Gradient Descent for Linear Regression: Performance and Scalability Analysis of Local, Snowflake and Spark

Bhuvan Malladihalli Shashidhara
msbhuvan@uw.edu

1 Introduction

Machine Learning algorithms are critical to many applications and these algorithms tend to be computationally expensive. Especially as data gets bigger, it takes longer to train machine learning models. Gradient Descent (GD) is an optimization algorithm that is at the core of various supervised machine learning algorithms, where full scan of the data is needed to perform one step (iteration) of gradient update in the training process, and typically models are trained over multiple iterations. Vectorization is very effective in performing a gradient update in GD, but it cannot be used where the complete data cannot fit in memory. Generally, this problem is overcome by using a slightly different version called Mini-Batch Gradient Descent and by utilizing GPUs. However, there are certain advantages of using the Gradient Descent algorithm, hence this project aims to provide a comprehensive comparative analysis of the performance of local and different big data systems in performing Gradient Descent on different sizes of data. This project would enable us to decide the appropriate big data system to consider given the size of the training data for Gradient Descent based Machine Learning algorithms.

In this project, I consider Linear Regression based on Gradient Descent optimization and implement it in the following different systems - Local, Snowflake and Spark on Elastic Map Reduce (EMR). The runtimes in different systems are compared over datasets of different sizes to understand which system is suitable for what size of data, and which system scales well as the size of data increases. I found that Local is suitable only very small datasets, and Snowflake is suitable for larger datasets as it scales very well as data size increases. Also, Snowflake has relatively comparable performance to Local even when the data is small. However, the programming interface of Snowflake is limited to SQL type queries and it can get tricky to implement procedural algorithms with Snowflake.

2 Evaluated Systems

In this project I consider the following systems to be evaluated in a Gradient Descent optimization task for Linear Regression on datasets of different sizes. This section briefly introduces the architecture and programming interface provided by these systems which could be relevant for this project.

2.1 Local

The Local system in this project refers to my personal computer (laptop) which runs on 8GB of primary memory and hosts am Intel i7 processor running at around 2 GHz clock frequency and has 8 hyper-threaded cores (4 physical cores). The system allows for multi-processing, but in this project, I will be running the python programs on a single processor, but the 'numpy' operations optimize by using multi-cores.

In this project, I will be using Python-based implementation which uses vectorized implementation by using 'numpy'. This is feasible in a Local system because the data is made available in the memory which can form matrices for vectorized computations. Such vectorized executions are lead to high

performance on small datasets, but if the data cannot fit in memory then such an execution is not possible in a Local system.

2.2 Snowflake

Snowflake [1] is a novel multi-cluster shared-data architecture by making the software scale elastically over a pool of commodity resources on the cloud. This service is provided by the Snowflake company which makes this a proprietary software. It supports relational databases with SQL transactions. It is elastic and highly-available where storage and compute resources are scaled independently. It uses a proprietary shared-nothing compute and Amazon S3 as a storage service. It uses an efficient query execution engine which is columnar (hybrid like PAX), vectorized and push-based. It uses min-max pruning instead of indexing to search faster across the data. Due to combined effect of columnar storage, optimistic conversion and pruning over semi-structured data, the query performance could be very effective for the gradient updates which is a fundamental operation in Gradient Descent.

Snowflake supports SQL-like queries to carry out the tasks but does not support procedural executions like PL-SQL. However, Snowflake provides a python-connector [2] interface which can be used to execute queries on Snowflake. This enables me to implement iterative procedural algorithms like Gradient Descent. Please note that there might be a cost associated with the interaction between python-connector and Snowflake. In this project, I am using a Snowflake 'small' cluster size.

2.3 Spark

Apache Spark [3] is an open-source system that is built from the inspiration from the Map-Reduce paradigm and Hadoop file systems. It uses in-memory abstractions through Resilient Distributed Datasets (RDD) which optimizes through in-memory computations on large clusters with fault tolerance resulting in significant performance improvement for iterative algorithms. It supports a set of transformations and actions built based on the map-reduce paradigm which can be used to implement complex algorithms through a simple interface. A chain of transformations can be applied to RDDs which form a lineage graph, which is not materialized until an action is called. Users can specify if the RDD needs to be persisted (cached) for re-use. RDDs are immutable which lets a system mitigate slow nodes by running backup copies of slow tasks in MapReduce. Note that the data is stored in partitions across the nodes and are not available in a columnar format as in Snowflake, therefore aggregation tasks on a column of a very large table might be slower on Spark compared to Snowflake.

