PyFireSQL

Release 0.1.1

Benny Cheung

Aug 26, 2023

OVERVIEW

CHAPTER

ONE

FIRESQL PARSER

The FireSQL parser consists of two parts: the lexical scanner and the grammar rule module. Python parser generator Lark is used to provide the lexical scanner and grammar rule to parse the FireSQL statement. In the end, the parser execution generates the parse tree, aka. AST (Abstract Syntax Tree). The complexity of the FireSQL syntax requires an equally complex structure that efficiently stores the information needed for executing every possible FireSQL statement.

For example, the AST parse tree for the FireSQL statement

```
SELECT id, date, email
  FROM Bookings
  WHERE date = '2022-04-04T00:00:00'
```

![An Example SQL Parse Tree]({{ site.baseurl }}images/firesql-in-python/sql_parse_tree.jpg)

Figure. Illustration of the parse tree generated by `lark`

This is delightful to use `lark` due to its design philosophy, which clearly separate the grammar specification from processing. The processing is applied to the parse tree by the Visitor or Transformer components.

1.1 Visitor and Transformer

Visitors and Transformer provide a convenient interface to process the parse-trees that Lark returns. `lark` documentation defines,

- **Visitors** - visit each node of the tree, and run the appropriate method on it according to the node's data. They work bottom-up, starting with the leaves and ending at the root of the tree.
- **Transformers** - work bottom-up (or depth-first), starting with visiting the leaves and working their way up until ending at the root of the tree.
  - For each node visited, the transformer will call the appropriate method (callbacks), according to the node's `data`, and use the returned value to replace the node, thereby creating a new tree structure.
  - Transformers can be used to implement map & reduce patterns. Because nodes are reduced from leaf to root, at any point the callbacks may assume the children have already been transformed.

  Using Visitor is simple at first, but you need to know exactly what you're fetching, the children chain can be difficult to navigate depending on the grammar which produce the parsed tree.

We decided to use Transformer to transform the parse tree to the corresponding SQL component objects that can be easily consumed by the subsequent processing.

For instance, the former example parse tree is transformed into SQL components as,

```
SQL_Select(
  columns=[SQL_ColumnRef(table=None, column='id'),
           SQL_ColumnRef(table=None, column='date'),
           SQL_ColumnRef(table=None, column='email')],
  froms=[SQL_SelectFrom(part='Bookings', alias=None)],
  where=SQL_BinaryExpression(operator='==',
                             left=SQL_ColumnRef(table=None, column='date'),
                             right=SQL_ValueString(value='2022-04-04T00:00:00'))
)
```

With this transformed data structure, we can write the processor walking through the components and produce a execution plan to the corresponding Firestore queries.

CHAPTER

TWO

JUST ENOUGH SQL FOR FIRESQL

To get going, we don't need the full SQL parser and transformer for the DML (Data Manipulation Language). We define ONLY the `SELECT` statement, just enough for Firestore collections query to serve our immediate needs.

2.1 FireSQL Grammar

A grammar is a formal description of a language that can be used to recognize its structure. The most used format to describe grammars is the **Extended Backus-Naur Form** (EBNF). A typical rules in a Backus-Naur grammar looks like this:

```
where_clause ::= bool_expression
bool_expression ::= bool_parentheses
                    | bool_expression "AND" bool_parentheses
                    | bool_expression "OR" bool_parentheses
bool_parentheses ::= comparison_type
                     | "(" bool_expression "AND" comparison_type ")"
                     | "(" bool_expression "OR" comparison_type ")"
...
CNAME ::= ("_"|"/"|LETTER) ("_"|"/"|LETTER|DIGIT)*
...
```

The `where_clause` is usually nonterminal, which means that it can be replaced by the group of elements on the right, `bool_expression`. The element `bool_expression` could contains other nonterminal symbols or terminal ones. Terminal symbols are simply the ones that do not appear as a <symbol> anywhere in the grammar and capitalized. A typical example of a terminal symbol is a string of characters, like "(", ")", "AND", "OR", "CNAME".

