tesseract Documentation

Release 1.0.0

Elliot Chance

June 11, 2015

Contents

tesseract is a SQL object database with Redis as the backend, think of it like a document store that you run SQL statements against.

Since it is backed by Redis and queries are compiled to Lua it makes running complex queries on complex data very fast (all considered). The entire server is written in Python and uses an [extremely simply client protocol](server-protocol.html).

CHAPTER 1

Getting Started

1.1 Installation

Since tesseract is very alpha I have not uploaded releases yet to `pip` so the easiest way get it is to clone out the repo and run off the stable `master` branch. You can view the *Changelog*.

```
git clone https://github.com/elliotchance/tesseract.git
```

1.2 Running the Server

You must have Redis running locally on the standard port. If not, run:

```
redis-server &
```

Then start the tesseract server. It's not wise to run it in the background so you can pay attention to errors and crashes during this wonderful time of alpha.

```
bin/tesseract
```

Remember that if you pull the latest changes for tesseract you will have to restart the server for those changes to take effect. Simply `CTRL+C` and run the command above again.

CHAPTER 2

SQL Reference

2.1 BEGIN

See *START TRANSACTION*.

2.1.1 See Also

- *Transactions*

2.2 COMMIT

2.2.1 Syntax

```
COMMIT [ WORK | TRANSACTION ]
```

WORK Optional keyword. It has no effect.

TRANSACTION Optional keyword. It has no effect.

2.2.2 Overview

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others.

2.2.3 Compatibility

The SQL standard only specifies the two forms `COMMIT` and `COMMIT WORK`.

2.2.4 See Also

- *Transactions*

2.3 CREATE INDEX

`CREATE INDEX` creates a mixed type index on a single column of a table.

Index are used to vastly improve performance when fetching data. By creating a separate mapping of values in a record that can be used to retrive records without needing to physically search them.

2.3.1 Syntax

```
CREATE INDEX <index_name> ON <table_name> (<field>)
```

index_name The name of the index not exist. Index names must be globally unique - that is different tables cannot have a index with the same name.

table_name The table to create the index on. Every time a record is added, modified or deleted it will require all index entries for that row to be updated.

field The field to index.

2.3.2 Overview

> For those already comfortable with SQL and indexes you need not read must more beyond this point. Indexes are intended to work exactly the same way in tesseract.

Each index will require space proportional to the amount of data that is in the original table and how big each entry is to store. As well as the space consideration every time a record is modified it will have to update all the indexes for that table with respect to the new or modified record. This is why it is not wise to create lots of indexes unless you have a specific need to filter on that attribute.

Lets take a real world example, lets say we have a simple address book that stores entries like:

```
{
    "first_name": "Bob",
    "last_name": "Smith",
    "phone": "1234 5678",
}
```

Finding all Bob's is simple:

```
SELECT * FROM contacts WHERE first_name = 'Bob'
```

This is fine for a small address book because computer can search very quickly. But there is a point in which this starts to make less sense and require more and more resources for the same operation - what if there were thousands or even million of contacts?

Using the example above we can run an explain to see how tesseract is deciding to carry out the SQL:

```
EXPLAIN SELECT * FROM contacts WHERE first_name = 'Bob'
```

```
[
    {"description": "Full table scan of 'contacts'"}
    {"description": "Filter: first_name = \"Bob\""}
]
```

It is choosing to do it this wway becuase it has no other choice. But we can introduce and index on the `first_name`:

```
CREATE INDEX contacts_first_name ON contacts (first_name)
```

Since indexes names must be globally unique it is a good idea (and often common practice) to put the table name and column name as the index name for both clarity and avoiding naming collisions.

By creating the index it will store all the `first_names` in a separate place, in a special order/structure and will automaitcally maintain this index with every modification.

Now we can run the same `EXPLAIN`:

```
EXPLAIN SELECT * FROM contacts WHERE first_name = 'Bob'
```

```
[
    {"description": "Index lookup using contacts_first_name for value \"Bob\""}
]
```

It does not matter if we have 10 contact records or 10 million the lookup time will be almost the same (provided the number of Bob's that exists doesn't scale up with the number of records) - at the cost of slightly more RAM.

2.4 CREATE NOTIFICATION

`CREATE NOTIFICATION` tells the tesseract server to publish changes to the Redis pub/sub model.

2.4.1 Syntax

```
CREATE NOTIFICATION <notification_name>
ON <table_name>
[ WHERE <where_clause> ]
```

notification_name The name of the notification must be unique and non-existent. It follows the same rules as naming an entity like a table.

table_name The table to watch for changes.

where_clause Any expression which will cause the notification to fire only if it evaluates to `true`:

```
CREATE NOTIFICATION bobs
ON people
WHERE first_name = 'Bob';
```

2.4.2 Notes

Multiple notifications can be fired for a single insert but is limited to one notification per `NOTIFICATION` defined.