Spark provides Python API called 'pyspark' which would be used in this project for implementing Gradient Descent for Linear Regression through map-reduce operations on cached RDDs. Note that the Pyspark's native 'mllib' module does not provide Gradient Descent (GD) optimization but provides only Stochastic Gradient Descent optimization (SGD), as SGD is known to run faster than GD in most of the problems. However, in this project, we would be testing Spark on Gradient Descent for Linear Regression. We would be using Amazon's Elastic Map Reduce (EMR) [4] cluster with a master and 2-nodes (m5.xlarge instances) for this project.

3 Problem Statement and Method

3.1 Dataset Description

TPC-H data [5] would be utilized for this analysis. This dataset has been chosen because it is one of the standard datasets used in several benchmarks, and since we have access to this data there would not be any dependency on data providers. Specifically, the plan is to use the 'Lineitem' table where a regression model would be built to predict 'L_Discount' from 'L_Quantity' and 'L_ExtendedPrice'. I use 10 GB version of the data which contains 60 Million rows, and I sample datasets of different sizes from 10M to 50M in steps of 10M rows for comparative analysis. Each of these datasets In order to compare with larger data subsets, the 50M dataset is repeated to create data subsets of size 100M, 150M and 200M rows.

3.2 Objectives

The primary goal of the project is to answer the following questions by comparing the performance of the following systems - Local, Snowflake and Spark on EMR:

- What is the approximate size of the data after which scaling up through big data systems is expected to be effective for Machine learning tasks optimized with Gradient Descent?
- Which system scales better with large datasets? In other words, how do the run-times vary in different platforms with the size of the data?
- Which of these systems are convenient to develop and maintain?

The above questions will be addressed by implementing Linear Regression model as it is one of the most widely used machine learning algorithms for which Gradient Descent is used for optimization. The observations from linear regression would also translate to any other Gradient Descent based optimization algorithms. However, the implementations might be more trickier for other algorithms especially when programming paradigms vary across systems.

3.3 Methodology

This section contains the data preparation details followed by an overview of implementations on local, Snowflake and Spark.

3.3.1 Data Preparation

The data preparation consists of multiple steps from gathering data to having normalized data subsets of different sizes on Amazon S3, as follows:

- **S3 Bucket Creation:** A public bucket was created on Amazon S3 to hold the datasets for the project.
- **Redshift Data Unloading:** Since we require only three columns from the 'Lineitem' table of TPCH-10GB data loaded on Redshift, I ran a query and unloaded its output to an S3 location, which was downloaded to Local.
- **Creating Subset of Difference Sizes:** Shuffled the data and created 5 files having 10M (0.57 GB), 20M (1.10 GB), 30M (1.70 GB), 40M (2.20 GB) and 50M (2.80 GB) rows using bash tools. Note that these files were normalized by using standard scaler (mean centering and scaling by standard deviation) before uploading, and each value was maintained in 'float64' precision (upto 17 decimal points), and each row contained three values - quantity, price and discount, separated by pipes ('|'). Further, I repeated the 50M dataset (after normalization) to create datasets having 100M (5.60 GB), 150M (8.40 GB) and 200M (11.20 GB) rows.
- **Data Ingestion:** The data is residing in Local, and it also has been uploaded to the S3 bucket from which EMR can access directly. But we need to ingest this data for Snowflake. The data-format 'Number' in Snowflake allows only upto 12 decimal points, due to which I had to cast the data while loading into Snowflake. Data ingestion is an additional process required for Snowflake.