2.2 Collection Path

The Firestore collection has a set of documents. Each document can be nested with more collections. Firestore identifies a collection by a path, looks like `Companies/bennycorp/Users` means `Companies` collection has a document `bennycorp`, which has `Users` collection.

If we want to query a nested collection, we can specify the collection name as a path. The paths can be long but we can use `AS` keyword to define their alias names.

For example, the subcollection `Users` and `Bookings` are specified with `Companies/bennycorp` document.

```
SELECT u.email, u.state, b.date, b.state
  FROM
```

(continues on next page)

(continued from previous page)

```
  Companies/bennycorp/Users as u JOIN Companies/bennycorp/Bookings as b
  ON u.email = b.email
WHERE
    u.state = 'ACTIVE' AND
    b.date >= '2022-03-18T04:00:00'
```

Interesting Firestore Fact: collection path must have odd number of parts.

2.3 Document Field and Sub-field

Since Firestore document field can have nested sub-field, FireSQL statement column reference can reach the document sub-fields by quoted string, using the " to escape the field name with `.` in it. The quoted string can be used anywhere that a column reference is allowed.

For example, the `Users` document's `location` field, which has a sub-field `displayName`. The sub-field can be reached by `"location.displayName"`

```
SELECT email, "location.displayName"
FROM Users
WHERE "location.displayName" = 'Work From Home'
```

2.4 Document ID

Firestore has a unique "document ID" that associated with each document. The document ID is not part of the document fields that we need to provide special handling to access. FireSQL introduced a special field `docid` to let any statement to reference to the unique "document ID".

For example, we can select where the document equals to a specific `docid` in the `Users` collection. Even though the document does not have `docid` field, we can also project the `docid` value in the output.

```
SELECT docid, email
FROM Users
WHERE docid = '4LLlLw6tZicB40HrjhDJNmvaTYw1'
```

Due to Firestore admin API limitations, we can ONLY express = equal or `IN` operators with `docid`. For example, the following statement will find documents that in the specified array of `docid`.

```
SELECT docid, email
FROM Users
WHERE docid IN ('4LLlLw6tZicB40HrjhDJNmvaTYw1', '74uWntZuVPeYcLVcoS0pFApGPdr2')
```

More interesting, if we want to project all the fields, including the `docid`. We can do the select statement like, `docid` and * are projected in the output.

```
SELECT docid, *
FROM Users
WHERE "location.displayName" = 'Work From Home'
```

2.5 DateTime Type

Consistent description of date-time is a big topic that we made a practical design choice. We are using ISO-8601 to express the date-time as a string, while Firestore stores the date-time as `Timestamp` data type in UTC. For example,

- writing "March 18, 2022, at time 4 Hours in UTC" date-time string, is "2022-03-18T04:00:00".
- writing "March 18, 2022, at time 0 Hours in Toronto Time EDT (-4 hours)" date-time string, is "2022-03-18T00:00:00-04".

If in doubt, we are using the following to convert, match and render to the ISO-8601 string for date-time values.

```
DATETIME_ISO_FORMAT = "%Y-%m-%dT%H:%M:%S"
DATETIME_ISO_FORMAT_REGEX = r'^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[01]|0[1-
↪9]|[12][0-9])T(2[0-3]|[01][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?(Z|[+-](?:2[0-
↪3]|[01][0-9]):[0-5][0-9])?$'
```

2.6 Pattern Matching by LIKE

The SQL expression `LIKE` or `NOT LIKE` can be used for matching string data.

```
SELECT docid, email, state
  FROM
    Users
  WHERE
    state IN ('ACTIVE') AND
    email LIKE '%benny%'
```

After the Firebase query, the pattern matching is used as the filtering expression. The SQL processor supports pattern for:

- prefix match `pattern%`
- suffix match `%pattern`
- infix match `%pattern%`

2.7 JSON Data

PyFireSQL provides JSON data supports, in particular, for the `INSERT` and `UPDATE` statements that must take complex data types. When the field value needs to take the complex data types, such as array or map (aka. Python dict), these complex data types must be encoded within a JSON enclosure. The JSON enclosure can interpret any valid JSON object; subsequently translates into the corresponding Firestore supported data types.

