sqlwhat Documentation

Release 3.8.1

DataCamp

Jan 14, 2020

For an introduction to SCTs and how they use sqlwhat, visit the README.

This documentation features:

- A glossary with typical use-cases and corresponding SCT constructs.
- Reference documentation of all actively maintained sqlwhat functions.
- Some articles that gradually expose of sqlwhat's functionality and best practices.

If you are new to writing SCTs for SQL exercises, start with the tutorial. The glossary is good to get a quick overview of how all functions play together after you have a basic understanding. The reference docs become useful when you grasp all concepts and want to look up details on how to call certain functions and specify custom feedback messages.

If you find yourself writing SCTs for more advanced SQL queries, you'll probably venture into SCT functions that verify the AST representation of SQL queries. When you do, make sure to keep the AST viewer open; it is a great tool to understand how different SQL statements are parsed and how to write your SCTs.

CHAPTER 1

Glossary

This article lists some example solutions. For each of these solutions, an SCT is included, as well as some example student submissions that would pass and fail. In all of these, a submission that is identical to the solution will evidently pass.

Note: These SCT examples are not golden bullets that are perfect for your situation. Depending on the exercise, you may want to focus on certain parts of a statement, or be more accepting for different alternative answers.

All of these examples come from the Intro to SQL for Data Science and Joining data in PostgreSQL courses. You can have a look at their respective GitHub sources:

- https://github.com/datacamp/courses-intro-to-SQL (public)
- https://github.com/datacamp/courses-joining-data-in-postgresql (DataCamp-only).

1.1 Selecting a column

```
-- solution query
SELECT title FROM films;
```

```
# sct
Ex().check_correct(
    check_column('title').has_equal_value(),
    check_node('SelectStmt').multi(
        check_edge('target_list', 0).has_equal_ast(),
        check_edge('from_clause').has_equal_ast()
    )
)
```

```
-- submissions that pass
SELECT title FROM films
```

(continues on next page)

(continued from previous page)

```
SELECT id, title FROM films
SELECT id, title FROM films ORDER by id

-- submissions that fail
SELECT id FROM films
SELECT id FROM films LIMIT 5
```

1.2 Star argument

```
-- solution query
SELECT * FROM films;
```

```
# sct
Ex().check_correct(
    check_all_columns().has_equal_value(),
    check_node('SelectStmt').multi(
        check_node('Star', missing_msg="Are you using `SELECT *` to select _all_
↪columns?"),
        check_edge('from_clause').has_equal_ast()
    )
)
```

```
-- submissions that pass
SELECT * FROM films
SELECT id, title, <all_other_cols>, year FROM films
SELECT * FROM films ORDER by year

-- submissions that fail
SELECT id FROM films
SELECT id, title FROM films
SELECT * FROM films LIMIT 5
```

1.3 COUNT(*)

```
-- solution query
SELECT COUNT(*) FROM films;
```

```
# sct
Ex().check_correct(
    check_column('count').has_equal_value(),
    check_node('SelectStmt').multi(
        check_node('Call').multi(
            check_edge('name').has_equal_ast(),
            check_edge('args').has_equal_ast()
        ),
        check_edge('from_clause').has_equal_ast()
    )
)
```

```
-- submissions that pass
SELECT COUNT(*) FROM films
SELECT COUNT(id) FROM films

-- submissions that fail
SELECT * FROM films
SELECT COUNT(*) FROM films WHERE id < 100
```

1.4 WHERE clause

```
-- solution query
SELECT name, birthdate
FROM people
WHERE birthdate = '1974-11-11';
```

```
# First check if the WHERE clause was correct
Ex().check_correct(
    has_nrows(),
    check_node('SelectStmt').multi(
        check_edge('from_clause').has_equal_ast(),
        check_edge('where_clause').has_equal_ast()
    )
)

# Next check if right columns were included
Ex().check_correct(
    check_all_columns().has_equal_value(),
    check_node('SelectStmt').check_edge('target_list').check_or(
        has_equal_ast(),
        has_equal_ast(sql = "birthdate, name")
    )
)
```

1.5 ORDER BY

```
SELECT name
FROM people
ORDER BY name;
```

```
# Check whether the right column was included
Ex().check_column('name')

Ex().check_correct(
    # Check whether the name column is correct (taking into account order)
    check_column('name').has_equal_value(ordered=True),
    check_node('SelectStmt').multi(
        check_edge('from_clause').has_equal_ast(),
        check_edge('order_by_clause').has_equal_ast()
    )
)
```

1.6 Joins

```
SELECT *
FROM cities
INNER JOIN countries
ON cities.country_code = countries.code;
```

```
# First check if the joining went well (through checking the number of rows)
Ex().check_correct(
    has_nrows(),
    check_node('SelectStmt').check_edge('from_clause').multi(
        check_edge('join_type').has_equal_ast(),
        check_edge('left').has_equal_ast(),
        check_edge('right').has_equal_ast(),
        check_edge('cond').check_or(
            has_equal_ast(),
            # the other way around should also work
            has_equal_ast(sql = 'countries.code = cities.country_code')
        )
    )
)

# Check if all columns are included and correct
Ex().check_correct(
    check_all_columns().has_equal_value(),
    check_node('SelectStmt').check_node('Star')
)
```

CHAPTER 2

Result checks

allow_error(*state*)

Allow submission to pass, even if it originally caused a database error.

This function has been renamed.

Use `Ex().allow_errors()` in your SCT if the intent of the exercise is to generate an error.

check_all_columns(*state*, *allow_extra=True*, *too_many_cols_msg=None*, *expand_msg=None*)

Zoom in on the columns that are specified by the solution

Behind the scenes, this is using `check_column()` for every column that is in the solution query result. Afterwards, it's selecting only these columns from the student query result and stores them in a child state that is returned, so you can use `has_equal_value()` on it.

This function does not allow you to customize the messages for `check_column()`. If you want to manually set those, simply use `check_column()` explicitly.

Parameters

- **allow_extra** – True by default, this determines whether students are allowed to have included other columns in their query result.
- **too_many_cols_msg** – If specified, this overrides the automatically generated feedback message in case `allow_extra` is False and the student's query returned extra columns when comparing the so the solution query result.
- **expand_msg** – if specified, this overrides the automatically generated feedback message that is prepended to feedback messages that are thrown further in the SCT chain.