3.3.2 Gradient Descent Implementation

We aim to answer the primary question - 'Which platform is suitable for my data having X rows (assuming a small number of columns)?'. Therefore, I consider optimal implementations possible in each of the platforms though it may not be the exact same implementation. For instance, in local, we have the liberty to use memory-based matrix multiplication through 'numpy' operations, but in Spark or SQL (Snowflake), we cannot do matrix multiplications, therefore we need to use either map-reduce operations or SQL-like queries. In all platforms and all data subsets, a common number of "iterations = 100" of gradient updates (with "step-size = 0.01") are made to make it a fair comparison to answer the above question. The nuances of the programming paradigms and the programming interfaces and restrictions of each system have been explained in section 2.

The following provide an overview of the implementation logic and the approach taken for analyzing the runtimes. However, for actual python implementations, please refer to the appendix.

- **Local:** A vectorized implementation which requires all data to be present in memory. This implementation was used, and the runtime for 100 iterations was recorded for each data subset.
- **Snowflake:** A Snowflake-python-connector based implementation that stores the weights for each feature in memory, and get the gradient updates by making SQL queries on Snowflake. I have 2 implementations - one which makes as many SQL calls as the number of features (2 for this dataset) in a single iteration of Gradient Descent (this implementation is referred as 'suboptimal'), and another which makes only one SQL call for each iteration (referred as 'optimal'). Therefore, runtimes were recorded for 100 iterations for both 'suboptimal' and 'optimal' implementations. Also, a third runtime was recorded after repeating the process on a single dataset to assess the runtime when Snowflake uses warm-cache on repeated queries (the least of the three trials was recorded). Runtimes for all the datasets were recorded in these three different settings of Snowflake.
- **Spark:** A map-reduce based implementation which also stores the weights in memory, and uses one map-reduce operation per iteration of Gradient Descent. This implementation is run on all data-subsets and runtimes are recorded.

The implementations were verified by running 5 iterations on 10M dataset on all the systems and observed matching updated weights through Gradient Descent.

4 Results

I discuss the results in quantitative and qualitative terms, where the quantitative aspect would answer the first two questions and the qualitative aspect would answer the third question mentioned section 3.2.

4.1 Quantitative

This section contains the analysis of runtimes observed from the above-mentioned Gradient Descent implementations on Local, Snowflake and Spark.

Firstly, referring to Figure 1 we observe a tremendously slow performance on Spark on EMR, where 100 iterations took over 920 seconds for our smallest dataset having 10M rows. I think this is due to the reduce phase where all the data from all partitions need to be pulled into a single reducer, causing a bottleneck at each iteration. This also may be the reason by Spark does not include Gradient Descent on their standard 'mllib' module but only include Stochastic Gradient Descent. However, Spark may be faster for Stochastic Gradient Descent which avoids such a single reducer bottleneck, which can be tested in future work.

Let us exclude Spark to observe the performance of other systems. Referring to figure 2, we can observe that Local has a linear runtime increase with the increase in the dataset size, but it threw memory error for data sizes of 100M rows and above, which shows that it cannot scale-up for datasets which cannot fit in memory. However, we can observe that Snowflake (suboptimal and optimal) has a higher runtime for smaller datasets than Local, but increases linearly with a slower factor (slope) of increase for larger datasets which indicates that Snowflakes scales better for larger datasets even with a cold cache. We can see that the optimal implementation of Snowflake is much faster than the suboptimal implementation, which indicates that the cost of the python interface is significant as we gained considerable speedup by reducing the python interface calls to half. The improvement of optimal over sub-optimal implementation would be more considerable for datasets having more features.

The reason for the significant faster runtimes observed in Snowflake may be attributed to the query optimizer, and the columnar data storage (where all data in a column are stored in a single partition) which is effective for taking aggregate over the column (we take the sum of 'partial-gradients' column of all rows in an intermediate relation). Snowflake uses a hybrid partitioning which has several advantages of such columnar storage as well. Unlike Snowflake, Spark distributes the tuples across partitions, where a single column is distributed across multiple partitions and nodes, due to which reduce-tasks to compute aggregates are expensive compared to columnar data storage, as the data needs to be pulled by the reducer from multiple nodes, which also has a communication overhead.

