Speed Up Query Execution By Pushdown in AsterixDB

Yue Gong
Department of Computer Science and Engineering
Southern University of Science and Technology
Email: yvetteyue1998@outlook.com

Wail Alkowaileet
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Email: wael.y.k@gmail.com

1. Abstract

In the original query execution model of AsterixDB, incomplete pushdown of query plan will bring significant memory copy costs between operators. To tackle this problem, we explore three patterns of queries on which the complete pushdown can have an up to 34.6% speed-up. More specifically, we push execution logic down to the data-scan stage as much as possible to reduce the overall number of operators.

2. Introduction

AsterixDB is a Big Data Management System (BDMS). Its query execution engine adopts the iterator-style model. To evaluate a query, the engine will compile it into a series of operators(e.g. aggregation, filter, projection). Each operator will implement a *next()* method to produce intermediate results to the upper operator or the final results. The whole process is illustrated in figure 1.

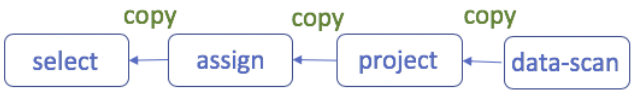

Figure 1. iterator-style model

The Iterator-style model provides great programming flexibility since it decouples operators. Thus, we can implement any query by combining these operators. However, the iterator-style model is clearly not CPU-friendly. Since *next()* is a virtual function, calling it again and again brings heavy overhead. As long as IO is the bound, it works fine. However, as storage devices become faster and faster today, utilizing CPU better shows more and more importance.[1]

AsterixDB has made some progress to improve the traditional volcano iterator model. Instead of tuple-at-a-time implementation, it introduces the concept of frame which serves as the unit of data to be passed between operators. By the batch-at-a-time implementation, it reduces the time of calling *next()* functions.[2]

But can we make it further? The answer is YES. Besides using batch-processing, reducing the number of operators can also decrease the time of calling virtual functions.

To accomplish the goal of reducing the number of operators, we develop a method called **pushdown** which aims to combine operators for queries matching certain patterns. By reducing the number of operators, the performance of query execution can be improved significantly.

3. Methodology

For queries with certain patterns, there are also specific patterns in the query plan. Thus, by detecting different patterns in the query plan, we can recognize different queries and rewrite them to the optimum. We mainly explore three kinds of query patterns to perform the **pushdown** method.

3.1. Pattern 1: Query with Field-access and Filter

The first pattern is for query with field-access(e.g. tweets.user) and Filter. For this kind of query, we will push both the filter and field accesses to the data-scan stage.

```
SELECT Attribute
FROM dataset
WHERE condition
```

Figure 2. query pattern 1

Here is a query which satisfies pattern 1. And the query plan before pushdown is illustrated in figure4.

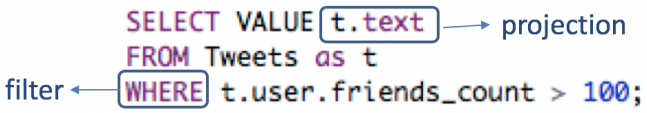

Figure 3. query pattern 1 example

In the original query plan, *data-scan()* only does one thing which is to scan the data-set Tweets and pass the data to upper layer. However, it is clear that we can do much more than this in the data-scan stage. Why not do filter and field-access at the same time of scanning the data.

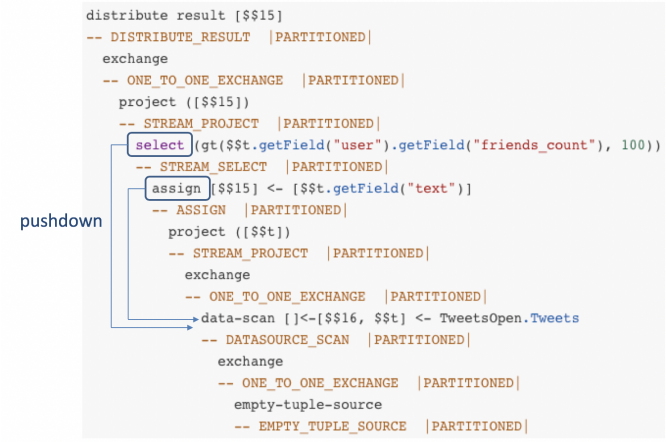

Figure 4. query plan before

The query plan after pushing down filter and field-access is illustrated in figure 5. This time, *data-scan()* performs all most all the logic in the query plan. When only looking at the *data-scan()* stage, the logic is quite like hand-written code which is commented as pseudo-code in figure 5.

Figure 5. query plan after push down

In addition, the power of pushing down filter and field-access is actually more general than the pattern 1 shows. For any complex query plan with filter and field-access, the filter part and field-access part can be pushed down to *data-scan()* stage, which will avoid the tedious *assign()*, *project()* chain in the query plan.

3.2. Pattern 2: Query with Quantifier

The second pattern is for queries with a quantifier (SOME/EVERY).

```
SELECT Attribute
FROM dataset
WHERE SOME/EVERY item IN array SATISFIES condition
```

Figure 6. query pattern 2

Figure 7 shows an example of query pattern 2. *t.entities.hashtags* in the example is an array of objects illustrated in figure 8.

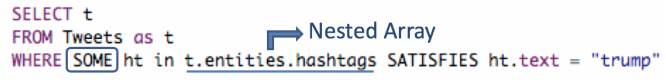

Figure 7. query pattern 2 example

Figure 8. hashtag structure

Since the *hashtags* is still an array, we cannot directly apply the condition to it. For this kind of query, asterixDB will generate a long subplan in the query plan to unnest the array and process each object in the array one by one like the figure 9.1 shows. What we will do is to put the entire subplan logic into the *data-scan* stage.

The new data-scan stage is illustrated in figure 9.2. When a tweet is read, we will first get its *hashtags*. We implement a new function called *for-each* to access each object inside the *hashtags* and evaluate the condition at the same time.

Figure 9.1 subplan for query with quantifier

Figure 9.2 pseudocode for data-scan

Figure 9.3 data-scan after pushdown

Figure 9. query plan of pattern 2 example

3.3. Pattern 3: Query with Aggregation

The last pattern is for query with aggregation.

```
SELECT count(*)
FROM dataset
WHERE condition
```

Figure 10. query pattern 3

For this kind of query, we will push the count function to data-scan, which means we will count the number while scanning the data.

4. Empirical Verification

4.1. Dataset

- **Tweets data**: Each tweet is about 3.78M

4.2. Test Environment

- **OS**: Linux
- **Processor**: i7-7567U CPU @ 3.50GHz, 2 cores
- **Memory Size**: 15GB

4.3. Benchmark for Pattern 1

The first benchmark will only test the performance of pushing down **filter**. The benchmark is as follows. We increase the selectivity from test 1 to test 4.

```
USE TweetsOpen;
SELECT count(*)
FROM Tweets as t
WHERE t.user.friends_count > 100  Test 1
                              1000 Test 2
                             10000 Test 3
                            100000 Test 4
```

Figure 11. benchmark for pushing down filter

The result is illustrated in figure 12. For 70G data in 1 partition, we get 8.56% speed-up in average; For 70G data in 2 partitions, we get 3.37% speed-up in average, which is relatively mild.

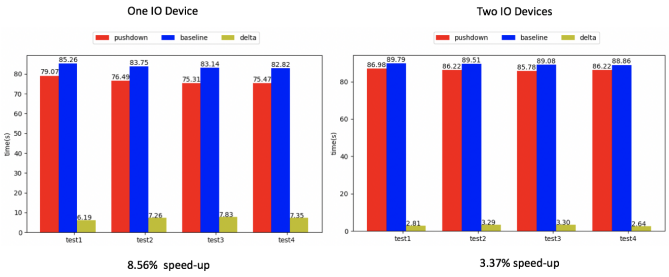

Figure 12. results of pushing down filter

The second benchmark will only test the performance of pushing down **field-access**. Figure 13 shows the benchmark. We test large-field access, small-field access and nested-field access.

We get slightly better speed-up ratio as the figure 14 shows this time. And there is not much difference between different kinds of field-accesses.

Figure 13. benchmark for pushing down field-access

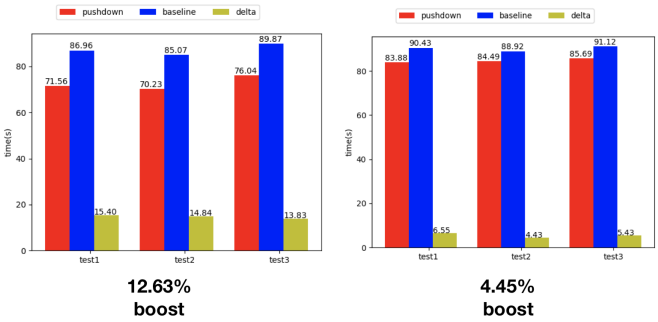

Figure 14. results of pushing down field-access

Third, we test queries with both filter and one field-access as the figure 15 shows. This time, we get better speed-up ratio by adding the power of pushing down filter and pushing down field-access which is illustrated in figure 16.

```
SELECT count(t.text)
FROM Tweets as t
WHERE t.user.friends_count > 100  Test 1
                              1000 Test 2
                             10000 Test 3
                            100000 Test 4
```

Figure 15. benchmark for pushing down both filter and field-access

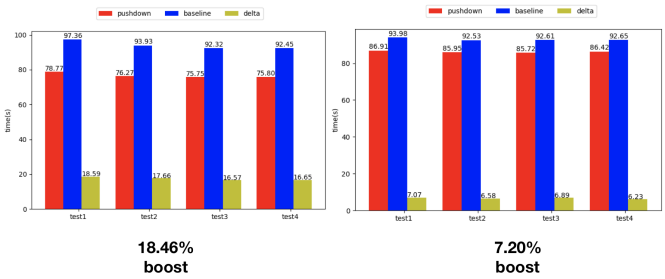

Figure 16. results of pushing down both filter and field-access

At last, we test queries with a filter and two field-accesses which means there will be larger amount of data that we will push down. The benchmark is illustrated in figure 17.

The speed-up is significant this time. As the figure 18 shows, for 70G data in one partition, we get a 28.10% speed-up; For 70G data in two partitions, we get a 11.61% speed-up. This is actually much better than our last benchmark. Why does that happen? It is because we only save one field passing time in the last benchmark while we save two fields passing time in this benchmark. The better speed-

```
SELECT count(t.user.description), count(t.user.followers_count)
FROM Tweets as t
WHERE t.user.friends_count > 100  Test 1
                                  1000 Test 2
                                  10000 Test 3
                                  100000 Test 4
```

Figure 17. bechmark for pushing down both filter and field-access 2

up ration when it moves to two field-accesses also proves our motivation that memory copy of data is an expensive operation.

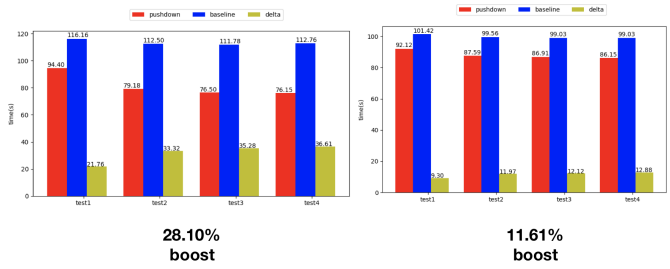

Figure 18. results of pushing down both filter and field-access 2

4.4. Benchmark for Pattern 2

To test the performance of pushdown for queries with a quantifier, we will apply the benchmark as figure 19 shows.

```
SELECT count(*)
FROM Tweets as t
WHERE some ht in t.entities.hashtags SATISFIES ht.text = "trump"
```

```
SELECT count(*)
FROM Tweets as t
WHERE every ht in t.entities.hashtags SATISFIES ht.text = "trump"
```

Figure 19. benchmark for query pattern 2

The result is shown in figure 20. Note that we add a benchmark query:

SELECT count(t.entities.hashtags) FROM Tweets as t

denoted as *io* in the result graph. The benchmark query aims to measure the time of query IO plus the time to access the hashtags attribute. This time serves as a bottom-line of our optimization since it is the time we must spend and cannot be accelerated.

As figure 20 shows, for 70G data in one partition, we can get 34.6% speed-up for *SOME* and 31.3% speed-up for EVERY in average; for 70G data in two partitions, we can get 18.7% speed-up for *SOME* and 19.1% speed-up for EVERY in average.

The result seems great, but we also get a strange observation – Why moving to two partitions always bring a worse

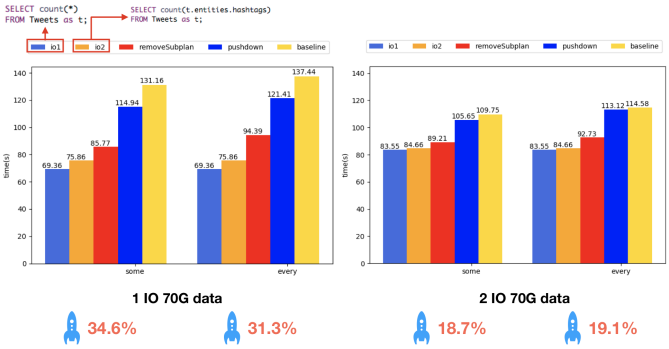

Figure 20. results of pushing down subplan

performance? More specifically, why does 2-partitions always hurt the performance of pushdown but benefit the performance of baseline implementation?

It is because although multiple partitions will utilize CPU better, it brings competition to IO (Partitions share the same buffer cache and memory budget). Since pushdown has done quite well on the CPU, it is IO-bounded. However, the original implementation is CPU-bounded. Thus, the IO-bounded pushdown will be hurt because of the competition between different partitions while the original implementation will benefit because of the better utilization of CPU.

4.5. Benchmark for Pattern 3

For query pattern 3, we adopt a quite simple benchmark as figure 21 shows. And we test lots of new ideas on this simple benchmark.

First, we pushdown the count function to the *data-scan* stage, which means we count the record when scanning the data. It is denoted as **pushdownCount** in the figure 22. Besides we also want to see the evaluator-creation overhead (AsterixDB creates an evaluator to evaluate each expression). The first attempt is to substitute asterixDB *greaterThan* evaluator by pure JAVA $>$.It is denoted as **removeGt** in the figure 22. The second attempt is to push down the count function, which means we count records while scanning the data. It is denoted as **pushCount** in the figure 22. In additional, we add a baseline query called **pureCount**:

SELECT count(*) FROM Tweets

to evaluate the base time we need to consume.

The results in figure 22 gives us lots of insight. First of all, **removeGt** seems to have little improvement but **pushCount** can give a close performance to **pureCount**. Thus, the evaluator overhead for Asterix *largerThan* is mild. Second, our **pushdownCount** makes sense since its time is really close to the **pureCount**.

5. Conclusion

To reduce the expensive memory copy cost between different operators, we develop a method called pushdown to

```
SELECT count(*)
FROM Tweets as t
WHERE t.user.friends_count > 100;
```

Figure 21. benchmark of pushdown count

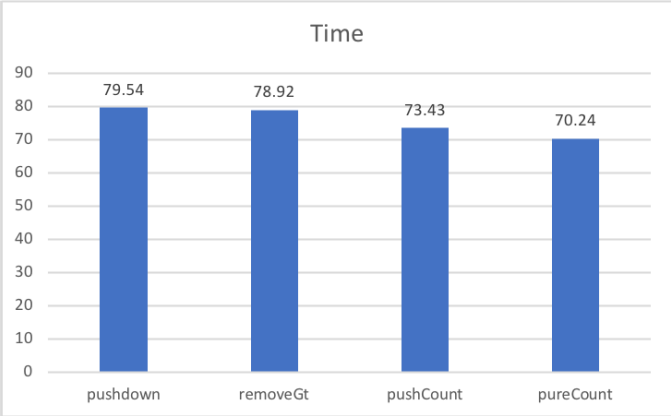

Figure 22. results of pushdown count

combine multiple operators. We define three query patterns and apply specific pushdown strategies to them. Pushdown method can achieve at most 34.6% speed-up, which is very significant. It's worth mentioning that pushdown has made the execution time very close to the pure IO time. If we want to optimize the time further, we need to reduce either the field evaluation time or the IO cost.

References

[1] Agarwal, S., Liu, D. and Xin, R. (2019). Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop. [online] Databricks. Available at: https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html [Accessed 27 Aug. 2019].

[2] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In ICDE, pages 1151–1162, 2011.