Example Consider the following solution and SCT:

```
# solution
SELECT artist_id as id, name FROM artists

# sct
Ex().check_all_columns()
```

(continues on next page)

(continued from previous page)

```
# passing submission
SELECT artist_id as id, name FROM artists

# failing submission (wrong names)
SELECT artist_id, name FROM artists

# passing submission (allow_extra is True by default)
SELECT artist_id as id, name, label FROM artists
```

check_column (*state*, *name*, *missing_msg=None*, *expand_msg=None*)

Zoom in on a particular column in the query result, by name.

After zooming in on a column, which is represented as a single-column query result, you can use `has_equal_value()` to verify whether the column in the solution query result matches the column in student query result.

Parameters

- **name** – name of the column to zoom in on.
- **missing_msg** – if specified, this overrides the automatically generated feedback message in case the column is missing in the student query result.
- **expand_msg** – if specified, this overrides the automatically generated feedback message that is prepended to feedback messages that are thrown further in the SCT chain.

Example Suppose we are testing the following SELECT statements

- solution: `SELECT artist_id as id, name FROM artists`
- student : `SELECT artist_id, name FROM artists`

We can write the following SCTs:

```
# fails, since no column named id in student result
Ex().check_column('id')

# passes, since a column named name is in student_result
Ex().check_column('name')
```

check_query (*state*, *query*, *error_msg=None*, *expand_msg=None*)

Run arbitrary queries against to the DB connection to verify the database state.

For queries that do not return any output (INSERTs, UPDATEs, . . .), you cannot use functions like `check_col()` and `has_equal_value()` to verify the query result.

`check_query()` will rerun the solution query in the transaction prepared by sqlbackend, and immediately afterwards run the query specified in `query`.

Next, it will also run this query after rerunning the student query in a transaction.

Finally, it produces a child state with these results, that you can then chain off of with functions like `check_column()` and `has_equal_value()`.

Parameters

- **query** – A SQL query as a string that is executed after the student query is re-executed.
- **error_msg** – if specified, this overrides the automatically generated feedback message in case the query generated an error.

- **expand_msg** – if specified, this overrides the automatically generated feedback message that is prepended to feedback messages that are thrown further in the SCT chain.

Example Suppose we are checking whether an INSERT happened correctly:

```
INSERT INTO company VALUES (2, 'filip', 28, 'sql-lane', 42)
```

We can write the following SCT:

```
Ex().check_query('SELECT COUNT(*) AS c FROM company').has_equal_
↪value()
```

check_result (*state*)

High level function which wraps other SCTs for checking results.

`check_result()`

- uses `lowercase()`, then
- runs `check_all_columns()` on the state produced by `lowercase()`, then
- runs `has_equal_value` on the state produced by `check_all_columns()`.

check_row (*state*, *index*, *missing_msg=None*, *expand_msg=None*)

Zoom in on a particular row in the query result, by index.

After zooming in on a row, which is represented as a single-row query result, you can use `has_equal_value()` to verify whether all columns in the zoomed in solution query result have a match in the student query result.

Parameters

- **index** – index of the row to zoom in on (zero-based indexed).
- **missing_msg** – if specified, this overrides the automatically generated feedback message in case the row is missing in the student query result.
- **expand_msg** – if specified, this overrides the automatically generated feedback message that is prepended to feedback messages that are thrown further in the SCT chain.

Example Suppose we are testing the following SELECT statements

- solution: `SELECT artist_id as id, name FROM artists LIMIT 5`
- student : `SELECT artist_id, name FROM artists LIMIT 2`

We can write the following SCTs:

```
# fails, since row 3 at index 2 is not in the student result
Ex().check_row(2)

# passes, since row 2 at index 1 is in the student result
Ex().check_row(0)
```

lowercase (*state*)

Convert all column names to their lower case versions to improve robustness

Example Suppose we are testing the following SELECT statements

- solution: `SELECT artist_id as id FROM artists`
- student : `SELECT artist_id as ID FROM artists`

We can write the following SCTs:

```
# fails, as id and ID have different case
Ex().check_column('id').has_equal_value()

# passes, as lowercase() is being used
Ex().lowercase().check_column('id').has_equal_value()
```

has_equal_value(*state*, *ordered=False*, *ndigits=None*, *incorrect_msg=None*)

Verify if a student and solution query result match up.

This function must always be used after 'zooming' in on certain columns or records (check_column, check_row or check_result). `has_equal_value` then goes over all columns that are still left in the solution query result, and compares each column with the corresponding column in the student query result.

Parameters

- **ordered** – if set to False, the default, all rows are sorted (according to the first column and the following columns as tie breakers). if set to True, the order of rows in student and solution query have to match.
- **ndigits** – if specified, number of decimals to use when comparing column values.
- **incorrect_msg** – if specified, this overrides the automatically generated feedback message in case a column in the student query result does not match a column in the solution query result.

Example Suppose we are testing the following SELECT statements

- solution: `SELECT artist_id as id, name FROM artists ORDER BY name`
- student : `SELECT artist_id, name FROM artists`

We can write the following SCTs:

```
# passes, as order is not important by default
Ex().check_column('name').has_equal_value()

# fails, as order is deemed important
Ex().check_column('name').has_equal_value(ordered=True)

# check_column fails, as id is not in the student query result
Ex().check_column('id').has_equal_value()

# check_all_columns fails, as id not in the student query result
Ex().check_all_columns().has_equal_value()
```

has_ncols(*state*, *incorrect_msg="Your query returned a table with {{n_stu}} column{{'s' if n_stu > 1 else ''}} while it should return a table with {{n_sol}} column{{'s' if n_sol > 1 else ''}}."*)

Test whether the student and solution query results have equal numbers of columns.

Parameters **incorrect_msg** – If specified, this overrides the automatically generated feedback message in case the number of columns in the student and solution query don't match.

Example Consider the following solution and SCT:

```
# solution
SELECT artist_id as id, name FROM artists

# sct
Ex().has_ncols()
```

(continues on next page)

(continued from previous page)

```
# passing submission
SELECT artist_id as id, name FROM artists

# failing submission (too little columns)
SELECT artist_id as id FROM artists

# passing submission (two columns, even though not correct ones)
SELECT artist_id, label FROM artists
```

has_no_error (*state*, *incorrect_msg='Your code generated an error. Fix it and try again!'*)

Check whether the submission did not generate a runtime error.