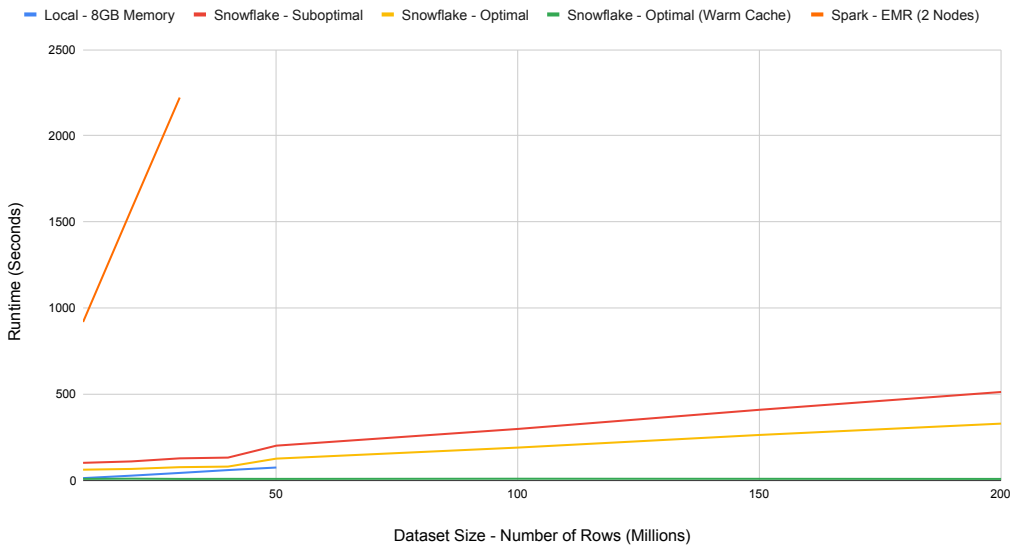

Figure 1: Runtimes for datasets of different sizes to run Gradient Descent for 100 iterations for Linear Regression, implemented in Local, Snowflake and Spark. Observe the relatively much slower Spark runtime.

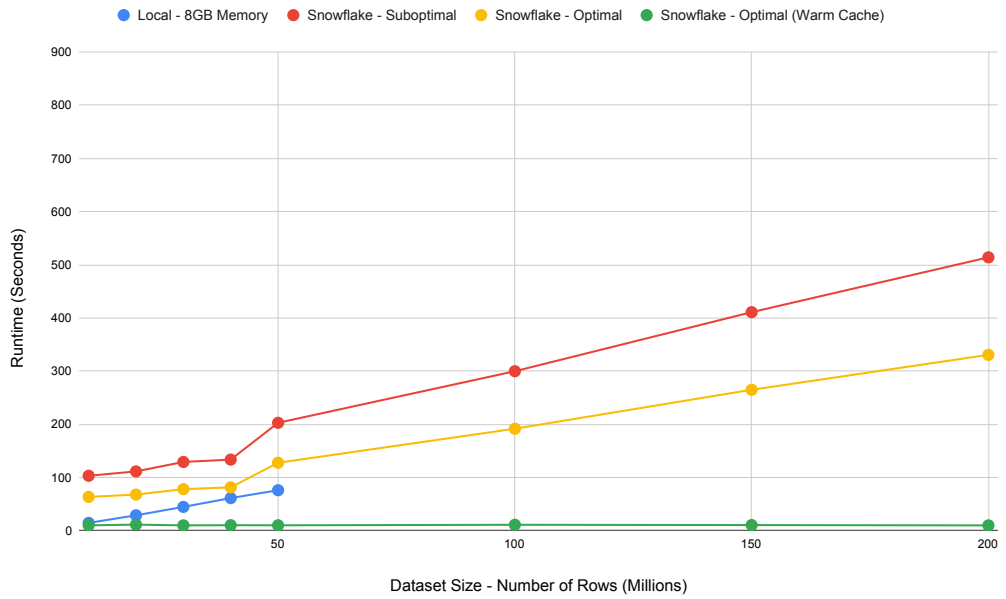

Figure 2: Runtimes for datasets of different sizes to run Gradient Descent for 100 iterations for Linear Regression, implemented in Local and different versions of Snowflake implementation. By excluding the Spark runtimes, we can observe the performance of Local and Snowflake more clearly in this figure.

