries supported by Verdict, see *Query Syntax*.

3.1.1 Count with arbitrary filters

```
v.sql("""
select l_shipmode, count(*)
from hive.tpch_sf1.lineitem_premerged
where l_shipdate >= date '1994-01-01' or l_shipdate <= date '1995-01-01'
group by l_shipmode
order by l_shipmode
""")
```

3.1.2 Joins of two large tables

```
v.sql("""
select o_orderstatus, count(*)
from hive.tpch_sf100.lineitem_premerged l inner join
     hive.tpch_sf100.orders_premerged o on l_orderkey = o_orderkey
group by o_orderstatus
order by o_orderstatus
""")
```

3.1.3 Exists predicate expressed using a join (TPC-H Q4)

```
v.sql("""
select
    o_orderpriority,
    count(*) as order_count
from
    hive.tpch_sf100.orders_premerged o left join
    (
        select l_orderkey, count(*) exist_count
        from hive.tpch_sf100.lineitem_premerged
        where l_commitdate < l_receiptdate
        group by l_orderkey
    ) t on o_orderkey = l_orderkey
where
    o_orderdate >= date '1996-05-01'
    and o_orderdate < date '1996-08-01'
    and exist_count > 0
group by
    o_orderpriority
order by
    o_orderpriority
""")
```

3.2 Stream Mode

Simply change `sql` to `sql_stream`. Then, verdict returns an iterator from which you can obtain a series of answers that converge to the exact one. Often, this is called progressive analytics. For example,

3.2.1 Count with arbitrary filters

```
results_itr = v.sql_stream("""
select count(*)
from hive.tpch_sf100.lineitem_premerged
where l_returnflag = 'R' and
      (l_shipdate >= date '1994-01-01' or l_shipdate <= date '1995-01-01')
""")

for result in results_itr:
    print(result)
```

These successive results can be used by upfront applications (e.g., visualization libraries) to deliver the results in a more intuitive way.

CHAPTER 4

Query Syntax

This page describes Verdict's query syntax. In general, Verdict follows the syntax of the standard SQL. There are some temporary limitations (highlighted using blue boxes below), which will be gradually lifted in future versions.

4.1 Query

The Verdict query must be an aggregate query with optional groupby, orderby, and limit clauses.

```
query := SELECT aggregate_function, ...
         FROM relation
         [GROUP BY base_attr, ...]
         [ORDER BY alias, ...]
         [LIMIT int]

alias := str

base_attr := str
```

Example:

```
SELECT sum(price) as price_sum, count(*) as c
FROM hive.tpch_sf100.lineitem_premerged
GROUP BY l_linestatus
ORDER BY price_sum
LIMIT 5
```

Note: If the groupby clause is present, the grouping columns will be prepended in the result set. In the future, this behavior will be changed to follow the standard SQL semantics.

4.2 Relation

A relation can be a base table, joins of relations, or subqueries.

```
relation := base_table |
            relation join_expr |
            (SELECT attr_alias, ...
             FROM relation
             [WHERE predicate]
             [GROUP BY attr, ...]) alias

join_expr := join_type relation ON base_attr = base_attr

join_type := INNER JOIN |
             LEFT JOIN |
             RIGHT JOIN |
             OUTER JOIN

attr_alias := attr [AS] alias
```

Examples of the relation:

```
-- example 1
(
    select
        l_returnflag,
        l_quantity,
        l_extendedprice,
        l_discount,
        l_extendedprice,
        l_extendedprice * (1 - l_discount) disc_price,
        l_extendedprice * (1 - l_discount) * (1 + l_tax) charge,
        l_returnflag,
        l_linestatus
    from
        hive.tpch_tiny.lineitem_premerged
    where
        l_shipdate <= date '1998-12-01'
) t1

-- example 2
(
    select
        l_orderkey,
        l_extendedprice * (1 - l_discount) revenue,
        o_orderdate,
        o_shippriority
    from
        hive.tpch_tiny.lineitem_premerged l
            inner join hive.tpch_tiny.orders_premerged o
            on l_orderkey = o_orderkey
    where
        c_mktsegment = 'BUILDING'
        and o_orderdate < date '1995-03-22'
        and l_shipdate > date '1995-03-22'
) t1
```

Note that in the above example, `t1` is the alias of the subquery relation. If the `alias` is omitted the same name is assigned for base attributes and an arbitrary name is assigned for derived attributes (e.g., `l_extendedprice * (1 - l_discount)`).