For example,

```
INSERT INTO Companies/bennycorp/Visits
  ( email, event )
  VALUES
    ( 'btscheung+test1@gmail.com', JSON(["event1","event2","event3"]) )
```

When the collection `Visits` has a field `event` which takes an array of event names, we assign `event` field by using the `JSON` enclosure to encode the array `["event1","event2","event3"]` with a valid JSON string.

Since we are dealing with Firestore as a document structure without a schema, we can insert all the key pairs from a JSON map into the collection.

For example, the following insert statement - column specification uses * to indicate all fields. We are inserting a list of `email`, `firstName`, `lastName`, `groups` (as array), `roles` (as array), `vaccination`, `access` (as map).

```
INSERT INTO Companies/bennycorp/Users
  ( * )
  VALUES (
    JSON(
      {
        "email": "btscheung+twotwo@gmail.com",
        "name": "Benny TwoTwo",
        "groups": [],
        "roles": [
            "ADMIN"
        ],
        "vaccination": null,
        "access": {
          "hasAccess": true
        }
      }
    )
  )
```

CHAPTER

THREE

FIRESQL STATEMENTS

FireSQL supports the following SQL-like statements.

The set of implemented SQL-like DML (Data Manipulation Language) statements are,

Please read the details in the corresponding FireSQL statement sections.

3.1 Multiple Statements

The `FireSQL.execute()` function can take one or more FireSQL statements. Sequence of statements must be separated by semi-colon ';'.

For example,

```
INSERT INTO Users (email, name) VALUES ('btscheung+oneone@gmail.com', 'Benny OneOne');
INSERT INTO Users (email, name) VALUES ('btscheung+twotwo@gmail.com', 'Benny TwoTwo');
INSERT INTO Users (email, name) VALUES ('btscheung+threethree@gmail.com', 'Benny␣
↪ThreeThree')
```

- The last FireSQL statement's semi-colon is optional.

CHAPTER

FOUR

SELECT STATEMENT

The SELECT statement is used to select documents from a collection.

4.1 SELECT Syntax

```
SELECT [[ALL] DISTINCT] field1, field2, ...
FROM collection_name
WHERE conditions
```

Here, field1, field2, ... are the field names of the collection to select data from. The `DISTINCT` modifier will select the unique values from field1 and `ALL DISTINCT` modifier will select the unique values from all (field1, field2, ...). If we want to select all the fields available in the collection, use the following syntax:

```
SELECT *
FROM collection_name
```

By using `lark` EBNF-like grammar, we have encoded the core `SELECT` statement, which is subsequently transformed into Firestore collection queries to be executed.

- SELECT columns for collection field's projection
  - DISTINCT modifier restricts the result only included the unique field value
  - ALL DISTINCT modifier restricts the result only included the unique all fields value
- FROM sub-clause for collections
- FROM/JOIN sub-clause for joining collections (restricted to 1 join)
- WHERE sub-clause with boolean algebra expression for each collection's queries on field values
  - boolean operators: AND (currently OR is not implemented)
  - operators: =, !=, >, <, <=, >=
  - container expressions: IN, NOT IN
  - array contains expressions: CONTAIN, ANY CONTAIN
  - filter expressions: LIKE, NOT LIKE
  - null expressions: IS NULL, IS NOT NULL
- Aggregation functions applied to the result set
  - COUNT for any field

  - SUM, AVG, MIN, MAX for numeric field

But the processor has the following limitations, which we can provide post-processing on the query results set.

- No ORDER BY sub-clause
- No GROUP BY/HAVING sub-clause
- No WINDOW sub-clause

4.2 SELECT Examples

For example, the following statements can be expressed,

All keywords are case insensitive. All whitespaces are ignored by the parser.