Notification use the Redis PUB/SUB model, so when the actual notification is fired it is sent as a publish through Redis. Your notification name is the channel that the message will be published to.

This means that software does not need to know anything about the tesseract server if they only intent to subscribe. Published notifications are the complete JSON record.

2.5 DELETE

`DELETE` is used to remove data.

2.5.1 Syntax

```
DELETE FROM <table_name>
[ WHERE <where_clause> ]
```

table_name The table name to remove the objects from. If the table does not exist then the statement will have no effect.

where_clause An expression that determines which records should be removed. If the `WHERE` clause is omitted all records will be deleted.

2.6 DROP INDEX

`DROP INDEX` deletes an already defined index.

2.6.1 Syntax

```
DROP INDEX <index_name>
```

index_name The name of the index must already exist. Index names must be globally unique - that is different tables cannot have a index with the same name.

2.7 DROP NOTIFICATION

`DROP NOTIFICATION` deletes an already defined notification.

2.7.1 Syntax

```
DROP NOTIFICATION <notification_name>
```

notification_name The name of the notification must already exist or an error it returned.

2.8 DROP TABLE

`DROP TABLE` deletes all data for a table and associated indexes.

2.8.1 Syntax

```
DROP TABLE <table_name>
```

table_name The name of table. If the table does not exist this command will effectively do nothing.

2.9 INSERT

`INSERT` is used to add an object to a table.

2.9.1 Syntax

```
INSERT INTO <table_name>
<json_object>
```

table_name The table name to insert the object into. This table will be created by simply adding an object into it if it has never been inserted into.

json_object A JSON object.

2.9.2 Examples

```
INSERT INTO people
{ "first_name": "John", "last_name": "Smith" }
```

2.10 ROLLBACK

2.10.1 Syntax

```
ROLLBACK [ WORK | TRANSACTION ]
```

WORK Optional keyword. It has no effect.

TRANSACTION Optional keyword. It has no effect.

2.10.2 Overview

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Use `COMMIT` to successfully terminate a transaction.

Issuing `ROLLBACK` when not inside a transaction does no harm, but it will provoke a warning message.

2.10.3 Compatibility

The SQL standard only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`.

2.10.4 See Also

- *Transactions*

2.11 SELECT

`SELECT` is used to receive data.

2.11.1 Syntax

```
[ EXPLAIN ]
SELECT <column_definitions>
[ FROM <table_name> ]
[ WHERE <condition> ]
[ GROUP BY <group_field> ]
[ ORDER BY <order_field> ]
[ LIMIT <limit> ]
```

EXPLAIN If the `EXPLAIN` keyword is present the SQL query is not actually run - but rather the query plan (the internal steps required to carry out the task) are returned. This is very useful for debugging slow queries and otherwise examining what tesseract is doing with a particular query.

column_definitions This can be an asterisk (`*`) to represent that the entire object should be retrieved without modification or a list of expressions.

table_name The table name to fetch the objects from. If the table does not exist (i.e. has no records) then no records will be returned since tables only come into existence when they have objects added to them.

You may omit the table just to parse expressions that do no need a table like:

```
SELECT 3 + 4
```

condition A filter expression.

group_field Rows will be collapsed into groups of distinct values of this field.

order_field This can be any field you wish to sort by. If field does not exist in a given record it will be sorted as if the value were `null`.

There are two modifiers for a column; `ASC` and `DESC` which represent **asc**ending and **desc**ending respectively. If no sort order is specified then `ASC` is assumed.

When data is sorted it is separated into four types:

- Booleans. Explicit `true` and `false`.
- Numbers. Integers and floating point, not including strings that look like numbers like `"123"`.
- Strings.
- Nulls.

Data will sorted by type, then value in the same order as above. That is to say:

- Any number is considered greater than a boolean.
- Any string is considered greater than a number.
- `null` is considered to be greater than any non-`null` value.

limit Limit the amount of records to be returned. This is performed after all operations on the sets are finished (such as ordering and grouping).

If you specify a `LIMIT` higher than the total number of records it will be the same as not specifying the limit at all (all records). Alternatively you can use `LIMIT ALL` to return all records.

You may also skip rows before the limit with `OFFSET`:

```
SELECT * FROM bla LIMIT 20 OFFSET 10
```

This will skip the first 10 rows and return a maximum of 20 rows - the limit is exclusive of the offset.

`LIMIT` is optional like `OFFSET`:

```
    SELECT * FROM bla OFFSET 10
```

If the offset is larger than the available rows then no rows will be returned.