Note: The join type must be equijoin (whether it be inner, left, or right). The attribute that appears on the left-hand side of the equality sign is assumed to the attribute in the left join table. The similar rule applies for the right attribute.

4.2.1 Attribute

An attribute can be a base attribute or some functions of it.

```
attr := base_attr |
        constant |
        scalar_function |
        aggregate_function

constant := int |
            str |
            date '0000-00-00' |
            timestamp '0000-00-00 00:00:00'

predicate := logical_expr |
             comparative_expr

logical_expr := predicate AND predicate |
                predicate OR predicate  |
                NOT predicate

comparative_expr := attr > attr |
                    attr < attr |
                    attr >= attr |
                    attr <= attr |
                    attr <> attr |
                    attr in [ constant, ... ]
```

We describe more details about functions in the subsequent sections.

4.3 Scalar Functions

A scalar function is the function that produces an output value for each input value.

```
scalar_function := math_function |
                   string_function
```

4.3.1 Mathematical Functions

```
math_function := attr + attr |
                 attr - attr |
                 attr * attr |
                 attr / attr |
                 floor(attr) |
```

(continues on next page)

(continued from previous page)

```
                 ceil(attr)  |
                 round(attr)
```

4.3.2 String Functions

```
string_function := SUBSTR(attr, start, length) |
                   TO_STRING(attr) |
                   CAST(attr AS VARCHAR) |
                   CONCAT(attr, attr) |
                   LENGTH(attr) |
                   REPLACE(old, new) |
                   UPPER(attr) |
                   LOWER(attr) |
                   STARTSWITH(attr, pattern) |
                   CONTAINS(attr, pattern) |
                   ENDSWITH(attr, pattern)
```

Note: We are adding more scalar functions.

4.4 Aggregate Functions

An aggregate function is a function that produces a single row given multiple rows.

```
aggregate_function := COUNT(*) |
                      SUM(base_attr) |
                      AVG(base_attr)
```

Note: To use a derived attribute within aggregate functions, you can first create new attributes using subqueries, then attribute those new attributes.

CHAPTER 5

Sample Creation

Verdict uses various types of samples to estimate the final query answers. Verdict automatically determines what types of samples to use at query time; however, those samples must exist in advance for Verdict to perform proper operations.

Each sample is uniquely determined by two properties:

1. **Table name**: This is the name of the base table.
2. **Column name**: This column serves as the *key* for creating a sample. Specifically, a sample is created by *sampling from the domain of the key column* (not simply randomly choosing individual records). When the key is set to `_rowid` (a special keyword), a sample is created by randomly choosing values from the row numbers, which is equivalent to uniform random sampling.

Before describing what types of samples Verdict needs, we describe the method for creating samples.

5.1 Sample Creation Method

To create a sample, use `VerdictSession.create_sample(table_name, key_col)`. For example,

```
table_name = "hive.tpch_sf100.lineitem"
key_col = "l_orderkey"
v.create_sample(table_name, key_col)
```

5.2 What Samples? Rule of Thumb

We describe what samples are likely to be needed for processing most queries.

1. **Table name**: We need samples for *large* tables. These tables are typically the tables including a large number of historical records.
2. **Column name**: `_rowid` and every column whose *support* is large, where *support* means the number of unique attribute values in the column.

5.3 More About Verdict's Sampling

5.3.1 What Happens?

When the above method is called, Verdict performs the following operations:

1. Verdict gathers basic statistics about the table (i.e., number of rows, column names and types). This information is used for query processing as well as sampling.
2. Verdict writes a SQL statement for sampling and sends it to the backend engine.
3. Verdict stores the information about the sample (i.e., what is the original table, what is the key column, etc.) in its metastore (i.e., Redis).
4. Verdict stores a small fraction of the sample (called *cache*) in its in-memory engine (i.e., Pandas SQL). The cache is used for estimating the sample size needed for satisfying the accuracy.

