datahappy

blogging about BI, (Big) Data and other dev stuff

Redshift server-side cursor with Psycopg2 Python adapter for better query performance

MARCH 8, 2020MARCH 13, 2020 DATAHAPPY LEAVE A COMMENT

When it comes to extracting data out of AWS Redshift using a Python client, you'd typically end up using Psycopg2 Postgres adapter because it's the most common one. Today I'd like to share some of my thoughts on how to boost up the performance when querying large datasets. The main topic is going to be AWS Redshift server-side vs. client-side cursors. However when it comes to querying large data sets from Redshift, you have as far as I know atleast 3 options.

#**1: Option number one is to unload the queried data directly** to a file in AWS S3. See the press for this option here (https://docs.aws.amazon.com/redshift/latest/dg/t_Unloading_tables.html). That is the best option when it comes to performance, but might not be usable every time (for instance when you need to post-process the data in the extracted files, this option also does some implicit file splitting per node slice which might be undesired behavior etc.)

#**2: Option number two** is the second best performing alternative, and that is to use a **server-side Redshift cursor**. This happens when you make use of the argument "name" while initializing a Postgres cursor, like I do below :

```
self.cur = self.conn.cursor(name=self.cursor_name)
```

I found server-side cursors to have a huge performance improvement impact for datasets >1M and all the way up to 10M rows (this is the max amount of rows I've been testing with, but can be much more) , however be careful, as at some point AWS is recommending to not use them for very large datasets … *Because cursors materialize the entire result set on the leader node before beginning to return results to the client, using cursors with very large result sets can have a negative impact on performance.*

It is needed to point out, that server-side cursors can be created only when the query is a plain SELECT or a SELECTs with CTEs. Basically usage of DDL and DML statements in your queries blocks creating a server side cursor, so it's handled in the example below with declaration of a list of the blocking statements and checking the query does not contain any of these.

#**3: Option number three is using a plain client-side cursor**, basically the difference in implementation is, that you do not make any use of the name argument while initializing the Postgres cursor, see:

```
self.cur = self.conn.cursor()
```

The performance is typically OK-ish for datasets having thousands or tens of thousands of rows.

For these 2 cursor types, I'd highly recommend to fetch by many rows using a Python generator yielding the rows, instead of the cursor fetchall method. Fetchall worked fine for me for small datasets (tested on ~ 10 columns, ~30k rows) but in general I don't recommend it. I've seen the fetchall method combined with the client-side cursor raise Memory errors on a 10 column 300k rows dataset in AWS Batch while having 2GB memory set in it's job definition.

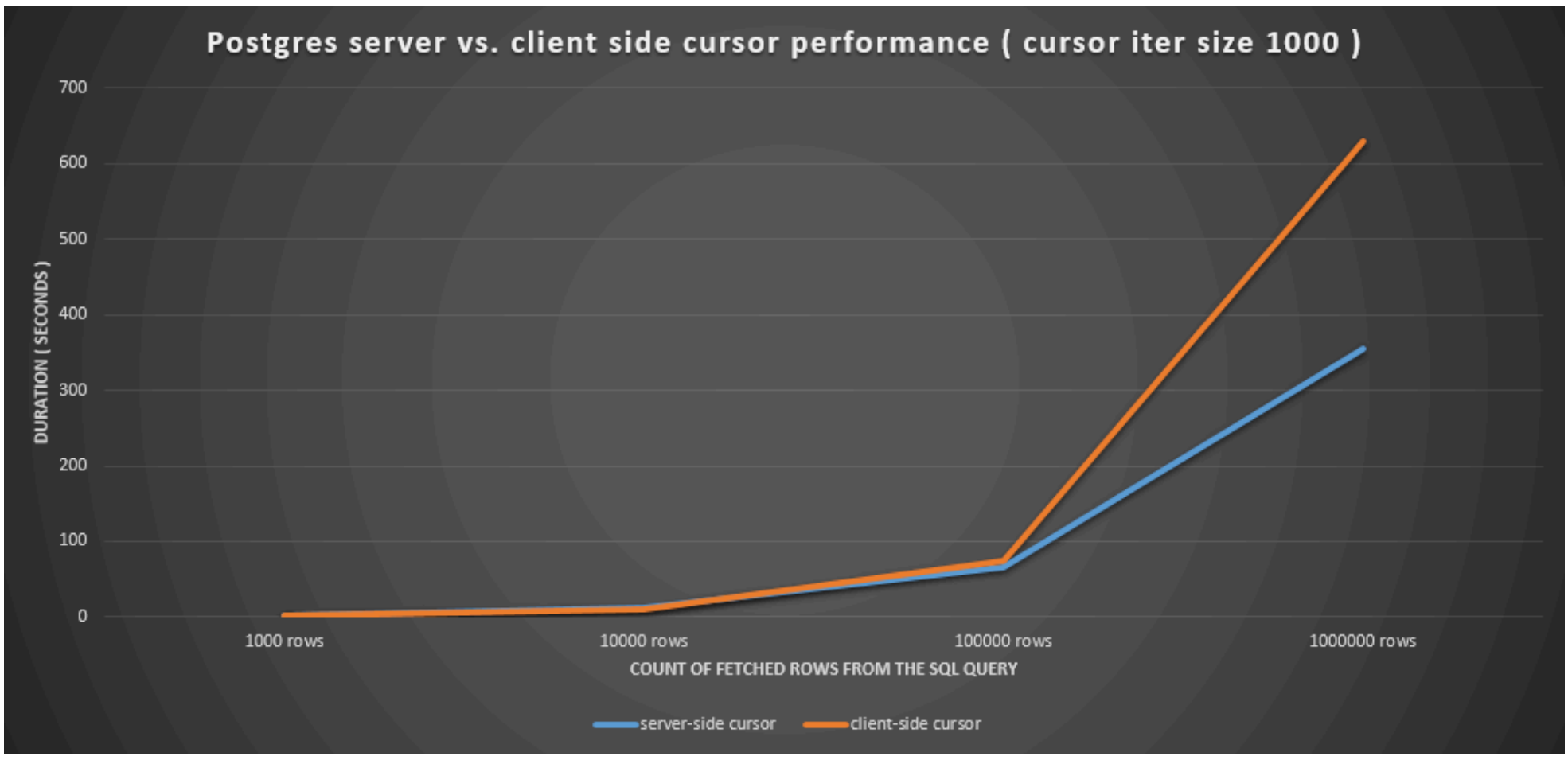
Don't forget to set the fetch_row_size parameter to your needs, 1000 is probably a small size and will result in many remote DB roundtrips. Size of 10000 would be a good starting point in my opinion.