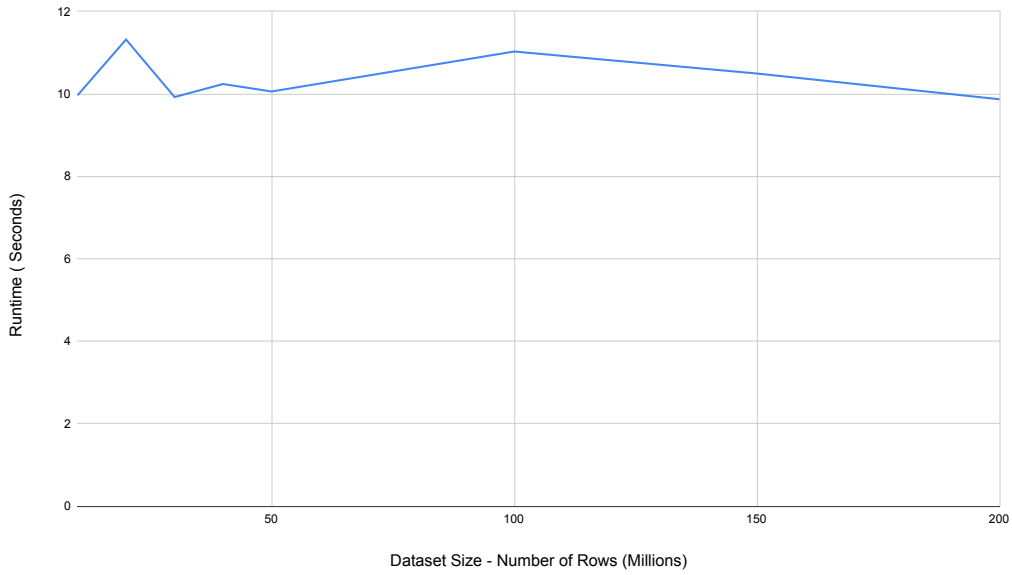

Figure 3: Runtimes for datasets of different sizes to run Gradient Descent for 100 iterations for Linear Regression for Snowflake - Optimal (warm cache). Plotting the Snowflake - Optimal (warm cache) runtime separately enables us to clearly observe its runtime trend.

Interestingly, under warm-cache scenarios, Snowflake with optimal implementation is very efficient having only about 10 seconds runtime for 100 iterations even for 200M dataset. Figure 3 shows a closer look at the warm-cache performance, and we can observe that Snowflake achieves a constant scale-up where the runtime remains almost constant with increase in the dataset size. In this scenario, Snowflake is able to cache this in memory completely to give such a high performance. But this may not be true when data size increases many-fold, in which case, we might have to increase the size of the Snowflake cluster to benefit from the warm-caching feature. This shows that Snowflake's warm cache is appropriate for Gradient Descent optimization tasks which are generally repeated after a certain period of time.

4.2 Qualitative

The implementation on Local seems to be most convenient due to complete access to Python control structures and having a completely shared-memory architecture, however, it doesn't scale well. The Pyspark's programming interface with transformations and actions requires a bit of practice to convert the iterative algorithms into a map-reduce paradigm, but it is relatively easy due to the rich set of transformations and actions provided by Spark. Also, Spark has a huge community that mostly contains answers to usual queries. On the other hand, Snowflake doesn't have a very big community, but their customer support pages contain answers to general queries and have good documentation as well. Snowflake has a very restrictive programming interface requiring us to use SQL-like statements. Especially for iterative procedural function, this is a very limited interface. Though Snowflake-Python-Connector makes it easier to perform operations by using a python interpreter, we have observed that each call to Snowflake through the connector comes with considerable overhead. An easier and efficient programming interface for Snowflake would make it the best of both worlds.