It is important to note that all the rows up to the `LIMIT` + the `OFFSET` must be calculated internally so using a large `OFFSET` can be expensive. In some cases all records of the entire set must be calculated before the limit can be applied - such as when there is an `ORDER BY` or `GROUP BY` clauses.

2.12 START TRANSACTION

2.12.1 Syntax

```
START TRANSACTION
BEGIN [ WORK | TRANSACTION ]
```

2.12.2 Overview

`START TRANSACTION` or the alias `BEGIN` starts a transaction.

2.12.3 Compatibility

In the standard, it is not necessary to issue `START TRANSACTION` to start a transaction block: any SQL command implicitly begins a block. Tesseract's behavior can be seen as implicitly issuing a `COMMIT` after each command that does not follow `START TRANSACTION` (or `BEGIN`), and it is therefore often called "autocommit".

2.13 UPDATE

`UPDATE` is used to modify data.

2.13.1 Syntax

```
UPDATE <table_name>
SET <column_expressions>
[ WHERE <condition> ]
```

table_name The table that will have its object modified. If the table does not exist (i.e. has no records) then no records will be updated since tables only come into existence when they have objects added to them.

column_expressions

```
    <column_name1> = <expression1>, ...
```

A list of columns to be updated by an expression.

name = "Bob", something_else = 3 + 5 * 2

You may use any value of the record in the expression as well:

counter = counter + 1, old_value = counter - 1

condition A filter expression. If no *WHERE* clause is provided then all objects in the table will be updated.

CHAPTER 3

Functions and Operators

3.1 Data Types

There are six distinct data types in tesseract:

- `null`
- `boolean`
- `number`
- `string`
- `array`
- `object`

To allow tesseract to be more confomitive to the SQL standard it uses the same meaning for the value of `null`. In short, all operations using the value `null` will also return `null`:

```
SELECT null = null
```

```
null
```

An important note is that when an object does not contain a key the value for that key is assumed to be `null` and all the normal rules apply to it.

Booleans have a value of `true` or `false` and tesseract sees each as a distinct value, different from any other truthy or falsy value like in most other scripting languages:

```
SELECT 0 = false
```

```
Error: No such operator number = boolean.
```

Numbers are both integers and floating-point values. Strings that represent numbers do not mean the same thing:

```
SELECT "123" = 123
```

```
Error: No such operator string = number.
```

3.2 Operators

3.2.1 Arithmetic

Addition:

```
a + b
```

Subtraction:

```
a - b
```

Multiplication:

```
a * b
```

Division:

```
a / b
```

Modulo (remainder):

```
a % b
```

Power:

```
a ^ b
```

`a` and `b` must be `null` or `number`.

3.2.2 Equality

Equality:

```
a = b
```

For inequality:

```
a != b
a <> b
```

`a` and `b` must be the same type, and accepted types are:

- `null`
- `number`
- `string`
- `array`
- `object`

In the case of an `array` both arrays must be the same length and contain exactly the same element in the same order:

```
SELECT [1, 2] = [1, 2]
```

```
true
```

```
SELECT [1, 2] = [1, "2"]
```

```
false
```

```
SELECT [1, 2] = [2, 1]
```

```
false
```

In the case of an `object` both objects must contain the same amount of keys and each key must exist in both objects with the exact same value.

```
SELECT {"foo": 123} = {"foo": 123}
```

```
true
```

```
SELECT {"foo": 123} = {"foo": 123, "bar": null}
```

```
false
```

Inequality has all the same rules but in reverse.

3.2.3 Greater or Less Than

Greater than:

```
a > b
```

Greater than or equal to:

```
a >= b
```

Less than:

```
a < b
```

Less than or equal to:

```
a <= b
```

`a` and `b` must be the same type, and accepted types are:

- `null`
- `number`
- `string`

When comparing strings it follows the same rules as how Lua compares strings.

3.2.4 Concatenation

Concatenation:

```
a || b
```

Will concatenate the string representations of both sides. For example `3 || 5` is `35`. Special values will be converted as follows:

Value	String Representation
`null`	`""`
`true`	`"true"`
`false`	`"false"`

You cannot concatenate arrays or objects on either or both sides.

3.2.5 Logical

For all logical operations `a` and `b` are only allowed to be `null` or `boolean`.

Logical AND:

```
a AND b
```

Results:

AND	true	false
true	true	false
false	false	false

Logical OR:

```
a OR b
```

Results:

OR	true	false
true	true	true
false	true	false

3.2.6 Regular Expressions

Regular Expressions:

```
value LIKE regex
value NOT LIKE regex
```

`value` must be a string, but can be of any length.

`regex` uses the SQL rules for `LIKE` expressions.