`docid` is a special field name to extract the selected document's Id

```
SELECT docid, email, state
  FROM
    Users
  WHERE
    state = 'ACTIVE'
```

The * will select all fields, boolean operator 'AND' to specify multiple query criteria.

```
SELECT *
  FROM
    Users
  WHERE
    state IN ('ACTIVE') AND
    u.email LIKE '%benny%'
```

The field-subfield can use the " to escape the field name with `.` in it.

```
SELECT *
  FROM
    Users as u
  WHERE
    u.state IN ('ACTIVE') AND
    u."location.displayName" = 'Work From Home'
```

The `JOIN` expression to join 2 collections together

```
SELECT u.email, u.state, b.date, b.state
  FROM
    Users as u JOIN Bookings as b
    ON u.email = b.email
  WHERE
      u.state = 'ACTIVE' AND
      u.email LIKE '%benny%' AND
      b.state IN ('CHECKED_IN', 'CHECKED_OUT') AND
      b.date >= '2022-03-18T04:00:00'
```

The `COUNT`, `MIN`, `MAX`, `SUM`, `AVG` are the aggregation functions computed against the result set. Only numeric field (e.g. `cost` here) is numeric to have a valid value for `MIN`, `MAX`, `SUM`, `AVG` computation.

```
SELECT COUNT(*), MIN(b.cost), MAX(b.cost), SUM(b.cost), AVG(b.cost)
  FROM
    Users as u JOIN Bookings as b
    ON u.email = b.email
  WHERE
      u.state = 'ACTIVE' AND
      u.email LIKE '%benny%' AND
      b.state IN ('CHECKED_IN', 'CHECKED_OUT') AND
```

The `DISTINCT` modifier will select only the unique field(s).

```
SELECT DISTINCT email
  FROM
    Bookings
  WHERE
    date > '2022-04-01T00:00:00'
```

See firesql.lark for the FireSQL grammar specification.

CHAPTER

FIVE

INSERT INTO STATEMENT

The INSERT INTO statement is used to insert new document in a collection.

5.1 INSERT INTO Syntax

Specify both the column names and the values to be inserted:

```
INSERT INTO collection_name (field1, field2, field3, ...)
VALUES (value1, value2, value3, ...);
```

5.2 INSERT INTO Examples

The following SQL statement inserts a new document in the `Users` collection

```
INSERT INTO Users
    ( email, name, vaccination )
  VALUES
    ( 'btscheung+test1@gmail.com', 'Benny TwoTwo', NULL )
```

Since we are dealing with Firestore as a document structure without a schema, we can insert all the key pairs from a JSON map into the collection.

For example, the following insert statement - column specification uses * to indicate all fields. We are inserting a list of `email`, `firstName`, `lastName`, `groups` (as array), `roles` (as array), `vaccination`, `access` (as map).

```
INSERT INTO Companies/bennycorp/Users
    ( * )
  VALUES (
    JSON(
      {
        "email": "btscheung+twotwo@gmail.com",
        "name": "Benny TwoTwo",
        "groups": [],
        "roles": [
            "ADMIN"
        ],
        "vaccination": null,
```

(continues on next page)

(continued from previous page)

```
        "access": {
          "hasAccess": true
        }
      }
    )
  )
```

CHAPTER

SIX

UPDATE STATEMENT

The UPDATE statement is used to modify the existing documents in a collection.

6.1 UPDATE Syntax

```
UPDATE collection_name
SET field1 = value1, field2 = value2, ...
WHERE condition;
```

> Note: Be careful when updating documents in a collection! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which document(s) that should be updated. If we omit the WHERE clause, all documents in the collection will be updated!

6.2 UPDATE Examples

The following UPDATE statement updates the user with email "btscheung+twotwo@gmail.com" to state "ACTIVE" in the "Users" collection:

```
UPDATE Users
SET
  state = 'ACTIVE'
WHERE
  email = 'btscheung+twotwo@gmail.com'
```

If we want to update with a field that takes complex data type, e.g. array or map, we must use "JSON()" data enclosure to encode the data.

```
UPDATE Users
  SET
    state = 'INACTIVE',
    groups = JSON(["TeamA", "TeamB"])
  WHERE
    state = 'ACTIVE' AND
    email = 'btscheung+twotwo@gmail.com'
```

CHAPTER

SEVEN

DELETE STATEMENT

The DELETE statement is used to delete existing documents in a collection.

