Order By

PQL does not support ORDER BY in GROUP BY. The goal is to add SQL equivalent support for ORDER BY in Pinot.

For complete ORDER BY support, we need to to be able to ORDER BY on a combination of

1) any of the aggregations

2) any of the group by keys

3) an expression formed out of 1 and 2

```
SELECT SUM(a), SUM(b) FROM table GROUP BY c, d ORDER BY <list of expressions formed using SUM(a), SUM(b), c and d>
```

Why ORDER BY is not possible in current implementation

The **AggregationGroupByOperator** is responsible for reading the records and generating group by keys and aggregations at the segment level. The group by key is generated as **1 concatenated string**. The values are generated as **Object array**. The **CombineGroupByOperator** takes the pair of concatenated group by key string and aggregations for the key **(String, Object[]),** and merges them across segments. From this point onwards in the data flow, we only have concatenated string. This makes it difficult to ORDER BY on specific value in the concatenated group by key. Additionally, the **Combine GroupByOperator** takes the (String groupByKey, Object[] aggregations) and **splits them into (String groupKey, Object aggregation) for each aggregation**. From this point onwards, we have separated the results for each aggregation, and no longer have any link across aggregations. This is also why in the final results, we display the results separately for each aggregation. **This makes it difficult to do any sort of ordering based on the aggregation results**.

Due to all these reasons, we don't have a holistic picture of the all the values which can be used for ordering.

Approaches

As discussed above, in the current implementation, the GroupByExecutor generates a concatenated string as the group by key, and an array of Objects for values. The CombineGroupByOperator takes this (String, Object[]) pair, and merges all the results, then splits it into (String, Object) for each aggregation.

We want to achieve 2 things:

1) To not split the results by each aggregation function, but instead have a single result row with (group_by_key, aggregations)

2) Be able to order by specific key within the group by key or aggregations or both.

For complete order by support, there were 3 approaches we considered:

1) **Parallel code path for plan/operator/execution**

In this approach, we would have completely new and parallel code for every step. and there will be no impact on the existing functionalities. This would also be a good chance to clean up the existing implementation. However, in this approach, we introduce yet another operator stack. Additionally, this would involve writing a parallel CombineOperator and AggregationGroupByOperator. Within the AggregationGroupByOperator, we would need to rewrite GroupKeyGenerators and the implementations DictionaryBased, NoDictionarySingleColumn, NoDictionaryMultiColumn, and also the various keyHolders within them - ArrayBased, , LongMapBased, IntMapBased, ArrayMapBased. This will also involve changing GroupByExecutors, and the GroupByResultHolders - DoubleGroupByResultHolder, ObjectGroupByResultHolder. These changes were quickly appearing to get out of hand to be attempted in one go.

2) **Modifying existing code throughout the stack**

This approach would be too risky. Also, the amount of files we would need to touch would be similar to the ones listed above.

3) **Hybrid approach: Modify parts of the stack at a time + introduce some parallel paths**

Since modifying the AggregationGroupByOperator involves changing a lot of places all at once, we decided to first work from the CombineOperator upwards. Once done, we will revisit the stack from AggregationGroupByOperator downwards.

Below we roughly outline the steps in the phases, and discuss some considerations and challenges and the phase we will address them in.

Phase 1 Goal

Phase 1: - PR https://github.com/apache/incubator-pinot/pull/4527

i. Basic functional Order By support

```
SELECT SUM(a), SUM(b) FROM table GROUP BY c, d ORDER BY <any combination of SUM(a), SUM(b), c and d>
```

We will not support expression evaluation in ORDER BY, as that requires transform operators to get involved. We will only allow order by on any combination of the aggregations and group by.

ii. Basic trimming, equivalent to current code's trimming behavior (As explained in the trimming sections below)

iii. Work mostly starting CombineGroupByOrderByOperator layer and above

TODO

i) Expression evaluation in order by

ii) Smarter trimming:

- Inner segment - Use Map + PQ for on the fly trimming. Can OrderByTrimmingService be used? This would mean AggregationGroupByOperator gets refactored to understand GroupByRecord. Is this feasible?
- Inter segment - Basic trimming and ordering exists. Use Map + PQ for on the fly operation. Apply smarter limits.
- Broker level - Basic collect all + sort approach implemented currently. Move to PQ based approach.

iii) A single ResultSet object to host all results + schema. Possibly use this in all queries.

iv) Remove duplicate CombineGroupByOperator and CombineGroupByOrderByOperator. Keep just 1 path of execution.