Simply use `Ex().has_no_error()` in your SCT whenever you want to check for errors. By default, after the entire SCT finished executing, `sqlwhat` will check for errors before marking the exercise as correct. You can disable this behavior by using `Ex().allow_error()`.

Parameters **incorrect_msg** – If specified, this overrides the automatically generated feedback message in case the student's query did not return a result.

has_nrows (*state*, *incorrect_msg="Your query returned a table with {{n_stu}} row{{'s' if n_stu > 1 else ''}} while it should return a table with {{n_sol}} row{{'s' if n_sol > 1 else ''}}."*)

Test whether the student and solution query results have equal numbers of rows.

Parameters **incorrect_msg** – If specified, this overrides the automatically generated feedback message in case the number of rows in the student and solution query don't match.

has_result (*state*, *incorrect_msg='Your query did not return a result.'*)

Checks if the student's query returned a result.

Parameters **incorrect_msg** – If specified, this overrides the automatically generated feedback message in case the student's query did not return a result.

CHAPTER 3

AST Checks

check_edge(*state, name, index=0, missing_msg='Check the {ast_path}. Could not find the {index}{field_name}.'*)
Select an attribute from an abstract syntax tree (AST) node, using the attribute name.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **name** – the name of the attribute to select from current AST node.
- **index** – entry to get from a list field. If too few entires, will fail with missing_msg.
- **missing_msg** – feedback message if attribute is not in student AST.

Example If both the student and solution code are..

```
SELECT a FROM b; SELECT x FROM y;
```

then we can get the from_clause using

```
# approach 1: with manually created State instance -----
state = State(*args, **kwargs)
select = check_node(state, 'SelectStmt', 0)
clause = check_edge(select, 'from_clause')

# approach 2: with Ex and chaining ---------------------
select = Ex().check_node('SelectStmt', 0)           # get first␣
↪select statement
clause =  select.check_edge('from_clause', None)    # get from_clause␣
↪(a list)
clause2 = select.check_edge('from_clause', 0)       # get first entry␣
↪in from_clause
```

check_node(*state, name, index=0, missing_msg='Check the {ast_path}. Could not find the {index}{node_name}.', priority=None*)
Select a node from abstract syntax tree (AST), using its name and index position.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **name** – the name of the abstract syntax tree node to find.
- **index** – the position of that node (see below for details).
- **missing_msg** – feedback message if node is not in student AST.
- **priority** – the priority level of the node being searched for. This determines whether to descend into other AST nodes during the search. Higher priority nodes descend into lower priority. Currently, the only important part of priority is that setting a very high priority (e.g. 99) will search every node.

Example If both the student and solution code are..

```
SELECT a FROM b; SELECT x FROM y;
```

then we can focus on the first select with:

```
# approach 1: with manually created State instance
state = State(*args, **kwargs)
new_state = check_node(state, 'SelectStmt', 0)

# approach 2:  with Ex and chaining
new_state = Ex().check_node('SelectStmt', 0)
```

has_code(*state*, *text*, *incorrect_msg='Check the {ast_path}. The checker expected to find {text}.'*, *fixed=False*)

Test whether the student code contains text.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **text** – text that student code must contain. Can be a regex pattern or a simple string.
- **incorrect_msg** – feedback message if text is not in student code.
- **fixed** – whether to match text exactly, rather than using regular expressions.

Note: Functions like `check_node` focus on certain parts of code. Using these functions followed by `has_code` will only look in the code being focused on.

Example If the student code is..

```
SELECT a FROM b WHERE id < 100
```

Then the first test below would (unfortunately) pass, but the second would fail..:

```
# contained in student code
Ex().has_code(text="id < 10")

# the $ means that you are matching the end of a line
Ex().has_code(text="id < 10$")
```

By setting `fixed = True`, you can search for fixed strings:

```
# without fixed = True, '*' matches any character
Ex().has_code(text="SELECT * FROM b")                   # passes
Ex().has_code(text="SELECT \\* FROM b")                 # fails
Ex().has_code(text="SELECT * FROM b", fixed=True)       # fails
```

You can check only the code corresponding to the WHERE clause, using

```
where = Ex().check_node('SelectStmt', 0).check_edge('where_clause')
where.has_code(text = "id < 10)
```

`has_equal_ast` (*state*, *incorrect_msg='Check the {ast_path}. {extra}'*, *sql=None*, *start='expression'*, *exact=None*)

Test whether the student and solution code have identical AST representations

Parameters

- **`state`** – State instance describing student and solution code. Can be omitted if used with Ex().
- **`incorrect_msg`** – feedback message if student and solution ASTs don't match
- **`sql`** – optional code to use instead of the solution ast that is zoomed in on.
- **`start`** – if `sql` arg is used, the parser rule to parse the sql code. One of 'expression' (the default), 'subquery', or 'sql_script'.
- **`exact`** – whether to require an exact match (True), or only that the student AST contains the solution AST. If not specified, this defaults to `True` if `sql` is not specified, and to `False` if `sql` is specified. You can always specify it manually.

Example Example 1 - Suppose the solution code is

```
SELECT * FROM cities
```

and you want to verify whether the *FROM* part is correct:

```
Ex().check_node('SelectStmt').from_clause().has_equal_ast()
```

Example 2 - Suppose the solution code is

```
SELECT * FROM b WHERE id > 1 AND name = 'filip'
```

Then the following SCT makes sure `id > 1` was used somewhere in the WHERE clause.:

```
Ex().check_node('SelectStmt') \/
    .check_edge('where_clause') \/
    .has_equal_ast(sql = 'id > 1')
```

CHAPTER 4

Logic

classmethod Ex.**__call__**(*state=None*)

Returns the current code state as a Chain instance.

This allows SCTs to be run without including their 1st argument, `state`.

When writing SCTs on DataCamp, no State argument to `Ex` is necessary. The exercise State is built for you.

Parameters **state** – a State instance, which contains the student/solution code and results.

