Motivation:

Currently, there is a vast difference in runtime between below 2 queries run on **tpch** data of scale 42 in Parquet format with SNAPPY Compression:

1. `select * from lineitem where l_comment like 'blithely unusual pinto bean' (~33s)`
2. `select l_orderkey from lineitem where l_comment like 'blithely unusual pinto bean' (~11s)`

*** Both the queries are run on single impalad with **mt_dot** set to 1 to highlight the slowdown and make results more deterministic. But the behaviour can easily be extrapolated to bigger data sets in distributed settings.*

Both return 79 rows, scan the same table and filter on the same predicate. Still the second query is 3X faster than the first query. The reasons can be found in the counters below derived from the profile of each query:

1. Query1 counters:
   ```
   - BytesRead: 8.75 GB (9392896748) // Total bytes read
   - DecodeTupleTime: 14s448ms // Total Time to decode, convert and validate
                               // parquet values
   - DecompressionTime: 13s712ms // Total time to decompress parquet pages.
   - MaterializeTupleTime: 17s919ms // Time to materialize all the tuples
                                   //  includes DecodeTupleTime above
   - NumColumns: 16 (16)         // Total number of columns scanned
   - RowsRead: 252.01M (252010621) // Total Rows Read
   - RowsReturned: 79 (79)         // Final number of rows returned
   - ScanRangesComplete: 36 (36)
   - TotalTime: 33s055ms
   ```
2. Query 2 counters
   ```
   - BytesRead: 3.40 GB (3645729601)
   - BytesReadLocal: 3.40 GB (3645729601)
   - DecodeTupleTime: 1s118ms
   - DecompressionTime: 7s683ms
   - MaterializeTupleTime: 2s743ms
   - NumColumns: 2 (2)
   - RowsRead: 252.01M (252010621)
   - RowsReturned: 79 (79)
   - ScanRangesComplete: 36 (36)
   - TotalTime: 10s915ms
   ```

Looking at the counters there are 3 major reasons for Query 1 being slower than Query 2:

1. More data is scanned in Query 1: 8.75 GB vs 3.4 GB. The reason for this is the number of columns being scanned in Query 1 is 16 vs 2 in Query2. As parquet is a columnar format, individual columns can be scanned and filtered separately.
2. Decompression of the parquet columns: Data is compressed in Snappy format. With more columns to decompress, the decompression time taken by Query 2 is more too: `17s919ms vs  7s683ms.`
3. Materialization Time: It is total time spent in materializing tuples and evaluating predicates against them. This also includes time to decode individual column values

(shown in `DecodeTupleTime`). Materialization time for Query 1 is `17s919ms vs 2s743ms`. This is also because the number of columns being materialized is different i.e., 16 vs 2.

In Query2, since only 79 rows remained after filtering all the above 3 steps could have been avoided for columns except `l_comment`. `l_comment` has to be scanned, decompressed and materialized as filter predicate has to be evaluated over it. But we could have selectively avoided it for the rest of the 15 columns. In this document we would discuss avoiding the Materialization phase for filtered out rows and speed up queries like Query 1.

Problem Statement:

For Parquet row groups scanned, all the required columns are materialized and then the static filters and dynamic filters filter out the rows. In this process, even materialization of columns of filtered out rows also happens. Avoiding materialization of it can improve the performance on scan. Overheads have been illustrated in the above section.

Scope:

- This document only addresses the problem of avoiding the materialization of the filtered column values. It doesn't address the problem of avoiding scan or decompression.
- Emphasis on no regression for cases where filtering doesn't happen.

Current Flow:

For Parquet row groups scanned, it is processed in batches of 1024 size. Every batch of rows are read from decompressed data pages into `scratch_batch_`, materialized, filtered and transferred to the output batch. Core of this logic is here and explained below:

PROCESSING OF ROW GROUP IN PARQUET [1]

1. Allocate memory for `scratch_batch_` to hold 1024 tuples of the row group. This memory will be transferred to the output batch reading the scan later on.
2. Array of parquet column readers (`ParquetColumnReader`) `column_readers_` exists to read individual logical columns from Parquet file. If the value is collection then the reader would be `CollectionColumnReader` else it would be `BaseScalarColumnReader`.
3. For every column reader:
    a. Read '1024' values in Batch. Either `ReadNonRepeatedValueBatch` or `ReadValueBatch` are used.
    b. On reading error, RowGroup is skipped and resources are freed.
    c. Readers fill the data in respective slots in every tuple at `scratch_batch_->tuple_mem`

4. Transfer N values to output row batch:
    a. For every tuple, filter it against runtime filters and static filters.
    b. Finalize the transfer [5]
        i. If Scan is a selective, compact output batch i.e., copy surviving tuples to new memory and release '`scratch_batch_->tuple_mem`' [6].
        ii. Else, pointers to surviving tuples are stored in the output batch. `scratch_batch_->tuple_mem` is made null and ownership of the memory is transferred to row batch.
5. Goto Step 1 to decode the rest of data in RowGroup.