7.1 DELETE Syntax

```
DELETE FROM table_name
WHERE condition;
```

> Note: Be careful when deleting documents in a collection! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which document(s) should be deleted. If you omit the WHERE clause, all documents in the collection will be deleted!

7.2 DELETE Examples

The following DELETE statement deletes the user with email "btscheung+twotwo@gmail.com" from the "Users" collection:

```
DELETE
  FROM Users
  WHERE
    email = 'btscheung+twotwo@gmail.com'
```

CHAPTER

EIGHT

FUTURE

If you've read up to this point, means that you're having the same pain with Cloud Firestore query. We hope this article motivates you to try out PyFireSQL. It can be your preferred programming interface to Firestore using Python!

FireSQL has many improvements to be implemented. Just to name a few future improvements,

- multiple JOIN statements in the FROM clause
- allow OR boolean expression in the WHERE clause
- optimize the query plan before sending queries to Firestore
- support sub-query in SELECT clause

Please join me on the PyFireSQL open source project, or provide feedbacks to improve FireSQL utilities!

CHAPTER

NINE

PROGRAMMING INTERFACE

In PyFireSQL, we offer a simple programming interface to parse and execute firebase SQL. Please consult Firebase Admin SDK Documentation to generate the project's service account `credentials.json` file.

```
from firesql.firebase import FirebaseClient
from firesql.sql.sql_fire_client import FireSQLClient
from firesql.sql import FireSQL

# make connection to Cloud Firestore
client = FirebaseClient()
client.connect(credentials_json='credentials.json')
# wrapped as FireSQL client interface
sqlClient = FireSQLClient(client)

# query via the FireSQL interface - the results are in list of docs (Dict)
query = "SELECT * FROM Users WHERE state = 'ACTIVE'"
fireSQL = FireSQL()
docs = fireSQL.execute(sqlClient, query)
```

After `fireSQL.execute()` query completed, the results are a list of docs (as Dict) that satisfied the query. Then we can pass the list of docs to render into any output format, in our case, the `DocPrinter` object can output `csv` or `json` with the select fields.

```
from firesql.sql.doc_printer import DocPrinter

docPrinter = DocPrinter()
if format == 'csv':
  docPrinter.printCSV(docs, fireSQL.select_fields())
else:
  docPrinter.printJSON(docs, fireSQL.select_fields())
```

For further post-processing, we can use Pandas's Dataframe to perform any data analysis, grouping, sorting and calculations. The list of docs (as Dict) can be directly imported into Dataframe! very convenience.

```
import pandas as pd

df = pd.DataFrame(docs)
```

CHAPTER

TEN

QUERY SCRIPT

In addition, we provide an interface script `firesql-query.py` to accept an FireSQL statement.

```
usage: firesql-query.py [-h] [-c CREDENTIALS] [-f FORMAT] [-i INPUT]
                        [-q QUERY]

optional arguments:
  -h, --help            show this help message and exit
  -c CREDENTIALS, --credentials CREDENTIALS
                        credentials JSON path
  -f FORMAT, --format FORMAT
                        output format (csv|json)
  -i INPUT, --input INPUT
                        FireSQL query input file (required)
  -q QUERY, --query QUERY
                        FireSQL query (required)
```

For example, finding all `ACTIVE` users from Users collection

```
python firesql-query.py -c credential.json \
  -q "SELECT docid, email, state FROM Users WHERE state IN ('ACTIVE')"
```

> `docid` is a special column name that is used to project the Firestore document ID.