Character	Description
`.`	Match any single character.
`%`	Match zero or more characters.

Examples

Test if a string starts with another string:

```
SELECT "Bob Smith" LIKE "Bob %"
```

Test if a string ends with another string:

```
SELECT "Bob Smith" LIKE "% Smith"
```

3.2.7 Checking Types

The following can be used to test the types of a value:

```
value IS null
value IS true
value IS false
value IS boolean
value IS number
value IS string
value IS array
value IS object
```

Each of the combinations can be used with `NOT` like:

```
value IS NOT boolean
```

The case of the type (`boolean`) is not important and there is no specific convention on case.

3.2.8 Set Membership

To test the existence of a value in a set:

```
a IN (b1, b2, ...)
a NOT IN (b1, b2, ...)
```

Will return `true` if `a` exists in one of the `b` values. There must be at least one `b` value. Comparison of each element follows the same rules as the = operator.

If `a` is `null` or any of the `b` values are `null` then the result is `null`. This is to conform is the SQL standard in dealing with `null` values.

3.2.9 Containment

To test if a value sits between two other values (inclusive):

```
a BETWEEN b AND c
a NOT BETWEEN b AND c
```

Is exactly equivalent to:

```
a >= b AND a <= c
a < b OR a > c
```

If at least one of `a`, `b` or `c` is `null` then the result will always be `null`.

3.2.10 Operator Precedence

Operator/Element	Associativity	Description
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for true, false, null
IN		set membership
BETWEEN		containment
LIKE ILIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

3.3 Aggregate Functions

Aggregate functions calculate a single value based ona set of values (this set may be zero values).

3.3.1 AVG() – Average

```
avg(<number>)
```

3.3.2 COUNT() – Count records

```
count(*)
count(<any>)
```

When the argument is * it will count all records without needing to provide any value.

3.3.3 MAX() – Maximum value

```
max(<number>)
```

3.3.4 MIN() – Minimum value

```
min(<number>)
```

3.3.5 SUM() – Total

```
sum(<number>)
```

3.4 Mathematical Functions

To comply with the SQL standard in all cases when a `null` is passed as an argument to any function below the result of the expression will be `null`. Values that are missing are treated as `null` values.

3.4.1 `ABS()` – Absolute value

Return the absolute value (positive value) of a number.

abs(<number>)

3.4.2 `CEIL()` – Round up

Round up to the next whole number.

ceil(<number>)

3.4.3 `COS()` – Cosine

Calculates the cosine of an angle.

cos(<number>)

3.4.4 `FLOOR()` – Round down

Remove the fractional part of a number. This is often refered to as truncating a number.

floor(<number>)

3.4.5 `SIN()` – Sine

Calculates the sine of an angle.

sin(<number>)

3.4.6 `SQRT()` – Square root

Calculates the square root of a number.

sqrt(<number>)

If the number is negative an error will be thrown:

Cannot calculate square root with negative number -17

3.4.7 `TAN()` – Tangent

Calculates the tangent of an angle.

tan(<number>)

3.5 String Functions

3.5.1 `BIT_LENGTH()` – Bit Length

Return the bit length of a string.

```
bit_length(<string>)
```

3.5.2 `CHAR_LENGTH()` – Character Length

Return the character length of a string.

```
char_length(<string>)
```

3.5.3 `OCTET_LENGTH()` – Octet Length

Return the octet length of a string.

```
octet_length(<string>)
```

CHAPTER 4

Engine

4.1 Grouping

Grouping (`GROUP BY`).

4.1.1 Extract Expressions

Gather up all the aggregate columns in the query and make their expressions unique, for example:

```
SELECT value, sum(a), sum(a + 2), sum(a) * 2
FROM some_table
```

The unique aggregates would be (in no particular order):

```
sum(a)
sum(a + 2)
```

It is important they are unique because we don't want to calculate them twice. Even if we didn't care about the performance hit it is still critical they are unique because these expressions become the key as we aggregate and so multiple expressions would cause all values to be run twice through all the aggregate functions.

Each unique expression becomes a key and is rendered so `sum(x + 1)` is the same as `sum(x+1)`. However, `sum(x + 1)` and `sum(1 + x)` are not the same thing.

4.1.2 Grouping

As we iterate the records we maintain a Redis hash called `group` which contains keys that represent JSON strings and a value of *1*.

Since a query does not need to have a *GROUP BY* clause this effectively means that all the rows in the set belong to the same group. So we give this master group a `true` value to group on.

In any case the value is encoded to JSON - this ensures the value is both a string and unique across values and equivalent values in different data types.