Example code

```
# How to write SCT on DataCamp.com
Ex().has_code(text="SELECT id")

# Experiment locally - chain off of Ex(), created from state
state = SomeStateProducingFunction()
Ex(state).has_code(text="SELECT id")

# Experiment locally - no Ex(), create and pass state explicitly
state = SomeStateProducingFunction()
has_code(state, text="SELECT id")
```

check_correct(*state*, *check*, *diagnose*)

Allows feedback from a diagnostic SCT, only if a check SCT fails.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **check** – An sct chain that must succeed.
- **diagnose** – An sct chain to run if the check fails.

Example The SCT below tests whether students query result is correct, before running diagnostic SCTs..

```
Ex().check_correct(
    check_result(),
    check_node('SelectStmt')
)
```

check_not (*state*, **tests*, *msg*)

Run multiple subtests that should fail. If all subtests fail, returns original state (for chaining)

- This function is currently only tested in working with `has_code()` in the subtests.
- This function can be thought as a `NOT(x OR y OR ...)` statement, since all tests it runs must fail
- This function can be considered a direct counterpart of multi.

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()
- ***tests** – one or more sub-SCTs to run
- **msg** – feedback message that is shown in case not all tests specified in `*tests` fail.

Example Thh SCT below runs two has_code cases..

```
Ex().check_not(
    has_code('INNER'),
    has_code('OUTER'),
    incorrect_msg="Don't use `INNER` or `OUTER`!"
)
```

If students use `INNER (JOIN)` or `OUTER (JOIN)` in their code, this test will fail.

check_or (*state*, **tests*)

Test whether at least one SCT passes.

If all of the tests fail, the feedback of the first test will be presented to the student.

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()
- **tests** – one or more sub-SCTs to run

Example The SCT below tests that the student typed either 'SELECT' or 'WHERE' (or both)..

```
Ex().check_or(
    has_code('SELECT'),
    has_code('WHERE')
)
```

The SCT below checks that a SELECT statement has at least a WHERE c or LIMIT clause..

```
Ex().check_node('SelectStmt', 0).check_or(
    check_edge('where_clause'),
    check_edge('limit_clause')
)
```

disable_highlighting (*state*)

Disable highlighting in the remainder of the SCT chain.

Include this function if you want to avoid that pythonwhat marks which part of the student submission is incorrect.

fail(*state*, *msg='fail'*)

Always fails the SCT, with an optional msg.

This function takes a single argument, `msg`, that is the feedback given to the student. Note that this would be a terrible idea for grading submissions, but may be useful while writing SCTs. For example, failing a test will highlight the code as if the previous test/check had failed.

multi(*state*, **tests*)

Run multiple subtests. Return original state (for chaining).

This function could be thought as an AND statement, since all tests it runs must pass

Parameters

- **state** – State instance describing student and solution code, can be omitted if used with Ex()
- **tests** – one or more sub-SCTs to run.

Example The SCT below checks two has_code cases..

```
Ex().multi(has_code('SELECT'), has_code('WHERE'))
```

The SCT below uses `multi` to 'branch out' to check that the SELECT statement has both a WHERE and LIMIT clause..

```
Ex().check_node('SelectStmt', 0).multi(
    check_edge('where_clause'),
    check_edge('limit_clause')
)
```

CHAPTER 5

Electives

allow_errors (*state*)

Allow running the student code to generate errors.

This has to be used only once for every time code is executed or a different xwhat library is used. In most exercises that means it should be used just once.

Example The following SCT allows the student code to generate errors:

```
Ex().allow_errors()
```

has_chosen (*state*, *correct*, *msgs*)

Verify exercises of the type MultipleChoiceExercise

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **correct** – index of correct option, where 1 is the first option.
- **msgs** – list of feedback messages corresponding to each option.

Example The following SCT is for a multiple choice exercise with 2 options, the first of which is correct.:

```
Ex().has_chosen(1, ['Correct!', 'Incorrect. Try again!'])
```

success_msg (*state*, *msg*)

Changes the success message to display if submission passes.

Parameters

- **state** – State instance describing student and solution code. Can be omitted if used with Ex().
- **msg** – feedback message if student and solution ASTs don't match

Example The following SCT changes the success message:

```
Ex().success_msg("You did it!")
```

CHAPTER 6

Tutorial

sqlwhat uses the `.` to 'chain together' SCT functions. Every chain starts with the `Ex()` function call, which holds the exercise state. This exercise state contains all the information that is required to check if an exercise is correct, which are:

- the student submission and the solution as text, and their corresponding parse trees.
- the result of running the solution query, as a dictionary of columns.
- the result of running the student's query, as a dictionary of columns
- the errors that running the student's query generated, if any.

As SCT functions are chained together with `.`, the `Ex()` exercise state is copied and adapted into 'sub states' to zoom in on particular parts of the state. Before this theory blows your brains out, let's have a look at some basic examples.

6.1 Example 1: query result

Assume we want to robustly check whether a student correctly selected the columns `title` from the table `films`:

```
SELECT title FROM films
```

The following SCT would verify this:

```
Ex().check_column('title').has_equal_value()
```

Let's see what happens when the SCT runs:

- `Ex()` returns the 'root state', which contains a reference to the student query, the solution query, the result of running the student's query, and the result of running the solution query.
- `check_column('title')` looks at the result of running the student's query and the solution query, and verifies that they are there.
  - If the student had submitted `SELECT title, year FROM films`, the query result will contain the column `title`, and the function will not fail.

    - If the student had submitted `SELECT year FROM films`, the query result will not contain the column `title`, the function will fail, and sqlwhat will automatically generate a meaningful feedback message.

  If `check_column('title')` runs without errors, it will return a sub-state of the root-state returned by `Ex()`, that zooms in solely on the `title` column of the query result. This means that `Ex().check_column('title')` will produce the same substate for the following student submissions:

  ```
  SELECT title FROM films
  SELECT title, year FROM films
  SELECT * FROM films
  ```

- `has_equal_value()` will consider the state produced by `check_column('title')` and verify whether the contents of the columns match between student and solution query result.
    - If the student had submitted `SELECT title FROM films`, the query result of the student submission an the solution will exactly match, and the function will pass.
    - If the student had submitted `SELECT title, year FROM films`, the `title` column of the student's query result and the solution query result will still exactly match, and the function will pass. Note: there are ways to be more restrictive about this, and making sure that the SCT fails if additonal columns were returned.
    - If the student had submitted `SELECT title FROM films LIMIT 5`, the query result of the student and solution submission will not match the `title` column in the student query result. The function will fail, and slqhwat will automatically genearte a meaningful feedback message.