The default query result is rendered in "csv" output format.

```
"docid","email","state"
"0r6YWowe9rW65yB1qTKsCe83cCm2","btscheung+real@gmail.com","ACTIVE"
"1utcUa9fdheOlrMe9GOCjrJ3wjh1","btscheung+bennycorp@gmail.com","ACTIVE"
"7CUJOqe6rlOTQuatc27EQGivZfn2","btscheung+twotwo@gmail.com","ACTIVE"
...
```

Alternatively, by specifying the `-f json` output format, the result will be,

```
[
  {"docid": "0r6YWowe9rW65yB1qTKsCe83cCm2", "email": "btscheung+real@gmail.com", "state
↪": "ACTIVE"},
  {"docid": "1utcUa9fdheOlrMe9GOCjrJ3wjh1", "email": "btscheung+bennycorp@gmail.com",
↪"state": "ACTIVE"},
  {"docid": "7CUJOqe6rlOTQuatc27EQGivZfn2", "email": "btscheung+twotwo@gmail.com", "state
↪": "ACTIVE"},
```

(continues on next page)

(continued from previous page)

```
  ...
]
```

10.1 SQL Input File

For more complicated SQL, we can use `-i input.sql` to specify the SQL input file.

`input.sql` file:

```
SELECT u.email, u.state, b.date, b.state
  FROM
    Users as u JOIN Bookings as b
    ON u.email = b.email
  WHERE
      u.state IN ('ACTIVE') and
      b.state IN ('CHECKED_IN', 'CHECKED_OUT') and
      b.date >= '2022-03-18T04:00:00'
```

By execute the input file

```
python firesql-query.py -c credentials.json -i input.sql
```

The result will be,

> NOTE: the column `state` from `Users` will be automatically disambiguated by appending the alias prefix `u_state`.

```
"email","u_state","date","state"
"btscheung+bennycorp@gmail.com","ACTIVE","2022-03-18T04:00:00","CHECKED_IN"
"btscheung+bennycorp@gmail.com","ACTIVE","2022-03-18T04:00:00","CHECKED_IN"
"btscheung+hill6@gmail.com","ACTIVE","2022-03-31T04:00:00","CHECKED_IN"
...
```

CHAPTER

ELEVEN

API REFERENCE

11.1 FireSQL

class firesql.sql.fire_sql.**FireSQL**

FireSQL is the main programming interface to execute FireSQL statements

During FireSQL initialization, the FireSQL parser is prepared from *sql/grammar/firesql.lark*.

select_fields() → List

From the parsed FireSQL select statement, return the select fields.

Parameters
None –

Returns
The list of select fields as strings

execute(*client: FireSQLAbstractClient*, *sql: str*, *options: Dict = {}*) → List

Given a Firebase connection, parse and execute all the FireSQL statements.

Parameters

- **client** (*FirebaseClient*) – The client has established a Firebase connection
- **sql** (*str*) – FireSQL statement to be executed
- **options** (*Dict*) – Unused

Returns
A list of executed documents

Return type
docs

execute_command(*client: FireSQLAbstractClient*, *sqlCommand: Union[SQL_Select, SQL_Insert, SQL_Update, SQL_Delete]*, *options: Dict = {}*) → List

Given a Firebase connection, execute a FireSQL statements.

Parameters

- **client** (*FirebaseClient*) – The client has established a Firebase connection
- **sqlCommand** (*SQL_Select* | *SQL_Insert* | *SQL_Update* | *SQL_Delete*) – FireSQL statement to be executed
- **options** (*Dict*) – Unused

Returns
A list of executed documents

Return type
docs

class firesql.sql.doc_printer.**DocPrinter**

The DocPrinter class is for printing the select documents as CSV or JSON format output.

Console output in the specified format.

printCSV(*docs*, *selectFields*)

printCSV is to print the given list of documents from the select fields in CSV output format

Parameters

- **docs** (*List of documents as Dict*) – the list of documents after FireSQL select query
- **selectFields** (*List of fields to output*) – the list of select fields to be picked out from each document (as Dict)

Returns
string output in CSV format

Return type
str

printJSON(*docs*, *selectFields*)

printJSON is to print the given list of documents from the select fields in JSON output format

Parameters

- **docs** (*List of documents as Dict*) – the list of documents after FireSQL select query
- **selectFields** (*List of fields to output*) – the list of select fields to be picked out from each document (as Dict)

Returns
string output in JSON format

Return type
str

CHAPTER

TWELVE

REFERENCES

For further research and developments, use the following references to start.