Snowflake also requires data to be first ingested which adds additional one-time task during setup. The plot in figure 4 shows the data ingestion runtimes for datasets of different sizes. And looks like it has a linear increase in runtime for larger datasets, which is acceptable to most use-cases.

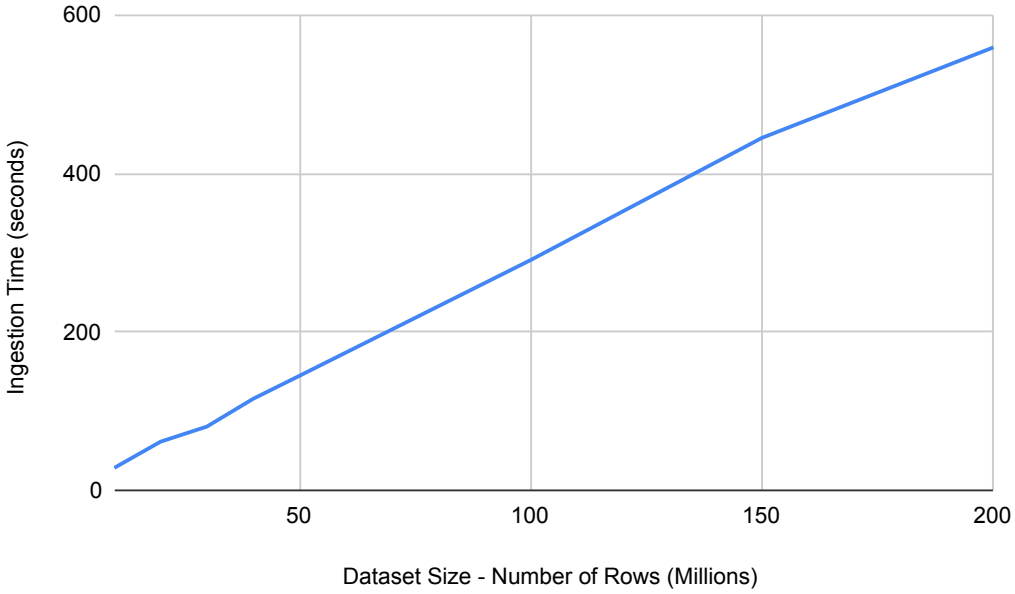

Figure 4: Time taken for data ingestion in Snowflake for datasets of different sizes. This shows that the time taken linearly increases with the data size which is acceptable in most use-cases.

5 Conclusion

Revisiting our research questions stated in the problem statement (see 3.2), we derive the answers from the results we discussed in the previous section.

Firstly, Local is suited only for really small datasets for which the relative-overhead of a big-data system is significant. For larger datasets, big data systems like Snowflake seem to be a better choice for Gradient Descent for Linear Regression, especially when we can utilize the warm-caching feature through repeated Gradient Descent optimizations.

Secondly, Snowflake turns out to be highly scalable providing almost constant scale-up with increased data sizes when warm-cache is utilized, and a linear increase in runtime with the increased dataset size in the cold-cache scenario, but with a lower factor (slope) of runtime increase compared to the factor (slope) observed in Local.

Thirdly, from the qualitative aspect, the implementation of Gradient descent and other complex algorithms are simpler on Local python due to the access to a shared memory. Spark supports a wide range of transformations and actions based on the map-reduce paradigm, which enable us to implement even complex algorithms in a fairly simple code. However, as observed Local and Spark do not scale well compared to Snowflake. Since Snowflake has a restrictive SQL-like interface, it is not straight forward to implement complex algorithms with simple SQL queries. Also, Snowflake requires a separate data ingestion step, but it is acceptable due to linear runtimes observed in data ingestion.

For more complex queries than the Gradient Descent, the Snowflake might even be slower due to queries which may be inefficient compared to Local or Spark, but for Gradient Descent on Linear Regression Snowflake turns out to be a great choice.

In future research studies in this area, optimized configurations of Spark can be considered along with a variety of objective functions for Gradient Descent to test the generalizable scalability of the systems across a set of optimization tasks. Revisiting our research questions stated in the problem statement (see 3.2), we derive the answers from the results we discussed in the previous section.