Considerations and Challenges

In order to figure out all the places that need changes, we need to understand all the levels of trimming that the group by results undergo. We do trimming of results at various stages

1) **Segment Level**: Trimming of group by keys, using innerSegmentLimit (AggregationGroupByOperator)

2) **Instance Level**: Trimming of results when merging results from all segments, using interSegmentLimit (CombineGroupByOperator)

3) **Broker Level**: Trimming of results after reduce using TOP N (BrokerReduceService)

Every stage that we trim, we should be ordering first and then trimming.

1) **Segment level**: In order to be able to order before trimming at the Segment Level, we need to change the GroupKeyGenerator. The GroupKeyGenerator simply drops keys after reaching innerSegmentLimit, which will make us lose valid groups. The GroupKeyGenerator also produces the group keys as a concatenated string, which makes it difficult to maneuver it for ordering. The GroupKeyGenerator has multiple implementations (DictionaryBased, NoDictionarySingleColumn, NoDictionaryMultiColumn), and we have to change them all. Within each we have various keyHolders - ArrayBased, , LongMapBased, IntMapBased, ArrayMapBased. This will also involve changing GroupByExecutors, and the GroupByResultHolders - DoubleGroupByResultHolder, ObjectGroupByResultHolder.

2) **Instance Level:** In order to trim at the Instance Level, we need to get away from the concatenated string key supplied by previous stage (AggregationGroupByOperator), and convert it into something we can easily use for ordering. This is the stage where the result set gets split into one result set per aggregation. We need to keep the aggregations closely associated with the group key, as we need to look at group keys + aggregations together for the ordering. The changes will mainly be needed only in CombineGroupByOperator.

3) **Broker Level**: In order to trim at the Broker Level, we need to be able to read the DataTables and convert the data back into a form where we have an association between group by keys and all aggregations, so that we can look at them together for ordering. Changes will be needed mainly in the BrokerReduceService where results are combined and then trimmed.

For the first phase, we proposed to begin working our way from level 2 onwards. Level 1 will involve significant rewriting and will impact the existing GROUP BY functionality.

Overall ordering and trimming strategy

1) **Segment Level** - Retain the existing trimming behaviour i.e. drop keys after innerSegmentLimit. This does mean that we will lose out on some valid results, and hence some correctness. In the 2nd phase, we will work on this layer.

2) **Instance Level** - In this stage we will order and trim by interSegmentLimit.

Possible options of doing this. In CombineGroupByOperator:

- i) Combine all the results from all segments, without any trimming. After combine, order them, and trim upto interSegmentLimit.
  Pros: very simple and straightforward
  Cons: We will gather results, possibly way above interSegmentLimit, in memory, only to trim them off later.
- ii) Combine all results from all segments using a PriorityQueue. We can maintain the size of the PriorityQueue to interSegmentLimit.
  Pros: This will help us achieve on the fly ordering. After combining, we will have no more processing.
  Cons: After we reach interSegmentLimit, in the worst case, every record will contribute O(nlogn) (peak + poll + offer)
- iii) Combine all results from all segments using a PriorityQueue. When queue becomes full, instead of removing just 1, evict 10%. This will potentially introduce some error in the bottom few results. More details need to be ironed out for this approach, regarding how much to evict, whether to maintain those evicted results etc.

3) **Broker Level** - Retain the existing behavior i.e. reduce results from all data tables sequentially. Only difference, before selecting the TOP N, order the results.

Some considerations about trimming

We need to revise the approach of trimming at various levels using innerSegmentLimit and interSegmentLimit. Depending on the type of order by clause, trimming anything at all could be wrong.

1) **Aggregations in ORDER BY**:

If we have aggregations in ORDER BY, it is never correct to trim, except at the final Broker Level. The aggregation values keep changing as we go up the levels. Trimming at any level other than the broker, might mean premature trimming of results.