The following records (grouped by `a`):

```
{"a": 123}
{"a": true}
{"a": "123"}
{"a": 123}
```

Would be:

```
"true"
"123"
"\"123\""
```

A query could contain multiple aggregate expressions and we want to keep them independent. Our two possible solutions is to maintain a different hash for each expression, or use a single hash but have a prefix/suffix to the keys. I chose to use the latter separated by a colon. The hash in Redis will look like this:

Key	Value
`count(*):"true"`	1
`count(*):"123"`	2
`count(*):"\"123\""`	1

The value is an integer that is incremented with each unique key. The number *123* appears twice.

4.1.3 Lua Processing

There are two Lua functions required to produce the final result. Lets take `AVG()` as an example. The two Lua functions would be:

```
function_avg(group, value)
function_avg_post(unique_group, group)
```

`function_avg` is run with each value as it's encountered. This is an opportunity to track values that may be needed for post processing:

```
function_avg('count(*)', 123)
function_avg('count(*)', true)
function_avg('count(*)', "123")
function_avg('count(*)', 123)
```

Once the grouping is complete we use a post processing Lua function to calculate the final result:

```
function_avg_post('agg_key', 'count(*)')
```

The first argument is the Redis key that contains the *original grouping hash* so you can lookup the original count if you need to.

4.1.4 Ensure Single Row

If there is no *GROUP BY* clause we must return one row, even if the original set did not have any rows. At the moment the default value will always be `0`, except in the case of `MIN()` and `MAX()` which is hard-coded to return `null`.

4.2 Indexing

An index is used to speed up lookup operations. You can expect that these work exactly the same as any other RDBMS.

Lets run through an example with a table called "mytable". The preceding number is the record ID which the index references and is non-linear on purpose so that we don't get confused later:

```
10. {"x": 123}
11. {"x": true}
15. {"x": "foo"}
17. {}
```

```
20. {"x": 123}
23. {"x": 57}
```

The index is created like:

```
CREATE INDEX myindex ON mytable (x)
```

Unlike most other database systems we need to index different data types in the same index. To achieve this we need to separate numbers from non-numbers and maintain a separate index for each respectively.

Lets look at the numbers first. The numbers index is sorted set that uses the score as the value (could be an integer or float) and value as the record ID:

```
57  -> "23"
123 -> "10"
123 -> "20"
```

For the non-numbers index which could contain NULLs, booleans, strings or even arrays and objects (more on arrays and objects later) we also use a sorted set but the roles are reversed with the score containing the record ID and the value is a string.

Now we run into the first minor problem. Redis requires all values in a sorted set to be unique. This is not an issue for the numbers index since the record ID is unique in all cases but is is a problem for the non-numbers index.

To get around this we add the record ID to the end of the string:

```
11 -> "T:11"
15 -> "Sfoo:15"
17 -> "N:17"
```

The first character is the type:

```
N null
T true
F false
S string (followed by the actual string)
```

The indexes are named with as `index:mytable:myindex:number` and `index:mytable:myindex:nonnumber`. The table name is kept in the key because an index can only apply to a single table.

Now that we know how indexes are stored we can request back records. The first and most important step is to determine if the value we are looking up is a number or a non-number then use the appropriate index.

In the case of a number we can use the Redis ZRANGEBYSCORE command, and for non-numbers we can use the ZRANGEBYLEX on the non-number index.

4.3 Tables

A table represents a permanent or temporary set of records. Table are not always public, they can also be intermediately steps during a query. This module provides ways to manipulate tables - abstracted away from Redis.

Tables are stored in Redis as a sorted set. Where the score is an integer representing the record ID. This presents two problems:

1. Scores in Redis are not required to be unique so it's possible to add a record with the same score making it very difficult to remove or manipulate a single record.
2. Values (the raw JSON records) must be unique in Redis. It is quite possible for records to contain the same data.

To get around both of those issues we set some basic rules for how tables are stored:

1. Each table has a separate associated key in Redis that acts as a incrementer to provide guaranteed unique scores for each row. This is incremented as it's needed. However, the table does not need to have a continuous set of IDs (such as when records are removed the IDs will also be removed forever.)
2. Each of the records includes a secret key that contains the record ID. For example the following record:

```
123 -> {"foo": "bar"}
```

   Is actually stored internally as:

```
123 -> {":id": 123, "foo": "bar"}
```

   This guarantees that the records are unique and also means that the scores do not need to be retrieved when records are iterated.

4.4 Transactions

4.4.1 Overview

A transaction is a block of work that will not be visible until a `COMMIT` has been issued. Or alternatively a `ROLLBACK` issued will completely undo any changes for the entire block.

Tesseract uses a MVCC (Multiversion Concurrency Control) methodology for record visibility. There is a lot of information on how MVCC works so for the purpose of the following example I will cover the important bits.