6.2 Example 2: AST-based checks

As another example, suppose we want to robustly check whether a student correctly selected the `id` column from an `artists` table with a certain condition:

```
SELECT id FROM artists WHERE id > 100
```

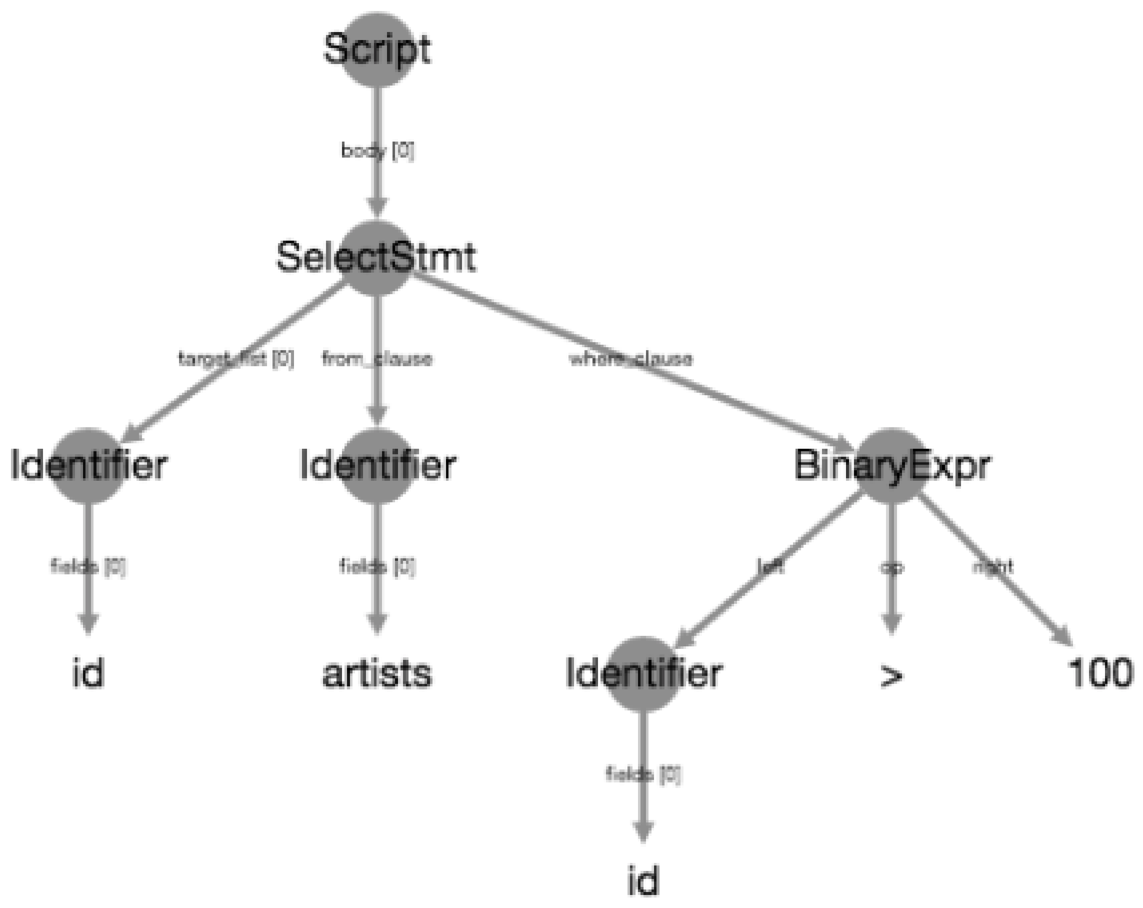
In line with Example 1, the following SCT would robustly verify the result of the query:

```
Ex().check_column('id').has_equal_value()
```

If the student makes a mistake, this SCT will produce a feedback message in the direction of: "Check the result of your query. Column `id` seems to be incorrect.". Although this is valid feedback, you'll typically want to be more targeted about the actual mistake in the code of the student. This is where the AST-based checks come in.

Where functions such as `check_column()` and `has_equal_value()` look at the *result* of your query, AST-based checks will look at the query *itself*. More specifically, they will look at the Abstract Syntax Tree representation of the query, which enables us to robustly look and check for certain patterns and constructs.

To explore the AST representation of a SQL query, visit the AST viewer. The AST for the `SELECT` statement above is:



Notice how the statement is neatly chopped up into its consituents: the `SELECT` statement is chopped up into three parts: the `target_list` (which columns to select), the `from_clause` (from which table to select) and the `where_clause` (the condition that has to be satisfied). Next, the `where_caluse` is a `BinaryExpr` that is further chopped up.

Similar to how `check_column('title')` zoomed in on only the `title` column of the student's and solution query result, you can use the `.` operator to chain together AST-verifying SCT functions that each zoom in on particular parts of the student's submission and the solution.

Suppose you want to check whether students have correctly specified the table from which to select columns (the `FROM artists` part). This SCT script does that:

```
Ex().check_node("SelectStmt").check_edge("from_clause").has_equal_ast()
```

We'll now explain step by step what happens when a student submits the following (incorrect) code:

```
SELECT id FROM producers WHERE id > 100
```

When the SCT executes:

- `Ex()` runs first, and fetches the root state that considers the entire student submission and solution:

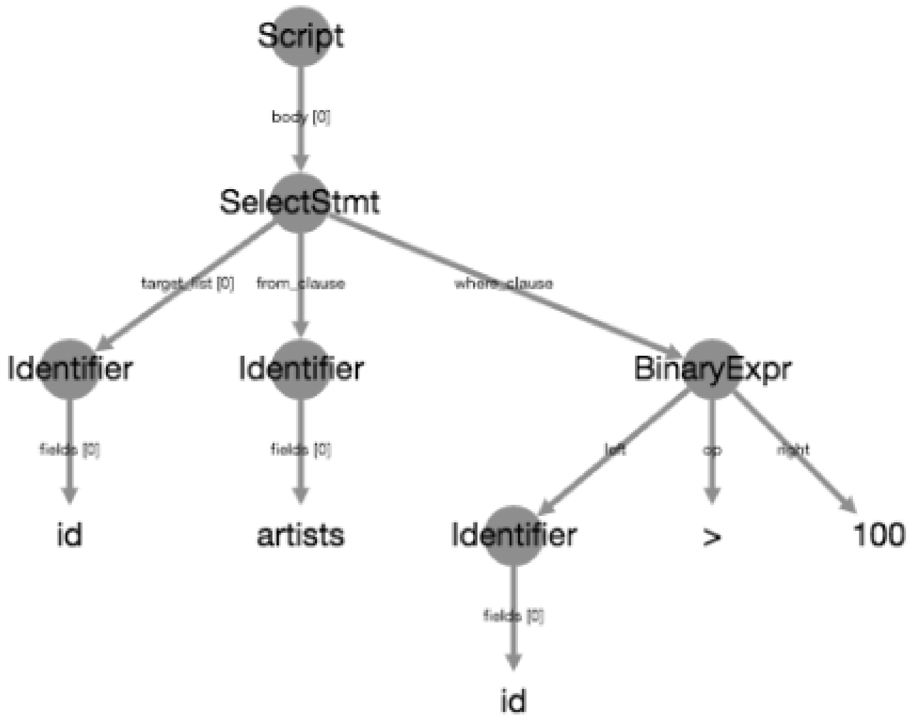
  ```
  -- solution
  SELECT id FROM artists WHERE id > 100

  ```

(continues on next page)

(continued from previous page)

```
-- student
SELECT id FROM producers WHERE id > 100
```

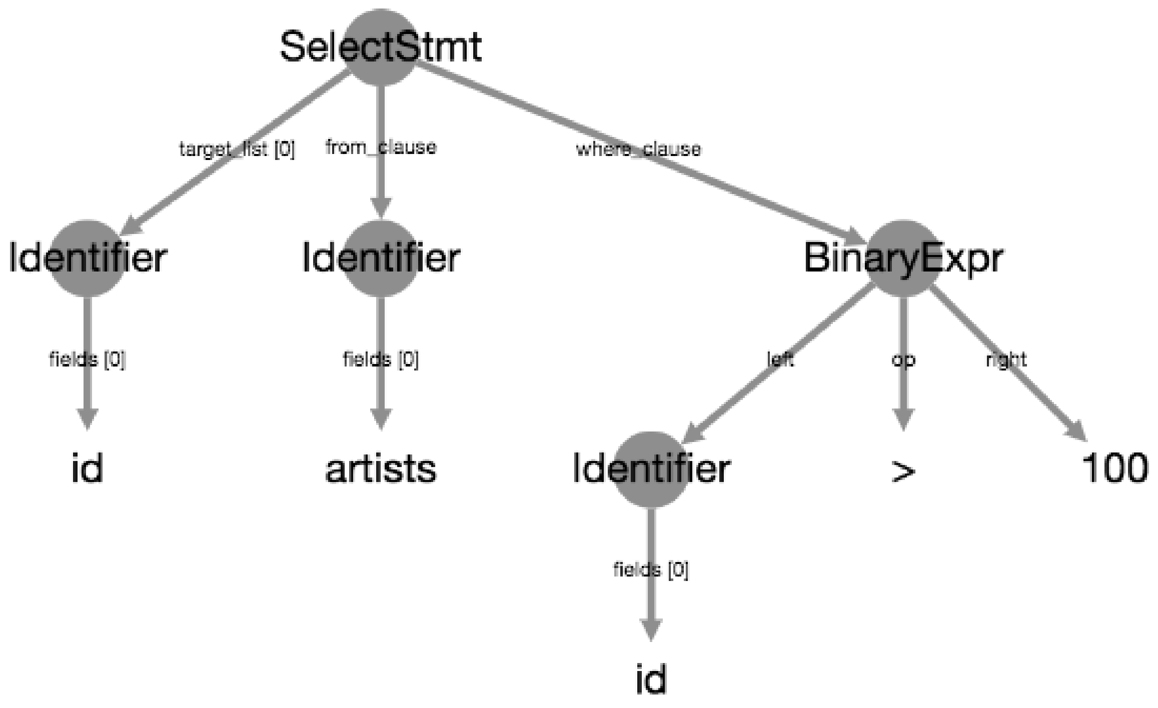
This is the corresponding AST of the solution. This is the same tree as included earlier in this article. The AST for the student submission will look very similar.



- Next, `check_node()` chains off of the state produced by `Ex()` and produces a child state that focuses on the `SelectStmt` portion of the submission and solution:

```
-- solution
SELECT id FROM artists WHERE id > 100

-- student
SELECT id FROM producers WHERE id > 100
```

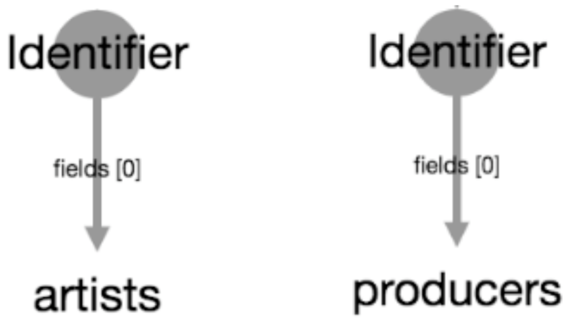
The corresponding AST of the solution is the following. Notice that although the textual representation is the same as `Ex()`, the AST representation no longer includes the `Script` node. The AST for the student submission will look very similar.



- Next, `check_edge()` chains off of the state produced by `check_node()` and zooms in on the `from_clause` branch of the AST:

```
-- solution
artists

-- student
producers
```

The corresponding ASTs for solution and student are as follows:



- Finally, `has_equal_ast()` chains off of the state produced by `check_edge()` and checks whether the student submission and solution sub-ASTs correspond. As the solution expects `artists` while the student specified `producers` the SCT fails and sqlwhat will generate a meaningful feedback message.

Note:

- `check_node()` is used to select a *node* of the AST tree (a circle in the image).
- `check_edge()` is used to walk down a *branch* of the AST tree (a line in the image).

6.3 Example 3: Combining result checks and AST checks

In general, you want to write flexible SCTs: if students get the end result right, you don't want to be picky about how they got there. However, when they do make a mistake, you want to be specific about the mistake they are making. In other words, a good SCT is robust against different ways of solving a problem, but specific when something's wrong.

