# Speed Up Query Execution By Pushdown In AsterixDB

Yue Gong [1]; Wail Alkowaileet[2]; Michael Carey[2]
[1]Southern University of Science and Technology, [2]Univeristy of California, Irvine

## Abstract

AsterixDB is a Big Data Management System (BDMS). To evaluate a query, the compiler will compile it into a series of operators(e.g. aggregation, filter, projection). Then, the query execution engine(hyracks) will execute each operator from bottom to top and the data will move between operators. The whole process is illustrated in figure 1.

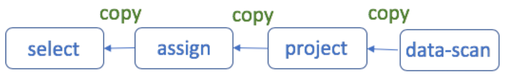



**Figure 1.** dataflow example

This implementation provides great programming flexibility since it decouples operators. Thus, we can implement any query by combining these operators. However, it also brings performance issues since there are expensive memory copies between operators and expensive virtual function calls [1].
To tackle this problem, we develop a method called **pushdown** to combine several operators for queries matching certain patterns. More specifically, we push execution logic down to the data-scan stage as much as possible to reduce the number of operators. By applying pushdown, the performance of query execution can be improved significantly.

## Introduction

```
SELECT VALUE t.text
FROM Tweets as t
WHERE t.user.friends_count > 100;
```

**Figure 2.** query with projection and selection

Before

```
distribute result [$$15]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    project ([$$15])
    -- STREAM_PROJECT  |PARTITIONED|
      select (gt($$t.getField("user").getField("friends_count"), 100))
      -- STREAM_SELECT  |PARTITIONED|
        assign [$$15] <- [$$t.getField("text")]
        -- ASSIGN  |PARTITIONED|
          project ([$$t])
          -- STREAM_PROJECT  |PARTITIONED|
            exchange
            -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
              data-scan []<-[$$16, $$t] <- TweetsOpen.Tweets
              -- DATASOURCE_SCAN  |PARTITIONED|
                exchange
                -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
                  empty-tuple-source
                  -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

pushdown

**Figure 3.** query plan before

After

```
distribute result [$$15]
-- DISTRIBUTE_RESULT  |PARTITIONED|
  exchange
  -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
    data-scan []<-[$$16, $$t] <- TweetsOpen.Tweets
    condition (gt($$t.getField("user").getField("friends_count"), 100))
    project ($$t.getField("text"))
    -- DATASOURCE_SCAN  |PARTITIONED|
      exchange
      -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
        empty-tuple-source
        -- EMPTY_TUPLE_SOURCE  |PARTITIONED|
```

```
for t in tweets:
  if t.user.friends_count > 100:
    project(t.text)
```

**Figure 3.** query plan after pushdown

## Methods

We mainly explore three kinds of query patterns to perform pushdown.
The first pattern is mentioned in the introduction. For queries with projection and filter, we will push filter and project operator down to the data-scan.

```
SELECT VALUE t.text        → projection
FROM Tweets as t
WHERE t.user.friends_count > 100;   ← filter
```

**Figure 4.** query pattern one

The second pattern is for queries with quantifier.

```
SELECT t
FROM Tweets as t
WHERE SOME ht in t.entities.hashtags SATISFIES ht.text = "trump"
                 → Nested Array
```

**Figure 5.** query pattern two

For this kind of query, asterixDB used to generate a long subplan in the query plan. Since t.entities.hashtags is a array of hashtag object, AsterixDB uses a subplan to unnest the array and process each object inside the array one by one. To avoid expensive memory copy cost, we will push all the subplan down to data-scan.

```
subplan {
      aggregate [$$24] <- [non-empty-stream()]
        -- AGGREGATE  |LOCAL|
        select (eq($$29, "trump"))
          -- STREAM_SELECT  |LOCAL|
          assign [$$29] <- [$$ht.getField("text")]
            -- ASSIGN  |LOCAL|
            unnest $$ht <- scan-collection($$28)
              -- UNNEST  |LOCAL|
              nested tuple source
                -- NESTED_TUPLE_SOURCE  |LOCAL|
      }
-- SUBPLAN  |PARTITIONED|
    assign [$$28] <- [$$t.getField("entities").getField("hashtags")]
      -- ASSIGN  |PARTITIONED|
      project ([$$t])
        -- STREAM_PROJECT  |PARTITIONED|
        exchange
          -- ONE_TO_ONE_EXCHANGE  |PARTITIONED|
          data-scan []<-[$$27, $$t] <- TweetsOpen.Tweets
            -- DATASOURCE_SCAN  |PARTITIONED|
```

**Figure 6.** subplan for query with quantifier

```
boolean_arr =  []
for hashtag in hashtags:
  boolean = (hashtag.get("text") == "trump")
  boolean_arr.append(boolean)
for boolean in boolean_arr:
  if quantifier == "SOME":
    return true when meeting one true value
  if quantifier == "EVERY":
    return false when meeting one false value
```

**Figure 8.** pseudocode for data-scan

```
data-scan [$$t]<-[$$27, $$t] <- TweetsOpen.Tweets
condition (foreach($$t.getField("entities").getField("hashtags"),
                   eq(item-accessor().getField("text"), "trump")))
project ($$t)
```

**Figure 7.** data-scan after pushdown

The third pattern is queries with count function. For this kind of query, we will push the count function to the data-scan, which means we will count the number while scanning the data.

## Results

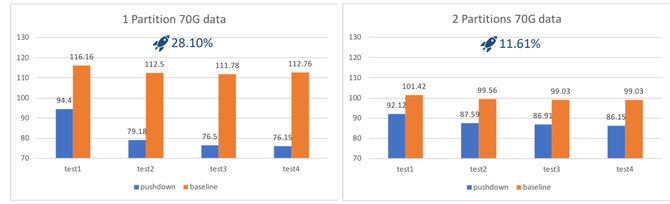



**Figure 9.** benchmark for pushdown selection and projection

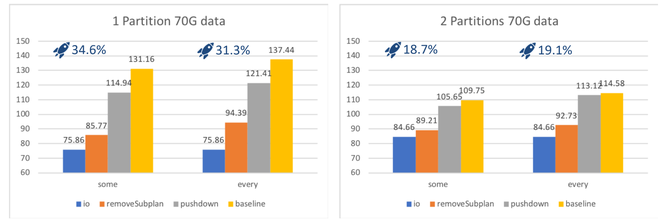
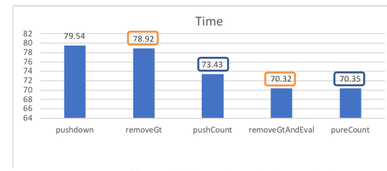



**Figure 10.** benchmark for pushdown subplan




**observation 1**
pushdown count is very close to pureCount

**observation 2**
Evaluate the field access is the performance bottleneck

**Figure 11.** benchmark for pushdown count

## Discussion

As we can see from the results, pushdown can improve the performance of queries significantly. There is an interesting observation that 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. 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.

## Conclusions

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

## Contact

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

## 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].

© Poster Template by Genigraphics® 1.800.790.4001 www.genigraphics.com