Most people that have used a database before expect "autocommit" where any statement that is not explicitly in a transaction is automatically committed. This is different to what the SQL standard states where `START TRANSACTION` is implicit but the `COMMIT` is not.

4.4.2 Internals

Following on, rather than surrounding every non-explicit transaction with a `START TRANSACTION` and `COMMIT` we define two states; in or not in a transaction.

Each connection maintains its current transaction ID. The `TransactionManager` maintains a set of all the transaction ID for all the connections that are explicitly in a transaction. Another way to look at it is when we say "in a transaction" we mean the current connection's transaction ID is in this set.

Each record has two special properties (amongst others) - prefixed with a colon:

- `xid`: The transaction ID when the record was created.
- `xex`: The transaction ID when the record was deleted. This will be `0` initially.

A record is visible only when all the following are true:

1. The `xid` is not in the active transactions.
2. The `xex` is `0` OR `xex` is not in the active transactions.

4.4.3 Collisions

One implicit limitation of transactions is that a connection must see the same database state, no matter how long it has been running or how many modifications to the databases have been made. But at the same time it must guarantee that

multiple transactions are editing the most recent version of rows (updating an expired row would have the changes lost).

At the moment tesseract handles collisions by simply throwing the error:

```
Transaction failed. Will ROLLBACK.
```

For the transaction that does not hold the lock for the row. This is crude solution as we should wait for that lock to be released but that's an improvement for another day.

4.4.4 Deadlocks

A deadlock should be impossible given that a transaction will immediately fail and `ROLLBACK`. When we have a scheduler for transactions then this will be a legitimate concern.

4.5 Server Protocol

The server protocol used by tesseract is pure JSON. This makes it very easy for any language or system to interact with - even `telnet` if you don't mind typing the JSON.

4.5.1 Connect

There is currently no authentication required for connecting. So simply knowing the host and port is sufficient for making a connection.

It is a plain TCP connection that normally runs on port 3679.

4.5.2 Request

The request is a JSON object:

```
{
  "sql": "SELECT 1 + 2"
}
```

4.5.3 Response

If the response is successful:

```
{
  "success": true,
  "data": [
    {
      "col1": 3
    }
  ]
}
```

If the response is an error:

```
{
  "success": false,
  "error": "Oh noes!"
}
```

Warnings are not considered a failure, so a prudent client should check to see if the `warnings` is present in the response:

```
{
  "success": true,
  "data": [
    {
      "col1": 3
    }
  ],
  "warnings": [
    "Take note..."
  ]
}
```

CHAPTER 5

Testing

5.1 Basic Test (*sql*)

Tesseract generates tests from YAML files. This makes it very easy to read, maintain and organise.

These files can be found in the `tests` directory. The most simple file may look like:

```
tests:
  my_test:
    sql: SELECT 1 + 2
    result:
    - {"col1": 3}
```

In the example above we have created one test called `my_test` that will run the SQL statement and confirm that the server returns one row containing that exact data.

5.1.1 Result (*result*)

Specify the expected output of the last `sql` statement. The data returned from the server must be exactly the same (including order) as the `result` items.

5.1.2 Result in Any Order (*result-unordered*)

If the order in which the records isn't important or is unpredictable you can use `result-unordered` instead of `result`.

```
tests:
  two_columns:
    data: table1
    sql: "SELECT foo, foo * 2 FROM table1"
    result-unordered:
    - {"foo": 123, "col2": 246}
    - {"foo": 124, "col2": 248}
    - {"foo": 125, "col2": 250}
```

5.1.3 Parser (*as*)

All tests that contain a `sql` attribute will be run through the parser and the statement will be rendered. This rendered statement is expected to be the same as this `sql` value. If you expect a different rendered string then you specify what

the result should be through `as`:

```
tests:
  alternate_operator:
    sql: 'SELECT null != 123'
    as: 'SELECT null <> 123'
    result:
    - {"col1": null}
```

5.1.4 Ignoring the Parser (*parse*)

Sometimes the SQL rendered from the SQL provided is not predictable, so we have to disable the parser test:

```
tests:
  json_object_with_two_elements:
    sql: 'SELECT {"foo": "bar", "baz": 123}'
    parse: false
    result:
    - {"col1": {"foo": "bar", "baz": 123}}
```

5.1.5 Commenting (*comment*)

Test can have an optional comment, this is preferred over using YAML inline comments so that comments can be injected is creating reports in the future.

```
tests:
  my_test:
    comment: Test everything!
    sql: 'SELECT 123'
```

This is also used at the root of the document to comment on the entire test suite like:

```
comment: |
  This file is responsible for stuff.

tests:
  my_test:
    comment: Test everything!
    sql: 'SELECT 123'
```

5.1.6 Tags (*tags*)

`tags` can be set at the file level which means that all tests in the file have the same tag:

```
comment: |
  All the tests are for 'foo'.

tags: foo

tests:
  ...
```

If you need multiple tags you can separate them with spaces:

```
tags: bar foo
```

It is not required, but it is good practice to keep the tags sorted alphabetically.