Changes to avoid materialization:

Proposed change to avoid materialization phase for column values of rows filtered out. The changes would avoid decoding, conversion and validation of such Parquet values like in the ReadSlot function here.

PROCESSING OF ROW GROUP IN PARQUET

1. PreProcess (Will be done in Scanner once):
    a. Extract list of 'SlotId' from runtime/static Filters.
    b. Map the list of 'SlotId' to the ColumnReaders. Based on this partition column readers into 2 groups:
        i. Filter Column Readers
        ii. Non-Filter Column Readers
2. If Filter Column Readers is empty, revert to the old method else new method as follows:
3. Allocate memory for `scratch_batch_` to hold 1024 tuples of the row group. This memory will be transferred to the output batch reading the scan later on.
4. Array of parquet column readers (`ParquetColumnReader`) `column_readers_` exists to read individual logical columns from Parquet file. If the value is a collection then the reader would be `CollectionColumnReader` else it would be `BaseScalarColumnReader`.
5. For every Filter Column Readers:
    a. Read 1024 values in Batch. Either `ReadNonRepeatedValueBatch` or `ReadValueBatch` are used.
    b. On reading error, RowGroup is skipped and resources are freed.
    c. Readers fill the data in respective slots in every tuple at `scratch_batch_->tuple_mem`
6. Save the address stored at `scratch_batch_->tuple_mem` in '`tuple_arr`'
7. Create `row_ranges_to_materilaize` when filtering below and initialize vector `prev_range` to empty.
8. Transfer 1024 values to output row batch ().
    a. For every tuple, filter it against runtime filters and static filters.

   - b. If tuple survives then add index to `prev_range` else add `prev_range` to `row_ranges_to_materilaize` and initialize `prev_range` to empty vector.
   - c. If Scan is a selective, compact output batch i.e., copy surviving tuples to new memory and release ‘`scratch_batch_->tuple_mem`’ [6]. Save new address in ‘`tuple_arr`’.
   - d. Else Pointers to surviving tuples are stored in the output batch, `scratch_batch_->tuple_mem` is made null and ownership of the memory is transferred to the output batch.

9. If all the 1024 tuples have been filtered then Page batch can be skipped:
   - a. For every column reader in Non-Filter Column Readers:
     - i. Use SkipTopLevelRows to skip 1024 rows.
     - ii. Goto Step 3
10. Merge ranges in `row_ranges_to_materilaize` that are separated less than MATERILIZATION_THREHOLD (see discussion below on this threshold)
11. For every column reader in Non-Filter Column Readers:
     - i. `rowIdx` = 0
     - ii. For every range `r` in `row_ranges_to_materilaize`
       1. If (rowIdx < r.start) SkipTopRows(r.start-rowIdx);
       2. Read the range `r` using either `ReadNonRepeatedValueBatch` or `ReadValueBatch`. Reader fills the data in respective slots in tuples whose pointers are stored at `tuple_arr`.
       3. `rowIdx = r.end + 1.`
     - iii. SkipTopRows(1023 - `rowIdx`)
12. Goto Step 1 to decode the rest of data in RowGroup.

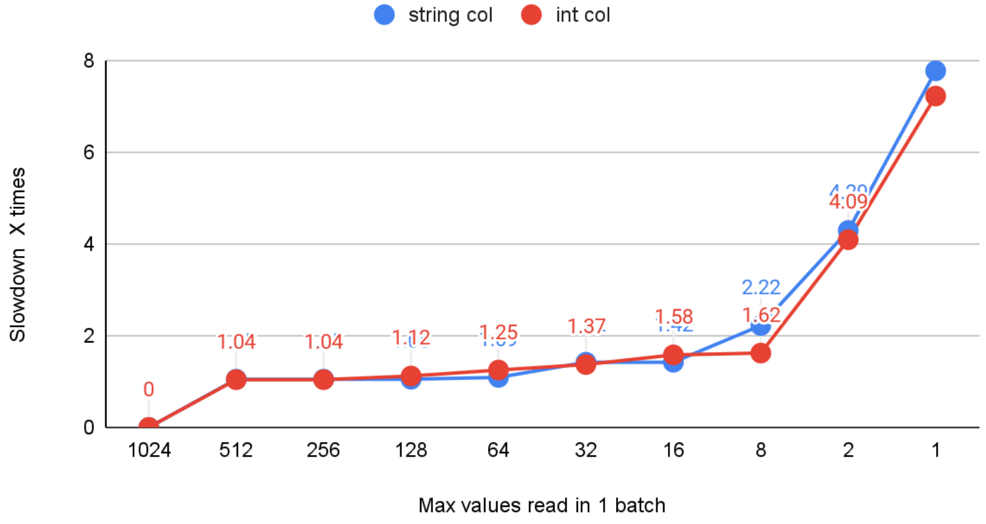
Materialization Threshold:

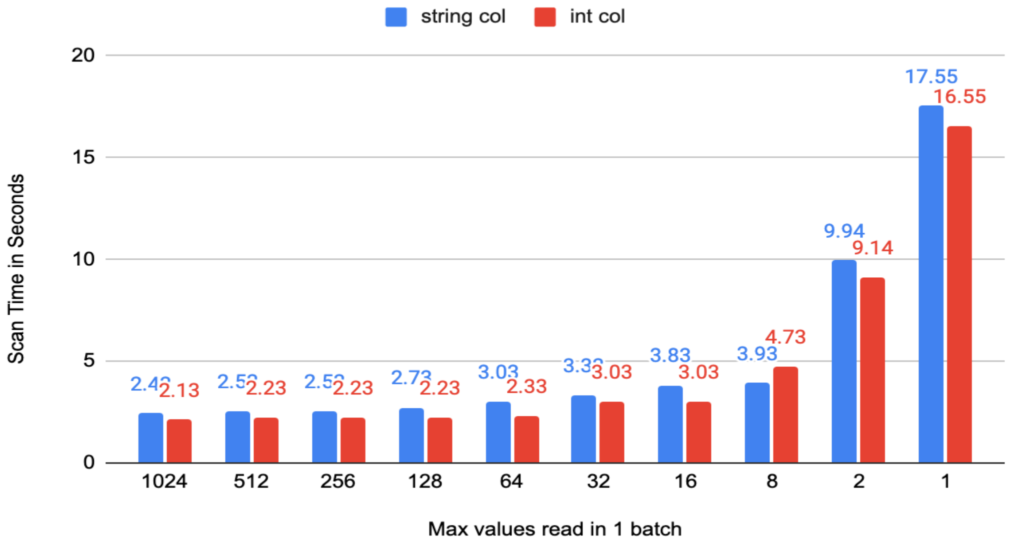
The reason for MATERILIZATION_THRESHOLD in the proposed change was due to decoding done in a batched manner. For a long streak of non-null or null values batch decoding is done by `ReadNonRepeatedValueBatch`. Breaking down that batch processing can severely affect the performance. We ran few experiments to check the slowdown when we batch together smaller number of rows for decoding via `ReadNonRepeatedValueBatch`: https://docs.google.com/spreadsheets/d/1v6PN7C52Ar8wzHhwxiyezLjph07uG-Ti0EiqkGVrqdw/edit?usp=sharing

We can see the effect in the charts below. As we reduce the batched values for decoding from 1024 to 1 we see a slowdown of around 7.22X and 7.77X for integer and string columns respectively. Other details:

1. Benchmark: TPCH 42 scale
2. Table: `lineitem`
3. Number of rows scanned or decoded: 252.01M (252010621).

Slowdown on decomposing batch decoding





Based on the above results, we check if it is worth it to skip the next ‘N’ rows (i.e., MATERILIZATION_THRESHOLD) as it has chances to cause slowdown. If ranges in `row_ranges_to_materilaize` are separated by more than MATERILIZATION_THRESHOLD, then we try to decode them in separate batches. Else those ranges are merged and decoded together in a single batch. MATERILIZATION_THRESHOLD will be configurable and will be set to default value of 10 (can change due to further experimentation).

RELATED CHANGES:

1. SkipTopLevelRows does not handle page boundaries. Either the batches need to be within the page boundary always or SkipTopLevelRows should handle those boundaries.
2. `ReadNonRepeatedValueBatch` or `ReadValueBatch` currently fills data in respective slots in tuples pointed to by `scratch_batch_->tuple_mem.` They need to be changed to take an array of tuple pointers instead (‘`tuple_arr`’).

Future Tasks:

1. Avoiding Decompression and Scanning of non-required column values.
2. Supporting Late Materialization for Broadcast Joins. Probe side tuples can be filtered out by Join condition and hence avoiding materialization until then can give more savings.
3. Work on supporting this for ORC format and other formats like Avro etc.

Code Reference:

1. AssembleRows (Core part of the processing that needs to change for Parquet): https://github.com/apache/impala/blob/84d784351c7a3606ec86abf6ea757aef72687b55/be/src/exec/parquet/hdfs-parquet-scanner.cc#L2095
2. Fetch SlotIds from filter: https://github.com/apache/impala/blob/b28da054f3595bb92873433211438306fc22fbc7/be/src/exprs/scalar-expr.cc#L330
3. Processing Scratch Batch and applying filters to it: https://github.com/apache/impala/blob/b28da054f3595bb92873433211438306fc22fbc7/be/src/exec/hdfs-columnar-scanner-ir.cc#L25

4. ReadSlot (Used to decode single slot): https://github.com/apache/impala/blob/b28da054f3595bb92873433211438306fc22fbc7/be/src/exec/parquet/parquet-column-readers.cc#L668
5. FinalizeTuples https://github.com/apache/impala/blob/9d46255739f94c53d686f670ee7b5981db59c148/be/src/exec/scratch-tuple-batch.h#L101
6. Compact Output Batch https://github.com/apache/impala/blob/9d46255739f94c53d686f670ee7b5981db59c148/be/src/exec/scratch-tuple-batch.h#L115