References

[1] Dageville, Benoit, et al. "The snowflake elastic data warehouse." Proceedings of the 2016 International Conference on Management of Data. ACM, 2016.

[2] Snowflake Connector for Python. 2019. Web. URL: https://docs.snowflake.net/manuals/user-guide/python-connector.html.

[3] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[4] Amazon EMR. 2019. Web. URL: https://aws.amazon.com/emr/.

[5] TPCH. 2019. Web. URL: http://www.tpc.org/tpch/.

6 Appendix

The appendix contains the actual python implementations on Local, Snowflake (optimal) and Spark.

6.1 Local Implementation

```
def _objective_func(X, y, beta):
    n = X.shape[1]
    residual = y   np.matmul(X.T, beta)
    return ((1/n) * sum(residual ** 2))

def _computegrad(X, y, beta):
    n = X.shape[1]
    residual = y   np.matmul(X.T, beta)
    grad = ( 2/n) * np.matmul(X, residual)
    return grad

def gradient_descent(X, y, beta_0=None, step_size=0.01, max_iter=100):
    if beta_0 is None:
        beta_0 = np.zeros((X.shape[0], 1))

    beta = beta_0
    for i in range(max_iter):
        beta  = step_size * _computegrad(X, y, beta)
    return beta
```

6.2 Snowflake Implementation

```
def gradient_descent(data_table, feature_columns, y_column,
                     step_size=0.01, max_iter=100):
    # Optimal Implementation.
    num_datapoints = int(execute_query(conn, "select␣count(*)␣from␣%s"
                                       % data_table)[0][0])
    num_features = len(feature_columns)

    weights = [0] * num_features
    for t in range(max_iter):
        # Currently, it is just 2 features, hence we can hardcode
        # the query instead of building it from feature names.
        cur_w_0, cur_w_1 = weights
        cur_sql = (
          """
          select %s   (%s * cur_avg_grad_0) as updated_weight_0,
                 %s   (%s * cur_avg_grad_1) as updated_weight_1
          from (
            select sum(partial_grads_0) / %s as cur_avg_grad_0,
                   sum(partial_grads_1) / %s as cur_avg_grad_1
```

```
            from (
              select (%s    ((%s * %s) + (%s * %s))) * ( 2 * %s)
                        as partial_grads_0,
                     (%s    ((%s * %s) + (%s * %s))) * ( 2 * %s)
                        as partial_grads_1
              from %s
            )
          );
          """ % (cur_w_0, step_size,
                 cur_w_1, step_size,
                 num_datapoints,
                 num_datapoints,
                 y_column, feature_columns[0],
                   cur_w_0, feature_columns[1],
                   cur_w_1, feature_columns[0],
                 y_column, feature_columns[0],
                   cur_w_0, feature_columns[1],
                   cur_w_1, feature_columns[1],
                 data_table)
        )
        cur_res = execute_query(conn, cur_sql)
        new_weight_1,= float(cur_res[0][0]),
        new_weight_2 = float(cur_res[0][1])
        weights = [new_weight_1, new_weight_2]

    return weights
```

6.3 Spark Implementation

```
def gradient_descent(data_rdd, num_datapoints, num_features,
                     step_size=0.01, max_iter=100):
    def _compute_partial_grad(features_y):
        # This assumes the number of features are small enough to
        # fit in executor memory.
        # Also, it assumes that the y_column is at the end.
        features, y = features_y
        residual = y   sum([weights[d_i] * features[d_i]
                            for d_i in range(num_features)])
        partial_grads = [residual * ( 2 * features[d_i])
                         for d_i in range(num_features)]
        return tuple(partial_grads)

    data_rdd.cache()
    weights = [0] * num_features
    for t in range(max_iter):
        partial_grads = data_rdd.map(_compute_partial_grad)
        sum_grads = partial_grads.reduce(
          lambda a, b: tuple([a[d_i] + b[d_i] \
                              for d_i in range(num_features)])
        )
        for d_i in range(num_features):
            weights[d_i] = weights[d_i]
                           (step_size *
                               (float(sum_grads[d_i])/num_datapoints))

    return weights
```