Tags are defined in `tags.yml`. While also not required for tags to be defined here is is good practice to leave a description.

5.1.7 Repeating Tests (*repeat*)

If a test lacks some predictability or you need to test the outcome multiple times for another reason you can use the `repeat`. This will still generate one test but it will loop through the `repeat` many times.

```
tests:
  my_test:
    sql: 'SELECT 123'
    repeat: 20
    result:
    - {"col1": 123}
```

5.2 Failures

5.2.1 Expecting Errors (*error*)

Use the `error` to test for an expected error:

```
tests:
  incompatible_types:
    sql: SELECT false AND 3.5
    error: No such operator boolean AND number.
```

Errors will be raised by the parser or by executing the SQL statement(s).

5.2.2 Expecting Warnings (*warning*)

You can assert one or more warnings are raised:

```
tests:
  json_object_duplicate_item_raises_warning:
    sql: 'SELECT {"foo": "bar", "foo": "baz"}'
    as: 'SELECT {"foo": "baz"}'
    warning: Duplicate key "foo", using last value.

  multiple_warnings_can_be_raised:
    sql: 'SELECT {"foo": "bar", "foo": "baz", "foo": "bax"}'
    as: 'SELECT {"foo": "bax"}'
    warning:
    - Duplicate key "foo", using last value.
    - Duplicate key "foo", using last value.
```

5.3 Data Sets (*data*)

It is common that you will want to test against an existing data fixture. Instead of inserting the data you need manually you can use fixtures:

```
data:
  table1:
  - {"foo": 125}
  - {"foo": 124}
  - {"foo": 123}

tests:
  where:
    data: table1
    sql: SELECT * FROM table1 WHERE foo = 124
    result:
    - {"foo": 124}
```

5.3.1 Randomizing Data (*data-randomized*)

For some tests you may want to randomize the order in which the records are loaded in. It is often used in conjunction with `repeat`.

```
data:
  table1:
  - {"foo": 125}
  - {"foo": 124}
  - {"foo": 123}
```

5.4 Verifying Notifications

When under test all notifications throughout the entire test case will be recorded. They can be asserted after all the SQL is executed. To test for a single notification:

```
tests:
  notification_will_be_fired_for_insert:
    sql:
    - CREATE NOTIFICATION foo ON some_table
    - 'INSERT INTO some_table {"a": "b"}'
    notification:
      to: foo
      with: {"a": "b"}
```

If you need to assert more than one notification:

```
tests:
  multiple_notifications_can_be_fired_from_a_single_select:
    sql:
    - CREATE NOTIFICATION foo1 ON some_table WHERE a = "b"
    - CREATE NOTIFICATION foo2 ON some_table WHERE a = "b"
    - 'INSERT INTO some_table {"a": "b"}'
    notification:
      - to: foo1
        with: {"a": "b"}
```

```
      - to: foo2
        with: {"a": "b"}
```

Or validate that no notifications have been fired:

```
tests:
  notification_will_respect_where_clause:
    sql:
    - CREATE NOTIFICATION foo ON some_table WHERE a = "c"
    - 'INSERT INTO some_table {"a": "b"}'
    notification: []
```

CHAPTER 6

Changelog

Each release is named after space stuff, with each release starting with a successive alphabet letter.

6.1 v1.0.0-alpha1 (Aurora)

This was the initial release and its goal was to get the most basic implementations of SELECT, INSERT, UPDATE and DELETE working.

1. SELECT statement supports single expression from a single table, with support for ORDER BY on a single column. [#6]
2. DELETE statement with `WHERE` clause. [#7]
3. UPDATE statement with `WHERE` clause. [#8]
4. INSERT statement.
5. CREATE NOTIFICATION and DROP NOTIFICATION. [#3]
6. Added logical operators: `AND`, `OR` and `NOT`.
7. Added containment operators: `BETWEEN` and `NOT BETWEEN`.
8. Added mathematical operators: `+`, `-`, `*`, `/`, `^` and `%`.
9. Added comparison operators: `=`, `<>`, `<`, `>`, `<=`, `>=` and alias `!=`.
10. Added set membership operators: `IN` and `NOT IN`.
11. Added type checking operators: `IS` and `IS NOT`. [#2]
12. Added pattern matching operators: `LIKE` and `NOT LIKE`. [#1]
13. Added functions: `ABS()`, `BIT_LENGTH()`, `CEIL()`, `CHAR_LENGTH()`, `FLOOR()`, `OCTET_LENGTH()`. [#4]

6.2 v1.0.0-alpha2 (Binary Star)

The focus of this release was on GROUP BY and aggregate functions.

1. GROUP BY single column for SELECT. [#10]
2. Added aggregate functions AVG(), COUNT(), MAX(), MIN() and SUM(). [#10]
3. Added string concatenation (`||`) operator. [#9]

4. Added ILIKE and NOT ILIKE operators. [#11]
5. Reformatted documentation to work with Rippledoc. [#11]

6.3 v1.0.0-alpha3 (Comet)

This releases theme was *indexing*. At the moment only exact lookups are supported.

1. Added support for LIMIT and OFFSET. [#12]
2. Added support for EXPLAIN on SELECT queries. [#14]
3. Added CREATE INDEX and DROP INDEX. [#15]
4. Added `COS()`. [#13]
5. Added `SIN()`. [#13]
6. Added `SQRT()`. [#13]
7. Added `TAN()`. [#13]
8. The query planner now understands some basic impossible `WHERE` clauses.

6.4 v1.0.0-alpha4 (Dark Matter)

The focus of this release was on supporting *transactions*.

1. Support for START TRANSACTION, COMMIT and ROLLBACK. [#16] [#17]
2. Refactored documentation for Sphinx, hosted on tesseractdb.readthedocs.org.

CHAPTER 7

FAQ

7.1 Really? Another SQL Database?

Yes, but not really. The SQL language was designed to work within a relational database so not all of the SQL standard makes sense to a document store. However, this hasn't stopped popular SQL vendors from implementing their own extensions to the standard that allow the database to be used like this.

7.2 I Thought SQL Was Dead?

That is a discussion greater than one product like tesseract can answer. I will offer my perspective on the issue since it shapes the goals and development of tesseract.

There is a lot of knowledge in the SQL language and framework that should not be dismissed simply because "it is old" or that it carries the stigma of only being usful in a relational model.

Over recent years there has been a fashionable snobbery to existing database systems in favour of the NoSQL solutions. If you stop and think about it the only thing categorising these databases is that they do not use the SQL language (understandable) but there has been no strides into a text based abstract language. This is less about portability - it's very unlikely that people change database products - and more about being about to use a product easily and reliably from the start.

7.3 What Are the Goals of Tesseract?

Based on the discussion above.

1. Follow as much of the SQL standard that makes sense.
2. Provide SQL language extensions to make it easier dealing with objects and nested data.
3. KISS.

CHAPTER 8

Appendix

8.1 Formatting

The tesseract manual is written in reStructuredText and processed with Sphinx. This document does not intend to outline how the documentation is generated. Only to provide a style guidelines and samples that are used through the rest of the manual.

There are some general rules to follow:

- Keep all documentation to 80 columns unless it cannot be avoided (for example, long tables or code snippets).
- The documentation here is intended to be easily read and maintained. Do not use fancy features that make the reStructuredText less readable simply to make the result HTML look more pretty.

8.1.1 Basic

Always H1 for first level headings and always use an underlined heading rather than hashes:

```
Heading 1
=========

Heading 2
---------

Heading 3
^^^^^^^^^
```

Each heading of any strength should have two blank lines above it (unless it is the first line of the file.)

8.1.2 SQL Examples

SQL examples should be used in fenced blocks and not include the following semi-colon:

```
.. code-block:: sql

   SELECT foo FROM bar
```

```
SELECT foo FROM bar
```

If you would like to show the result of the SQL it must be in a separate block for JSON:

```
.. code-block:: json

   {"foo": "bar"}
```

```
{"foo": "bar"}
```

8.1.3 Syntax Descriptions

When describing SQL syntax use a SQL block:

```
.. code-block:: sql

   SELECT <some_columns>
   FROM <table>
```

```
SELECT <some_columns>
FROM <table>
```

For each of the placeholders use a definition style with a blank line between each definition:

```
some_columns
  Lots of nice description here.

table
  Even more wonderful information!
```

some_columns Lots of nice description here.

table Even more wonderful information!

8.1.4 Notes

Notes are meant to stand out from other text and contain important information.

```
.. highlights::

This is important information.
```

> This is important information.

8.1.5 Tables

There are two types of table syntax that make the table as small as possible around the text or type to span as much space as possible. Always use the greater span syntax:

```
.. table::

   =========  =============
   full span  syntax
   =========  =============
   for        tables
   with       multiple rows
   =========  =============
```

full span	syntax
for	tables
with	multiple rows

t

T