Here's an quick example tied to a public Postgres database (https://rnacentral.org/help/public-database) (which should be good enough for illustration) connection I prepared for sandboxing purposes so you can see how it's possible to utilize server-side cursor in your codebase:

```
import psycopg2


class CustomException(Exception):
    pass


class CNX:
    def __init__(self):
        self.conn = None
        self.fetch_row_size = 1000
        # here I set the query limit for the performance
        # comparison chart below
        self.query = "SELECT * from rna LIMIT 1000;"
        self.cursor_name = 'my_cursor'

    def init_connection(self):
        """
        Initiate a DB connection
        :return: connection conn object
        """
        try:
            self.conn = psycopg2.connect(database="pfmegrnargs", user="reader",
                                         password="NWDMCE5xdipIjRrp",
                                         host="hh-pgsql-public.ebi.ac.uk", port="5432")

    except psycopg2.Error as postgres_exc:
        raise CustomException("Psycopg2 connect exception: {}".format(postgres_exc))

    return

def connect_to_data_source(self):
    with self.conn:
        # AWS Redshift Psycopg2 data source DOES support named
        # server-side cursors and setting its itersize for
        # single-select queries and select queries with CTEs
        _stmts_blocking_named_server_side_cursor = ["CREATE", "INSERT", "DELETE", "UPDATE",
                                                    "DROP", "TRUNCATE", "CALL"]

        if not any(x in self.query.upper() for x in _stmts_blocking_named_server_side_cursor):
            print('Named server side cursor used for query execution')
            self.cur = self.conn.cursor(name=self.cursor_name)
            self.cur.itersize = self.fetch_row_size

        # AWS Redshift Psycopg2 data source DOES NOT support
        # server-side cursors for queries using temp tables to store temporary results, DDL etc.
        else:
            print('Simple cursor used for query execution')
            self.cur = self.conn.cursor()

        return

    def execute(self):
        """
        execute the query
        :param query:
        :return:
        """
        try:
            self.cur.execute(self.query)

        except psycopg2.Error as postgres_exc:
            raise CustomException("Psycopg2 execute query exception: {}".format(postgres_exc))

        return

    def fetch_from_db(self):
        print("fetching rows START")
        while True:
            results = self.cur.fetchmany(size=self.fetch_row_size)
            if not results:
                print("fetching rows END")
                break
            print("fetching rows...")
            yield results
```

```
# usage of fetched rows example:
cnx_obj = CNX()
cnx_obj.init_connection()
cnx_obj.connect_to_data_source()
cnx_obj.execute()

for batch_rows in cnx_obj.fetch_from_db():
    print(batch_rows)
```

Let's evaluate the query performance results by comparing the duration of execution runtime of the code snippet I shared above. I think the chart is pretty self-explanatory in this case, server-side cursor performance wins massively over client-side as the amount of fetched rows returned from the query increases.



(https://datahappy.wordpress.com/wp-content/uploads/2020/03/ss-cs-1.png)

AWS, PYTHON, REDSHIFT AWS, CLIENT-SIDE CURSOR, POSTGRES, QUERY PERFORMANCE, REDSHIFT, SERVER-SIDE CURSOR