Example 1 looked at the result of a query, which are perfect to match the flexibility requirement. Example 2 looks at the AST representation of the code, which is good to dig deeper into the code a student submitted to see what's going on.

With `check_correct()`, you can effectively combine these two seemingly conflicting approaches. Reiterating over the query of Example 2:

```
SELECT id FROM artists WHERE id > 100
```

we can write the following SCT, that is both flexible:

```
Ex().check_correct(
    check_column('id').has_equal_value(),
    check_node('SelectStmt').multi(
        check_edge('from_clause').has_equal_ast(),
        check_edge('where_clause').has_equal_ast()
    )
)
```

Notice how `check_correct()` starts from `Ex()` and is passed two arguments.

The first argument is an SCT chain of result-based checks that starts from the state passed to `check_correct()` (`Ex()` in this case). If this SCT chain passes when executed, the second argument of `check_correct()` is no longer executed. If this SCT chain fails, the SCT does not fail and throw a feedback message. Instead, the second argument, an SCT of AST-based checks is executed that also starts from the state passed to `check_correct()`. This chain will generate more detaild feedback.

Note:

- `check_correct()` passes the state it chains off of to its arguments, the subchains.
- `multi()` is used to 'split SCT chains' from the state it chains off of, and passes this state to its arguments.

This SCT:

- is flexible if the student got the right end result. For all of the following queries, the *check_column('id').has_equal_value()* chain passes, so the *check_node()* chain does not run:

  ```
  SELECT id FROM artists WHERE id > 100
  SELECT id, name FROM artists WHERE id > 100 AND id < 0
  SELECT id FROM artists WHERE id > 100 ORDER BY id DESC
  ```

- generates specific feedback if the student made a mistake: Instead of the generic "Column `id` seems to be incorrect" message, there are more targeted messages:

  ```
  SELECT id FROM labels WHERE id > 100          -- "Check the FROM clause."
  SELECT id FROM artists WHERE id > 50          -- "Check the WHERE clause."
  SELECT id FROM artists WHERE id > 100 LIMIT 5   -- "The column id seems to be␣
  ↪incorrect."
  ```

  Notice how for the last example here, all functions in the AST-based SCT chain passed. In this case, the more generic message is still thrown in the end.

Have a look at the glossary for more examples of SCTs that nicely combine these two families of SCT functions.

For other guidelines on writing good SCTs, check out the 'How to write good SCTs' section on DataCamp's general SCT documentation page.

CHAPTER 7

Result Checks

As you could read in the tutorial, there are two families of SCT functions in sqlwhat:

- result-based checks, that look at the result of the student's query and the solution query
- AST-based checks, that look at the abstract syntax tree representation of the query and allow you to check for elements.

This article gives an overview of all result-based checks, that are typically used as the first argument to `check_correct()`.

On the lowest level, you can have the SCT fail when the student query generated an error with `has_no_error()`:

```
Ex().has_no_error()
```

Next, you can verify whether the student query actually returned a result with `has_result()`. Behind the scenes, this function first uses `has_no_error()`: if the query resulted in an error, it cannot return a result:

```
Ex().has_result()
```

More high-level, you can compare the number of rows of the student's query result and the solution query result with `has_nrows()`. This is useful to check whether e.g. a `WHERE` or `LIMIT` clause was coded up correctly to only return a subset of results:

```
Ex().has_nrows()
```

Similarly, but less-used, you can also compare the number of columns the student's query and solution query returned. This function fails if the number of columns doesn't match. It passes if they do match, even if the column names differ:

```
Ex().has_ncols()
```

With `check_row()`, you can zoom in on a particular record of the student and solution query result, so you can compare there values later on with `has_equal_value()` (see further). The example below zooms in on the third row (with index 2) of the student query result, returning a state that only considers the data for the third row. It fails if the student query does not contain 3 rows.

```
Ex().check_row(2)
```

Similarly, and more often used than `check_row()`, you can use `check_column()` to zoom in on a particular column in the student's and solution query result by name. The function fails if the column cannot be found in the student query:

```
Ex().check_column('title')
```

Often, the solution selects multiple columns from a table. Writing a `check_column()` for every column in there would be tedious. Therefore, there is a utility function, `check_all_columns()`, that behind the scenes runs `check_column()` for every column that is found in the solution query result, after which it zooms in on these columns. Suppose you have a solution query that returns three columns, named `column1`, `column2` and `column3`. If you want to verify whether these columns are also included in the student query result, you have different options:

```
# verbose: use check_column thrice
Ex().multi(
    check_column('column1'),
    check_column('column2'),
    check_column('column3')
)

# short: use check_all_columns()
Ex().check_all_columns()
```

As an extra in `check_all_columns()`, you can also set `allow_extra` to `False`, which causes the function to fail if the student query contains columns that the solution column *does not* contain. `allow_extra` is `True` by default.

All of the functions above were about checking whether the number of rows/columns are correct, whether some rows/columns could be found in the query, but none of them look at the actual contents of the returned table. For this, you can use `has_equal_value()`. The function simply looks at the student's query result and solution query result or a subset of them (if `check_row`, `check_column` or `check_all_columns()` were used):

```
# compare entire query result (needs exact match)
Ex().has_equal_value()

# compare records on row 3
Ex().check_row(2).has_equal_value()

# compare columns title
Ex().check_column('title').has_equal_value()

# zoom in on all columsn that are also in the solution and compare them
Ex().check_all_columns().has_equal_value()
```

By default, `has_equal_value()` will order the records, so that order does not matter. If you want order to matter, you can set `ordered=True`:

```
Ex().check_all_columns().has_equal_value(ordered = True)
```

Finally, there is a utility function called `lowercase()` that takes the state it is passed, and converts all column names in both the student and solution query result to their lowercase versions. This increases the chance for 'matches' when using `check_column()` and `check_all_columns()`.

Suppose the student did

```
SELECT Title FROM artists
```

while the solution expected

```
SELECT title FROM artists
```

Depending on the SCT you write, it will pass or fail:

```
# SCT that will fail
Ex().check_column('title').has_equal_value()

# SCT that will pass (because Title is converted to title)
Ex().check_column('Title').has_equal_value()
```

For advanced examples on how result-based checks are typically used in combination with `check_correct()`, check out the glossary!