2) **Only Group By values in ORDER BY**:

If the ORDER BY clause contains only the values from the GROUP BY, trimming can be performed at every level. This is because the values are never going to change. However, the trimming we can do should be TOP N. Trimming by innerSegmentLimit or interSegmentLimit can make us lose results, if those limits are lesser than TOP.

3) **Some aggregations, some Group By values in ORDER BY**:

If the ORDER BY clause contains a mixture of Group By values and aggregations,

if the first order by clause if on a Group By key, it is still okay to trim upto TOP N at each level, provided for the last key, we take in all of the values.

Although **correctness demands that we not trim at all or trim TOP N** depending on the clause, it will not be practical to follow this. We will be adding strain on the memory to carry and process all the results all along the stack. We will also be introducing latencies due to the added processing. We need to take a call about what limits we put at which level, or come up with ways to better approximate the limits to use. We would love to hear suggestions on the limits of trimming from the community.

Design

The proposal is to introduce these new constructs which will make it easier to maneuver the group by results for ordering and trimming.

Initial prototype can be found: https://github.com/npawar/incubator-pinot/tree/group_by_order_by

Server Side

GroupByRecord

GroupByRecord

```
/**
 * Maintains the group by key and aggregations for each record
 */
public class GroupByRecord {
  String[] _groupByValues;
  Object[] _aggregations;
}
```

OrderByType

OrderByType

```
/**
 * Order by clause can have either a group by value, or an aggregation value. This enum to distinguish between
the 2 cases.
 */
public enum OrderByType {
  GROUP_BY_KEY,
  AGGREGATION_VALUE
}
```

OrderByInfo

OrderByDefinition

```
/**
 * One OrderByInfo will be made for each order by expression. This definition will capture all elements needed
to execute an order by on this column
 */
public class OrderByInfo {
  OrderByType _orderByType; // from group by key, or from aggregation results
  int _index; // given a group by record, at what index does this column reside in the GroupByRecord::
_groupByValues or GroupByRecord::_aggregations array
  boolean _ascending;
  ColumnDataType _columnDataType;

  /**
   * Method to convert a List of order by clause into List of OrderByDefinitions
   */
  public static List<OrderByDefinition> getOrderByDefinitionsFromBrokerRequest(List<SelectionSort> orderBy,
GroupBy groupBy, List<AggregationInfo> aggregationInfos, DataSchema dataSchema);
}
```

OrderByUtils

OrderByExecutor

```
/**
 * Given a List of OrderByInfo, generates a comparator
 */
public class OrderByUtils {
  public void getComparator(List<OrderByInfo> orderByInfos);
}
```

CombineGroupByOrderByOperator

This will be similar to CombineGroupByOperator. This class will attempt to order and trim the results based on the trimming strategy chosen from above approaches. This class will be breaking away from the concatenated group by string approach. This class will also change the results from per aggregation to just all aggregations together. Accordingly, all the classes higher up (IntermediateResultsBlock, DataTable etc) will have to adapt to new result set construct.

GroupByOrderByTrimmingService

A new trimming service (very similar in duties to the AggregationGroupByTrimmingService).

Broker side

GroupByOrderByResults - a class similar to SelectionResults, to set results in the BrokerResponseNative

GroupByOrderByResults

```
@JsonPropertyOrder({"columns", "groupByKeys", "results"})
public class GroupByOrderByResults {
  List<String> _columns;
  List<String[]>_groupByKeys;
  List<Serializable[]> _rows;
}
```

Benchmarking

We can only do an apples to apples comparison of before and after performance for only 1 case - single aggregation with ORDER BY aggregation. We will benchmark to check that we are not performing worse
PQL:
SELECT SUM(metric) FROM table GROUP BY dimensions..
Is equivalent to
SQL:
SELECT SUM(metric) FROM table GROUP BY dimensions.. ORDER BY SUM(metric) DESC

For another case, we will likely do better than existing behaviour - multiple aggregations in selections. In PQL we return N results for each aggregation. In SQL, we will return only total N results. Benchmark these to collect proof that we're doing better