5.3.2 Limitations

The current version of Verdict has some known limitations:

1. Verdict does not automatically maintain the consistency between the original table and its samples. Thus, if new records are inserted into the original table, the sample becomes stale.
2. Verdict does not automatically determine what samples are needed for your data. The manual steps must be performed as described below.

We are working to address these limitations.

CHAPTER 6

System Overview

This page overviews Verdict's architecture. First, we describe how Verdict interacts with users and other analytics engines. Next, we describe Verdict's internal architecture.

6.1 Deployment

There are three entities involved:

1. **Client (or User)**: The client or user is the entity that composes/issues analytical queries and consumes their results. The client can be a data analyst, BI tools, Web dashboards, operational services, etc.
2. **Verdict**: Verdict is our library that imposes itself as a middleware between the client and analytics engines. Verdict intercepts the queries from the client and send rewritten queries to analytics engines.
3. **Backend Engine**: This entity performs traditional aggregate-style computations. The backend engines can be modern cloud services (e.g., BigQuery, Redshift) or traditional database engines (e.g., MySQL, Oracle, etc.).

6.1.1 Query workflow

We describe Verdict's query processing workflow starting from the query issued by the client. Given a query from the client, Verdict performs **planning**, which consists of two following operations:

1. Verdict look for the data summaries created for the tables in the query
2. Verdict determines how much portion of summaries to process. That is, the actual amount of data to process should be small enough to reduce query latency as much as possible. At the same time. the size should be large enough to ensure enough accuracy.

Once the planning is done, Verdict rewrites the query; the rewritten query uses the summaries in place of the original tables. Then, Verdict sends the rewritten query to an analytics engine (e.g., Presto). The written query (which is sent to the analytics engine) looks like a regular query to the analytics engine. After Verdict receives the answer for its rewritten query, Verdict performs post-processing, and returns a final result back to the client.

Note: For optimal performance, Verdict also optionally caches summaries and uses its in-memory analytics engine.

6.2 Verdict Architecture

In this section, we describe internal components of Verdict and their functions. These are Verdict's internal components.

1. **Query Planner**: This component determines what summaries to use. Verdict chooses the optimal sample type and ratio among available summaries.
2. **Sampler**: This component specifies the rules for different types of samples.
3. **SQL2IR**: This component translates SQL to Verdict's internal json representation
4. **Drivers**: This component translates Verdict's internal json representation to the engine-specific language.

6.2.1 Internal workflow

Offline Sampling

Given a source table name and a sample type, Sampler specifies how the sample table should be structured. For example, a sample table can be composed of multiple partitions where each partition corresponds to a random subset of a different size. Depending on attribute values, different sampling probabilities may be assigned for faster convergence.

Verdict then passes this structure information to the driver for a target analytics engine. Then, this driver composes a query that can actually run on the target engine (e.g., SQL for Presto).

Query Processing

Given a query, Verdict's Query Planner determines the right sample among the available ones. This planning stage considers various criteria as follows:

1. **User-requested accuracy:** You may want different accuracy requirement for faster processing or more accurate answers.
2. **Types of aggregate functions:** Depending on the types of aggregate functions (e.g., avg, count), we use different formulations to derive the optimal amount of data to process.
3. **Query operations:** Depending on the join patterns, groupby clauses, etc., we may need to use different types of samples in a specific way.

Once the planning is done, Verdict's composes a query in its internal json format, which is sent to engine-specific drivers. Then, these drivers translate them and run on analytics engines.

Note: Creating right types of samples is critical for Verdict's operations. Although we provide general mechanisms and guidelines, this may not always be straightforward to all users or simply you may not have time to understand it. To help such cases, we are developing an automatic designer.

CHAPTER 7

Reference

7.1 Entry Point

7.2 Verdict Session

7.3 Internal Interface

7.3.1 SQL <-> Verdict Query

7.3.2 Verdict Query <-> Relational Objects

7.4 Query Processing