CHAPTER 8

AST Checks

As you could read in the tutorial, there are two families of SCT functions in sqlwhat:

- result-based checks, that look at the result of the student's query and the solution query
- AST-based checks, that look at the abstract syntax tree representation of the query and allow you to check for elements.

This article gives an overview of all AST-based checks, that are typically used as the second argument to `check_correct()`.

Warning: If a student submission cannot be parsed properly because of a syntax error, all AST-based checks will *not* run! It is therefore vital that you include your AST-based checks in the second argument of `check_correct()` so they can serve as 'diagnosing' SCTs rather than as 'correctness verifying' SCTs.

8.1 Navigating the tree

The tutorial gave a gentle introduction into how the AST of a SQL query can be 'walked' as it were, using `check_edge()` and `check_node()`. In addition to the state they start from, these functions take a couple of different arguments:

- `name`: the name of the node or field you want to zoom in on,
- `index`: an optional argument to specify which occurrence of the node or field to zoom in on,
- `missing_msg`: if specified, this overrides the automatically generated feedback message in case the node or field could not be found.

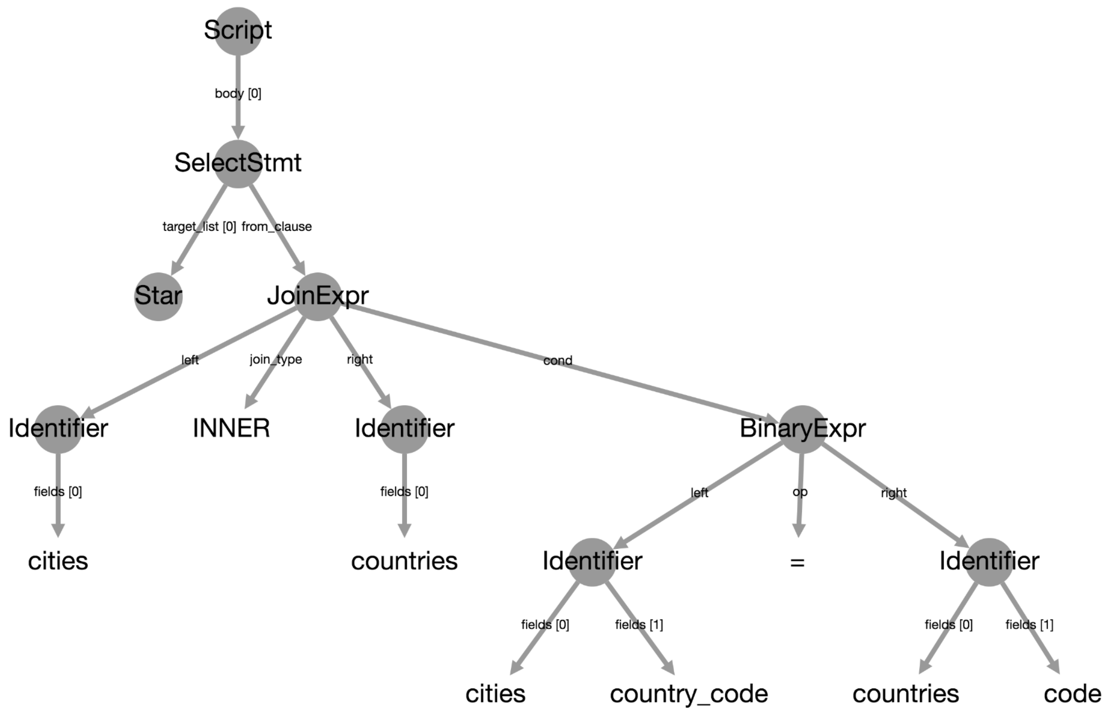
How to figure out which `name` to use? Navigate to the AST viewer and in Editor mode, paste the SQL query of the solution. This will produce a parse tree in the first box, and an Astract Syntax Tree in the second box.

For a full example of how this 'walk down the tree' works, read Example 2 of the tutorial.

As another example, suppose you want to check whether the `ON` part of an `INNER JOIN` is correct according to the following solution:

```
SELECT *
FROM cities
INNER JOIN countries
ON cities.country_code = countries.code;
```

The AST representation of this SQL query looks as follows:



To zoom in on the sub-tree that corresponds to the `ON` part, we have to:

- take the `SelectStmt` node (with `check_node('SelectStmt')`)
- walk down the `from_clause` edge (with `check_edge('from_clause')`)
- walk down the `cond` edge (with `check_edge('cond')`)

```
Ex(). \
    check_node('SelectStmt'). \
    check_edge('from_clause'). \
    check_edge('cond')
```

This SCT will focus on the following part of the solution query:

```
cities.country_code = countries.code
```

Suppose that the student made a mistake (using `code` twice), and submitted the following query:

```
SELECT *
FROM cities
INNER JOIN countries
ON cities.code = countries.code;
```

The SCT will focus on the following part of the student query:

```
cities.code = countries.code
```

8.2 Checking the tree

Once you've used a combination of `check_node()` and `check_edge()` to zoom in on a part of interest you can use `has_equal_ast()` to verify whether the elements correspond.

Continuing from the `INNER JOIN` example, we can verify whether the snippets of SQL code that have been zoomed in have a matching AST representation:

```
Ex(). \
    check_node('SelectStmt'). \
    check_edge('from_clause'). \
    check_edge('cond'). \
    has_equal_ast()
```

You can supplement this with a `check_or()` call and a manually specified `sql` snippet if you want to allow for multiple ways of specifying the condition:

```
Ex(). \
    check_node('SelectStmt'). \
    check_edge('from_clause'). \
    check_edge('cond'). \
    check_or(
        has_equal_ast(),
        has_equal_ast(sql = "countries.code = cities.code")
    )
```

Now, using either `ON cities.code = countries.code` or `countries.code = cities.code` will be accepted.

For a more complete and robust example of an `INNER JOIN` query, visit the glossary.

In addition to `has_equal_ast()`, you can also use `has_code()` to look at the actual code of a part of the SQL query and verify it with a regular expression, but you will rarely find yourself using it.

For details, questions and suggestions, contact us.

Python Module Index

Index