12.1 FireSQL

- PyFireSQL https://github.com/bennycheung/PyFireSQL
  - PyPi https://pypi.org/project/pyfiresql/

12.2 Firebase Python

- Google Cloud Firestore https://firebase.google.com/products/firestore
- Google Cloud Firestore Python Client SDK https://googleapis.dev/python/firestore/latest/client.html
- Firebase Admin SDK Documentation https://firebase.google.com/docs/admin/setup

12.3 Language Parsing

- Gabriele Tomassetti, Parsing In Python: Tools And Libraries, https://tomassetti.me/parsing-in-python/
- Lark Documentation https://lark-parser.readthedocs.io/en/latest/
  - Code Repo: lark-parser

12.4 Similar Projects

The following projects inspired PyFireSQL development. They have a different purpose or different base language.

- From SQL to Ibis Parsing - sql_to_ibis https://github.com/zbrookle/sql_to_ibis
  - `sql_to_ibis` is a Python package that translates SQL syntax into ibis expressions. This provides the capability of using only one SQL dialect to target many different backends.
- Ibis: Python data analysis framework for Hadoop and SQL engines, https://ibis-project.org/docs/dev/
  - Code Repo: https://github.com/ibis-project/ibis

  - Ibis is a Python framework to access data and perform analytical computations from different sources, in a standard way.

- FireSQL (Node.js - Typescript) Project, https://firebaseopensource.com/projects/jsayol/firesql/
  - Code Repo: https://github.com/jsayol/firesql
  - FireSQL is a library built on top of the official Firebase SDK that allows you to query Cloud Firestore using SQL syntax. It's smart enough to issue the minimum amount of queries necessary to the Firestore servers in order to get the data that you request.

CHAPTER

THIRTEEN

WHAT IS PYFIRESQL

PyFireSQL is a SQL-like programming interface to query Cloud Firestore collections using Python. Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in documents, which are organized into collections.

There is no formal query language to Cloud Firestore - NoSQL collection/document structure. For many instances, we need to use the useful but clunky Firestore UI to navigate, scroll and filter through the endless records. With the UI, we have no way to extract the found documents. Even though we attempted to extract and update by writing a unique program for the specific task, we felt many scripts are almost the same that something must be done to limit the endless program writing. What if we can use SQL-like statements to perform the data extraction, which is both formal and reusable? - This idea will be the motivation for the FireSQL language!

Even though we see no relational data model of (table, row, column), we can easily see the equivalent between table -> collection, row -> document and column -> field in the Firestore data model. The SQL-like statement can be transformed accordingly.

CHAPTER

FOURTEEN

INSTALL PYFIRESQL

```
$ pip install pyfiresql
```

To install from [PyFireSQL source](https://github.com/bennycheung/PyFireSQL), checkout the project

```
cd PyFireSQL
# install require packages
pip install -r requirements.txt
# install (optional) development require packages
pip install -r requirements_dev.txt

python setup.py install
```

CHAPTER

FIFTEEN

PROGRAMMING INTERFACE

In PyFireSQL, we offer a simple programming interface to parse and execute firebase SQL. Please consult [Firebase Admin SDK Documentation](https://firebase.google.com/docs/admin/setup) to generate the project's service account *credentials.json* file.

```
from firesql.firebase import FirebaseClient
from firesql.sql import FireSQL

# make connection to Cloud Firestore
client = FirebaseClient()
client.connect(credentials_json='credentials.json')

# query via the FireSQL interface - the results are in list of docs (Dict)
query = "SELECT * FROM Users WHERE state = 'ACTIVE'"
fireSQL = FireSQL()
docs = fireSQL.sql(client, query)
```

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX