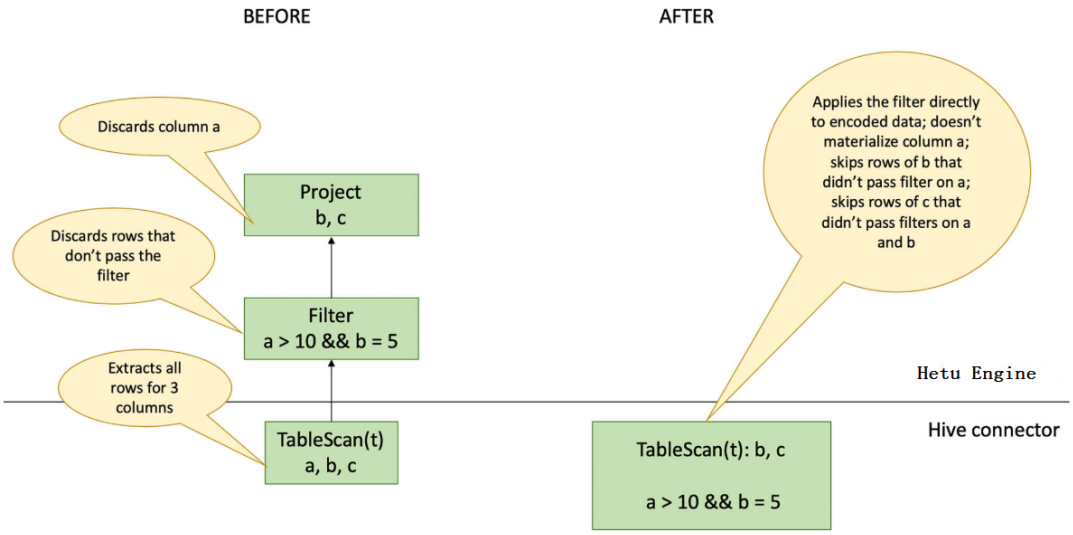
Nearly 60 percent of our global Hetu Queries CPU time is attributed to table scan, so with making scan improvement, many queries will get performance improvement. Currently we focus on table scan optimization for table stored in ORC format (but should be possible to extend to other data format also).

Scan Optimization

Overall approach can be depicted as in the below picture. As can be seen, in the new approach filter evaluation has been pushed to HIVE connector from engine.



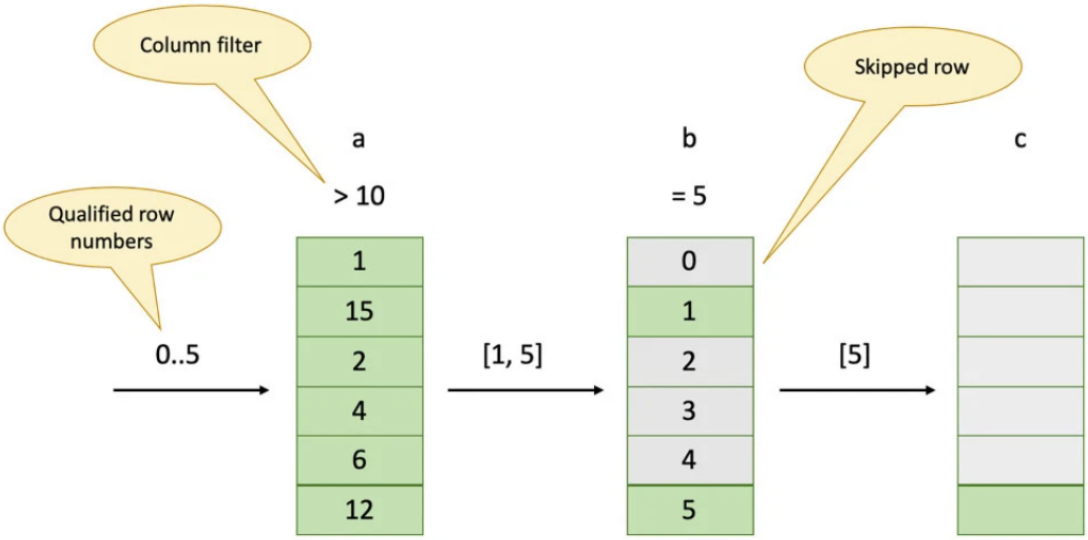
To start with we have chosen below two idea to optimize table scan by reducing CPU time. Design of both of the below optimization will be presented together in subsequent section:

1. Filter Pushdown

This optimization is required to push the filter from engine to hive connector so that only filtered rows/column gets returned back to engine layer. This in itself does not give much benefit but it sets the foundation layer for further optimizer related to filter.

2. Efficient row skipping (Selective filter)

Currently if a filter on one column matches only for small set of rows, then still we read all values in subsequent columns which are part of the query and immediately discard them. This waste CPU cycle unnecessarily. This can be optimized to read only rows for next columns which matched as part of the previous column. So each scan of columns after applying filter will result in list of row id and only according to those next column value will be read. Note that the first columns scan will still read all rows before applying the filter. Below is an illustration of scan for a query `SELECT b, c FROM t WHERE a > 10 AND b = 5 query`.



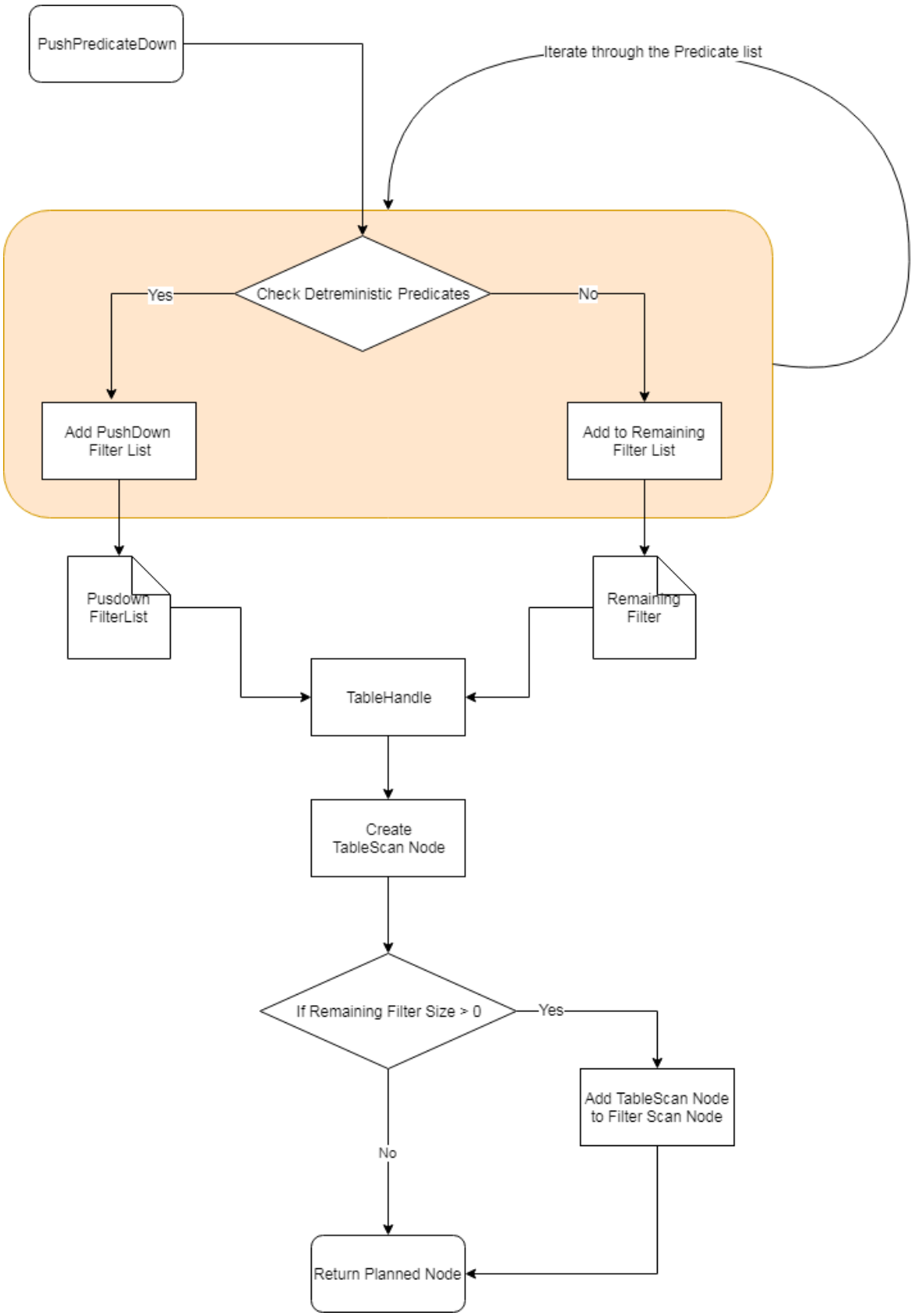
Here input to first column "a" scan is all row number (0 to 5), then after applying filter on column "a", input to scanning column "b" is only list of rows which matched in earlier filter. So "b" has to only fetch for row 1 and 5 and then finally column "c" has to fetch for only row 5. Rest all rows are discarded.

Design

NOTE: Currently only simple (deterministic) filter expression (e.g. a=10, a>10, a in (1,2) etc) will be supported.

In order to push the filter down to connector layer, Hetu optimizer will create a rule to convert the plan with ScanFilterProject being converted to TableScan. Also filter, columns involved in filter (along with other required details) will be passed to the connector as part of HiveTableHandle. Details of Optimizer design will be explained separately.

Optimizer:

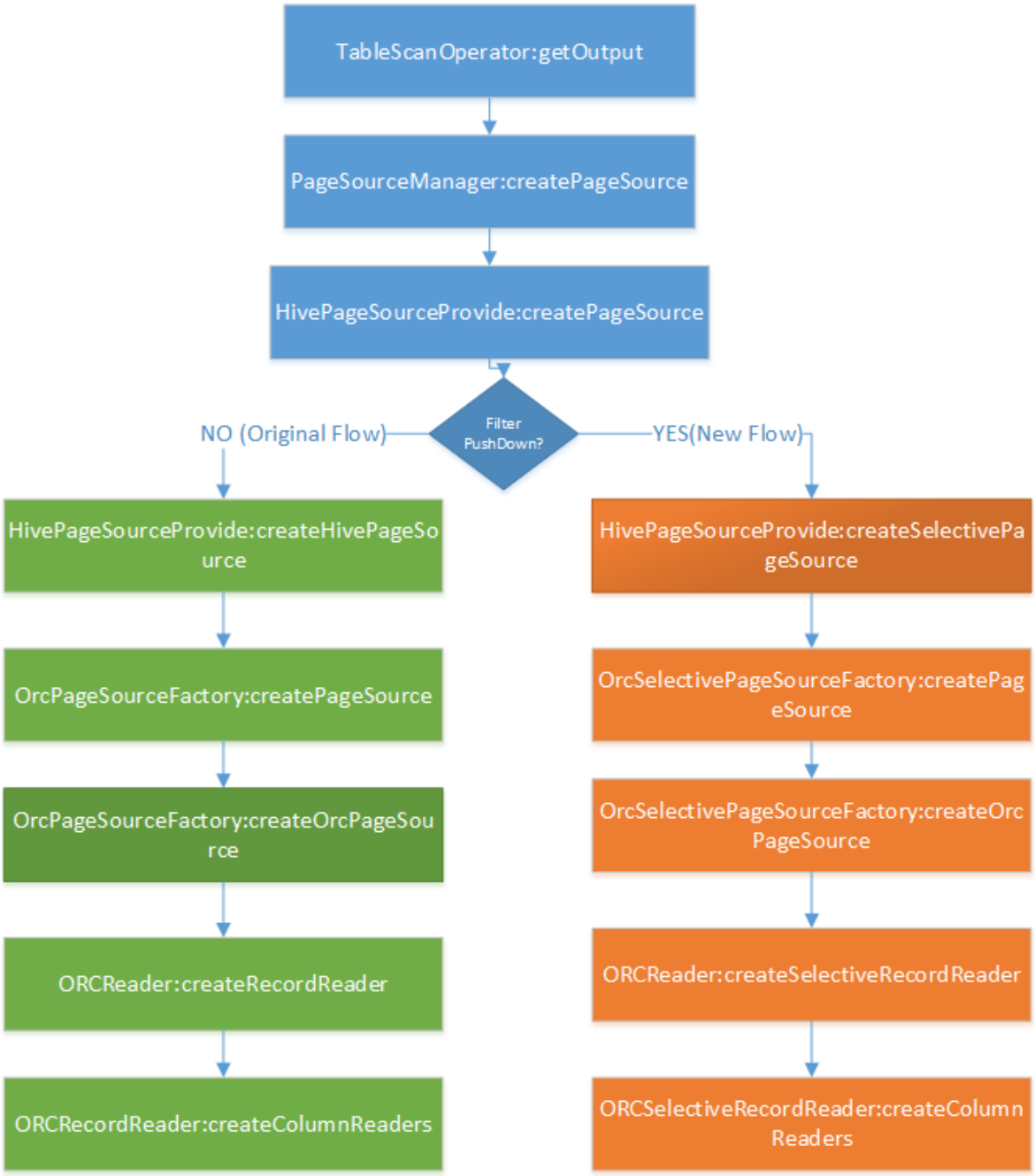


1. Expose Interface from connector metadata; and let decision for allowing specific pushdown for a given predicate to the connector itself.
2. If connector indicate true; identify predicate subset which can be pushed down; segregate the same in separate lists. i.e. Pushdown Predicates and remaining predicates.
3. If no remaining predicates then create a tableScan node and return from the rule.
4. Else, place the tablescan node created with Pushdown predicates; and pass the remaining with in Filter Node and return the FilterScan node as rewritten node.
5. In-case predicate pushed down, consider the same while calculating the stats for the corresponding table.

Page Source Initialization

Once the filter is available in HiveTableHandle, the same will be used as part of create page source to push filter down till ColumnReader.

Also in order to create the flow of Selective reading of column, a separate parallel page source, page source factory, column reader etc will be created. Depending on whether filter is pushed-down or not either actual reader path will be invoked or Selective reader as explain below:



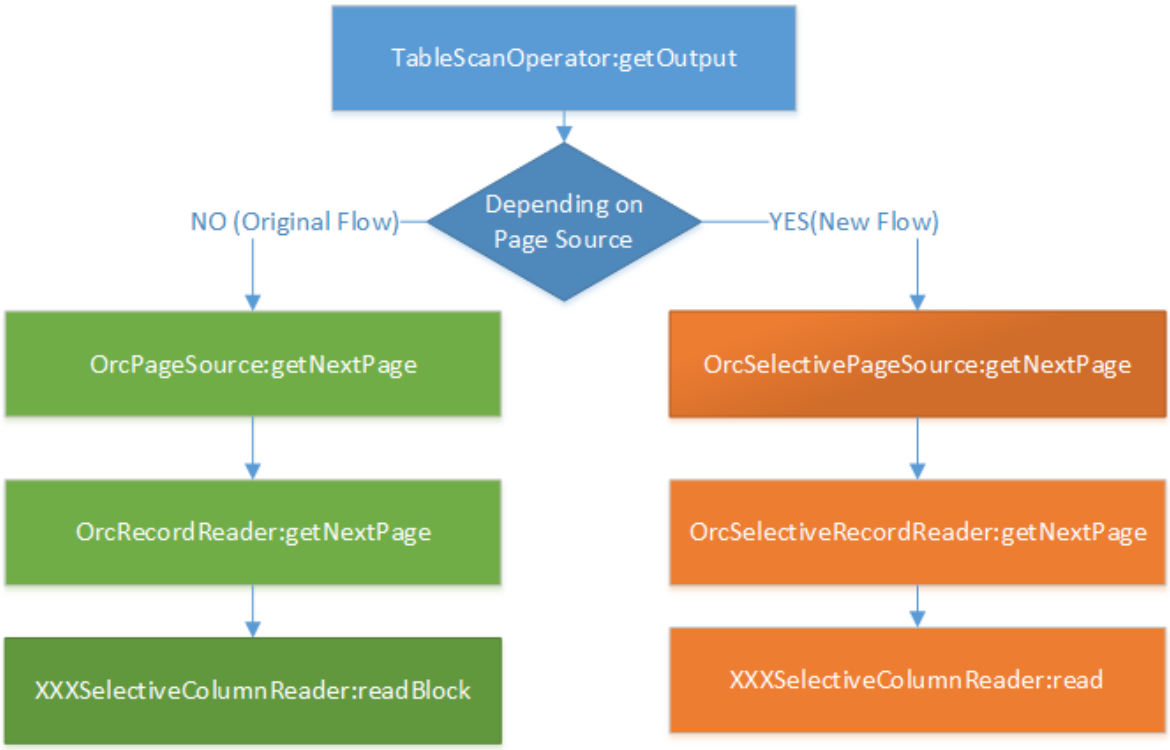
As shown in above diagram, left side flow (in green color) is the original flow to create page source, whereas left side (in pink color) is the new flow. Functionality of all steps remains same but at each stage it requires to pass the additional argument in order to handle filter.

> Except in the ORCSelectiveRecordReader:createColumnReaders, Like original flow, it will create column readers for all columns but it will requires below additional arguments in order to handle it:
>
> a. List of output columns (exclude column only with predicate)
>
> b. List of filters column wise.

> All these parameters will be sent from top level.
>
> Unlike normal column reader (created earlier), the newly created column readers will have following additional details:
>
> a. Whether column is part of projection.
>
> b. Whether filter on this column.
>
> c. List of row positions to read.
>
> d. Number of positions to read.
>
> Last 2 parameters will be explained as part of execution.

Getting Page

Once the page source is initialized, it will fetch data from the corresponding page source. So in our case, Selective Page would have been initialized, then it will call getNextPage of this page source in order to read the data. The overall flow to read data is as below.



There are considerable logic change (in last 2 function i.e. OrcSelectiveRecordReader:getNextPage and XXXSelectiveColumnReader:read) here in order to read selective columns as explained below:

1. First column is read,

   a. if there is any filter on this column then filter is applied otherwise this value will be directly selected.

   b. If this column is part of projection, then its values will be stored.

   c. Store the position of row and also increment position count.

2. Next column if any will be read in same way as in (1) but it will have input as positionsToRead[] and positionCount from previous column read. This column will be read only from the given row position. Remaining logic of (1) will be applied as it is.

3. Any subsequent column (if any) will be handles as mentioned in (2).

Once all these columns are read, then whet-ever resultant positionToRead (along with count) returned only corresponding to those position value for each column will be considered to make Block for page.

```
e.g. Suppose
column-1 read from position [1,5,10,20] with position count as 4.
Column-2 read from position [1,10] with position count as 2.
Then finally it will create Block for page with position count as 2 and values
corresponding to row position [1,10].
```

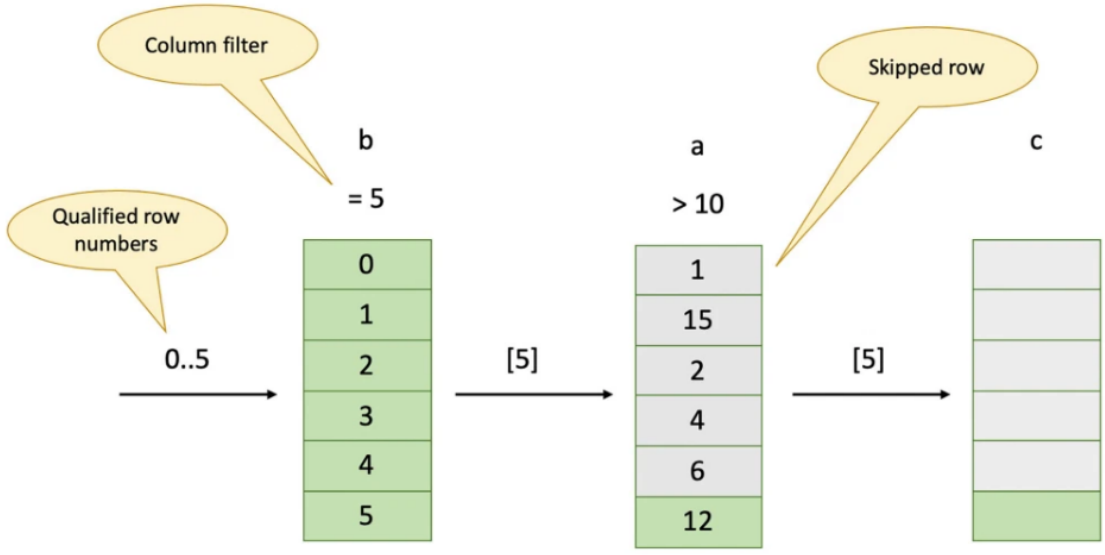
Also it will create page block from only projection columns and hence unwanted data wont be sent back.

Configuration:

We will require a session configuration to enable/disable filter pushdown. By default we will keep it as "**false**". Tentative name of the configuration variable is "**orc_predicate_pushdow_enabled**".

3. Filter Re-ordering:

Some column filter results in relatively less qualified rows compared to other filter (or column without filter). Since as part of Efficient Row skipping functionality whatever rows result from applying filter, only on those subsequent filter will be applied. So it is better to apply filter first which result in lesser rows.



So applying filter "b=5" result in just one row compared to filter a>10 which result would have resulted in 2 rows. Also fetching first "c" would have resulted in 6 rows. So it is better to fetch column "b" first and apply filter "b=5".

Design:

As per the current scope of implementation, we will only re-order scan of columns with and without filter i.e. any columns with filter will be scanned first and then column with filter. Multiple columns with the filter will be given same priority.

This re-ordering will be implemented by maintaining the column index of the Column reader with and without filter separately. Then while reading the page (**getNextPage**), first column reader with filter will be called and then without column reader. Below pseudo code will help to understand:

```
for (Integer idx : colReaderWithFilter) {
    positionCount = columnReaders[idx].read(getNextRowInGroup(),
positionsToRead,       positionCount);
    if (positionCount == 0) {
    break;
    }

    // Get list of row position to read for the next column. Output of positions
from
    // current column is input to the next column
    positionsToRead = columnReaders[columnIdx].getReadPositions();
    }
}

if (positionCount != 0) {
    // Once column with filter evaluated and resulted in at-least one row,
evaluate all
    // other columns
    for (Integer idx : colReaderWithoutFilter) {
        if (columnReaders[columnIdx] != null) {
            positionCount = columnReaders[idx].read(getNextRowInGroup(),
positionsToRead, positionCount);
            if (positionCount == 0) {
                break;
            }
        }

        // Get list of row position to read for the next column. Output of
positions from
        // current column is input to the next column
        positionsToRead = columnReaders[columnIdx].getReadPositions();
        }
    }
}
```

NOTE: Re-ordering among the filter column will be taken up later. Here we can apply statistics to make it even more efficient.

Performance numbers

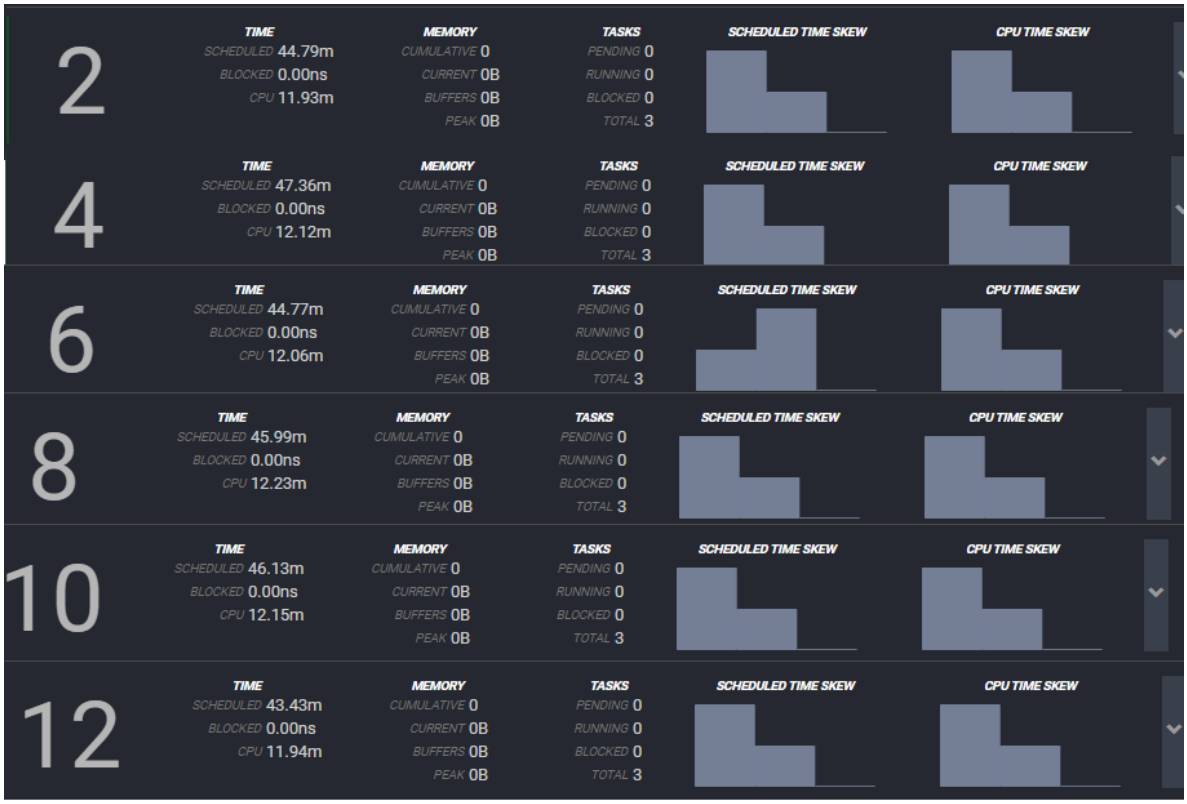
Below is the initial performance number and corresponding analysis for few of the TPC-DS queries along with some random query (Here only few queries result shown to show this feature benefit):

4 RH2288 nodes, 3 Worker, 1TB ORC Data Time(sec)							
Query	Branch	Reading-1	Reading-2	Reading-3	Reading-4	Reading-5	Average
Query-28	Original	58	52	58	52	58	55.60
	Modified	43	41	42	43	43	42.40
Query-52	Original	12	15	11	11	12	12.20
	Modified	13	12	12	13	11	12.20
Query-55	Original	11	11	13	12	12	11.80
	Modified	12	13	11	13	13	12.40
Random(select avg(ss_list_price), count(ss_list_price), count(distinct ss_list_price) from store_sales where ss_quantity >3 and ss_quantity <5)	Original	7	6	7	7	7	6.80
	Modified	5	5	6	5	5	5.20

So Query-28 gives performance improvement of around 24% with filter pushdown. Other queries 52 and 58 remains same. One of the other random query also give performance improvement of 24%. So in some cases (in this case query 52 and 55), performance improvement is actually there but not noticeable as overall CPU time consumed by scan + filter is too low compared to overall CPU time. Let's analysis the same in following section:

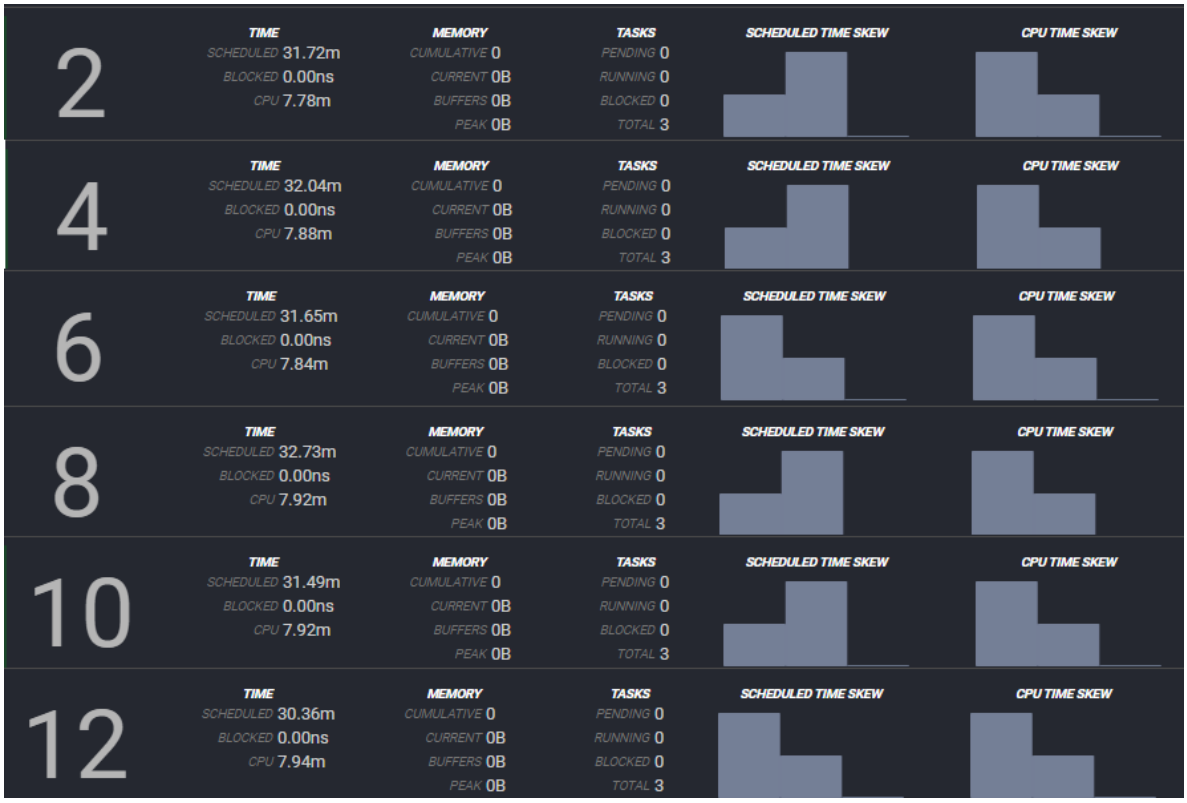
Analysis:

Query-28 Original:



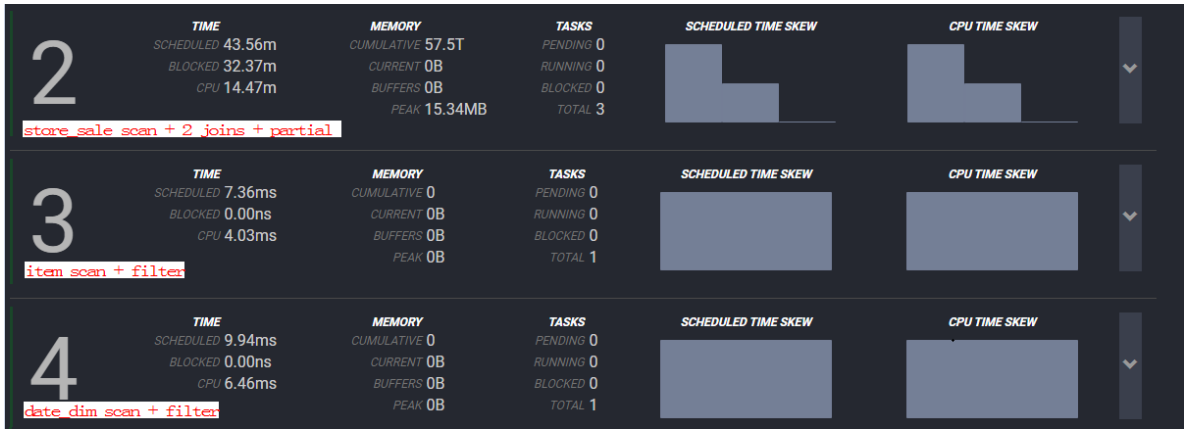

```
    Stage-2 (store_sales): Fetched Rows: 2.95B, CPU Time: 11.93m
    Stage-4 (store_sales): Fetched Rows: 2.95B, CPU Time: 12.12m
    Stage-6 (store_sales): Fetched Rows: 2.95B, CPU Time: 12.06m
    Stage-8 (store_sales): Fetched Rows: 2.95B, CPU Time: 12.23m
    Stage-10(store_sales): Fetched Rows: 2.95B, CPU Time: 12.15m
    Stage-12(store_sales): Fetched Rows: 2.95B, CPU Time: 11.94m
    Overall CPU Time: 1.24h
    Execution Time  58.18s
```

Query-28 Optimized:



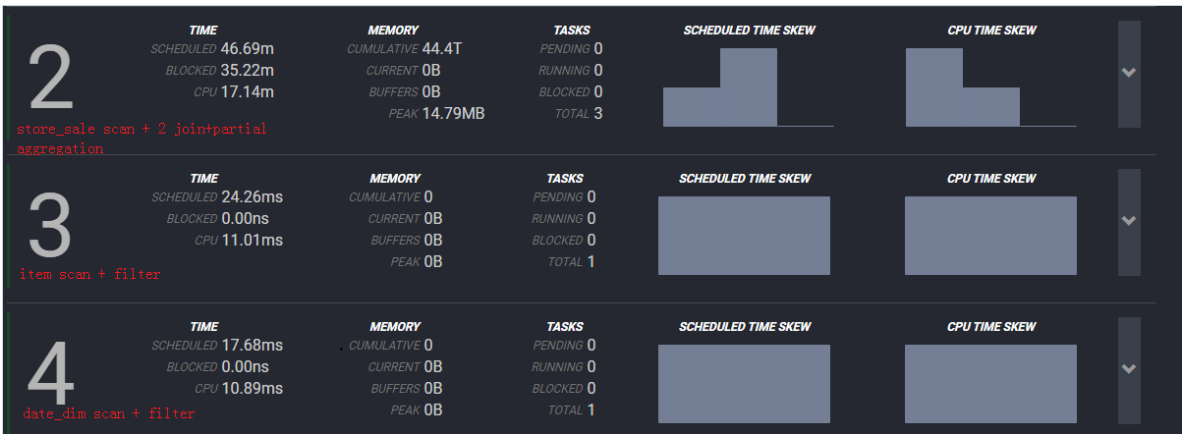

```
    Stage-2 (store_sales): Fetched Rows: 141M, CPU Time: 7.78m
    Stage-4 (store_sales): Fetched Rows: 141M, CPU Time: 7.88m
    Stage-6 (store_sales): Fetched Rows: 141M, CPU Time: 7.84m
    Stage-8 (store_sales): Fetched Rows: 141M, CPU Time: 7.92m
    Stage-10(store_sales): Fetched Rows: 141M, CPU Time: 7.92m
    Stage-12(store_sales): Fetched Rows: 141M, CPU Time: 7.94m
    Overall CPU Time: 49.25m
    Execution Time  43.32s
```

Query-52 Original:



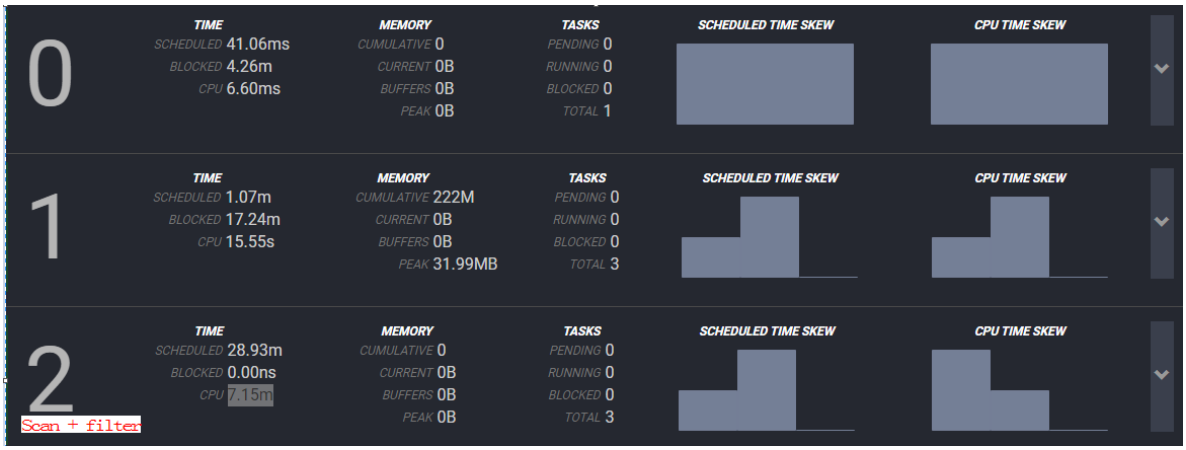

```
Stage-3 (item): Fetched Rows: 10K, 4.03ms CPU Time
Stage-4 (date_dim): Fetched Rows: 18K, 6.46ms CPU Time
Stage-2 takes maximum of 14.47m (Includes 2 join, partial aggregation)

Overall CPU Time: 14.53m
Execution Time  11.57s
```

Query-52 With Filter PushDown:



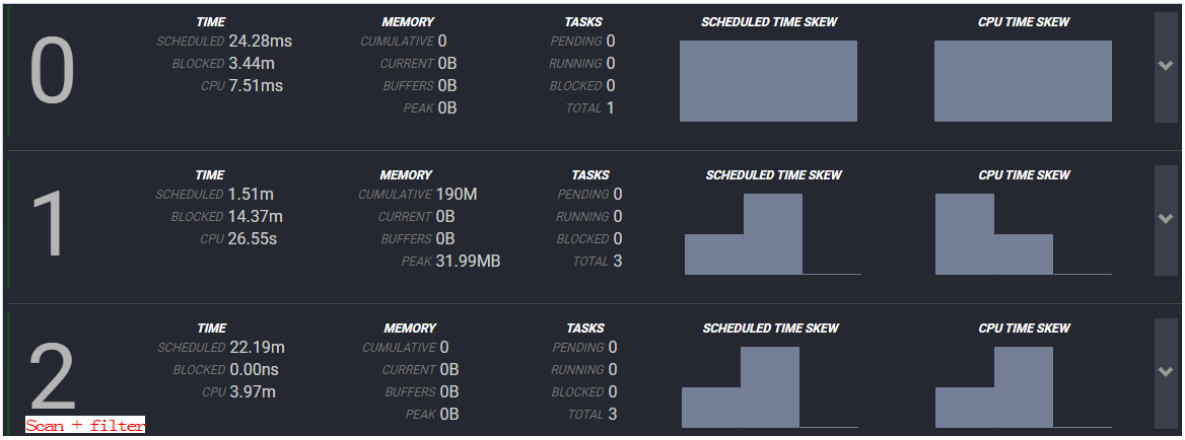

```
Stage-3 (Item): Fetched Rows: 30: 11.01ms CPU Time
Stage-4 (date_dim): Fetched Rows: 321: 10.89ms CPU Time
Stage-2 takes maximum of 17.14m (Includes 2 join, partial aggregation)

Overall CPU Time: 17.19m
Execution Time  11.54s
```

Random Query Original:




```
Stage-2 (store_sale): Fetched Rows: 2.95B, 7.15m CPU Time
Overall CPU Time: 7.41m
Execution Time  7.84s
```

Random Query with Filter PushDown:




```
Stage-2 (store_sale): Fetched Rows: 28.2M, 3.97m CPU Time
Overall CPU Time: 4.41m
Execution Time  6.35s
```

So from above analysis, it is clear wherever scan+filter is major contributor to overall query time, there performance is quite visible wit filter pushdown. Otherwise it remains almost same as without filter push down.

TODO:

Handling below scenario along with filter push-down:

1. Handling of remaining data-type:
    1. BYTE
    2. FLOAT
    3. DOUBLE
    4. LIST
    5. STRUCT
    6. UNION
    7. MAP.
2. ORC Caching
3. Dynamic filter
4. Heuristic index.
5. Adjust other DML operation.