CDC performance Improvement

Author: Akash R Nilugal
Version: V1

Background:

The Carbondata's CDC is an important feature as CDC is an important use case in data analytics of merging the source data changes to target table. With the current design of CDC, the performance is not good when the data is huge in the target table and input source data also. So this document focuses on improving the carbondata CDC performance.

Bottleneck:

In the existing architecture, basically, we do the join operations(we decide the join operation based on the operations involved and the matched conditions involved in the merge statement) on the source and target table based on the join or key column and then based on that we perform the underlying operations.

1. The join operation is costly as it involves shuffle and more data to join.
2. Currently, we join the whole target table without any pruning which will be huge in the actual use case.

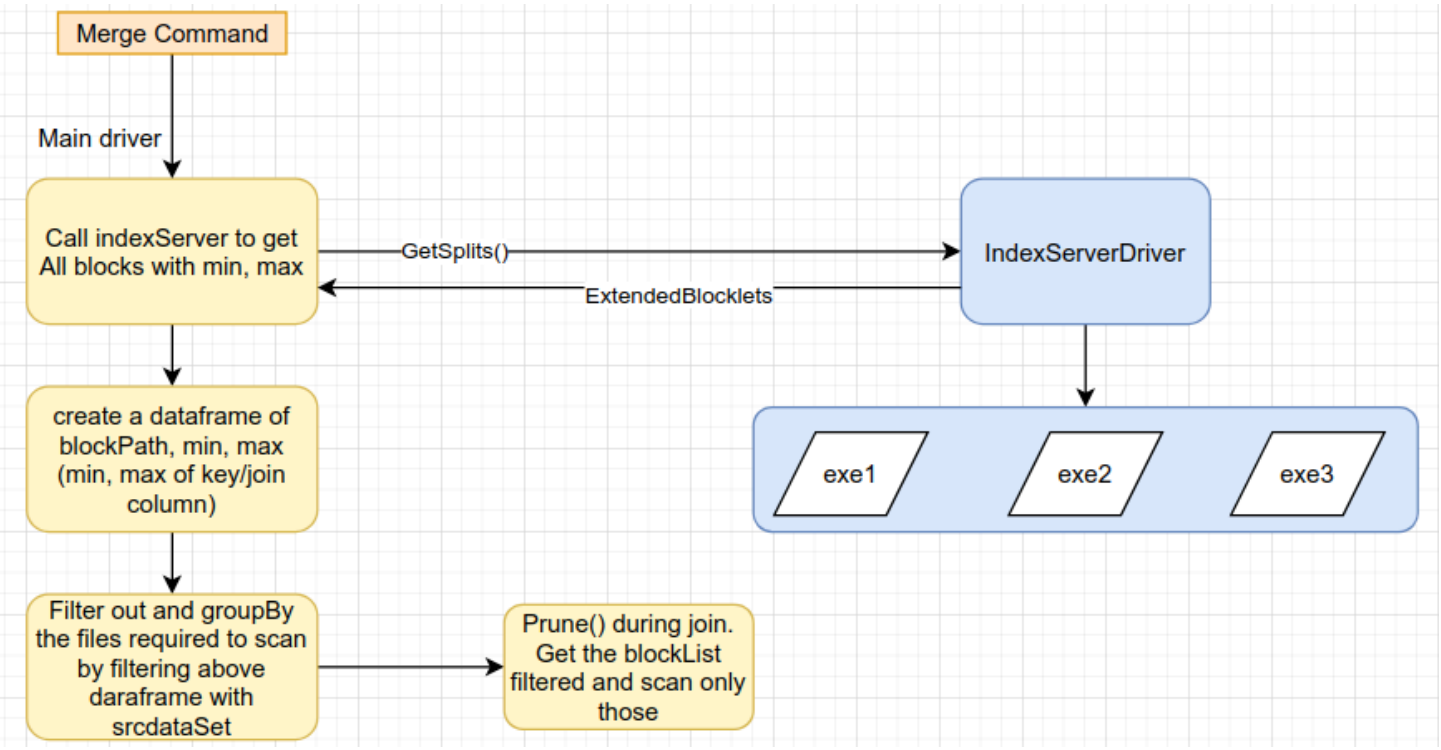
So we should basically reduce the data on the target table involved in join and fasten the scan and the join and in turn, reduce the data involved in the shuffle to improve the performance significantly.

Proposed Solutions:
Here I propose the improvement solutions into three phases,
Phase1: use partition and min-max pruning on the target table based on the source dataset
Phase2: Use bloom index on the pruned files of the phase1 to do more fine graned pruning
Phase3: Add Hbase index for faster pruning on the pruned files of the phase1.

Phase1 in turn I divided into two parts,
Part1: min-max pruning using index server and getting the required files to scan
Part2: Use repartitioning logic based on the output of the Part1 and try to avoid the shuffle cost of join

This document as of now focuses on Part1 of the phase1, we will add other improvements to the same document as and when we take up the improvements one after the other.

The below is the simple diagram of how the new flow will be



Steps Involved:

1. When the merge command is called, make a call to the index server if enabled, with the valid segments which returns the list of extended blocklets back to the main driver
    a. If there only one join column, no need to send a filter
    b. If multiple join columns, and all are simple equalto condition of two table columns, no need of filter
    c. If there are multiple filters involved, then from the existing join condition get the filters on the target table (where the filter will be based on some literal rather than equal to filter on source table column) and send to index server to get the splits based on that and cache in the index server.

If the index server is disabled, then call the getSplits() and cache in the main driver itself.

2. Index server returns the list of extended blocklets from which we can get the column min, max for all the join columns involved, and file paths or the block

paths. Based on partitioned table or not, make the unique block paths that help in pruning, and then prepare a spark dataframe of these as columns. The column names in the dataframe will be (FilePath, min, max) [**TODO: to make it common when there are multiple join columns involved**]

3. Based on the source dataframe, apply filter on the dataframe created of min max and then get only files/splits to scan which might contain the source data. The query can be like below

```
SELECT distinct(target.filepath)
        FROM targetTable,srcTable
WHERE srcTable.value BETWEEN targetTable.min AND targetTable.max;
```

   1. **Since this query contains a range filter in a join condition, if any of the dataset doesn't fit in memory it goes for Cartesian product which will make it very slow. So is it better to go for our own search logic of files in a distributed way is better or is there any other better way to rewrite the above query? Or do we need to write our partition logic for these datasets to avoid Cartesian and maybe make it a sort-merge join?**
   2. **Use the interval-based tree, where a tree is constructed with fileName, min and max of the columns, then the search happens in executor for each row of the soruce dataset, then we do group by and distinct to identify the final files to scan during actual join.**

   Any comments or suggestions are welcomed here as it's still not finalized.

4. Once the required files obtained, add a UDF which basically takes all these file paths and adds them as a filter during target table pruning, so that during actual block pruning, it takes care to consider only these carbon data blocks.

   UDF name is "**block_paths**" and the class will be **BlockPathsUDF**
5. When the dataframe of join of the source and the target dataset is prepared, we give a where filter with the above UDF mentioned which accepts the blocks list to scan. This dataframe with the UDFfilter is processed further.
6. A new expression called **CDCBlockImplicitExpression** is created which basically takes the filtered block paths of step 3 and added as a filter to IndexFilter of scanRDD.
7. **CDCBlockImplictExecutorImpl** implements **ImplicitColumnFilterExecutor**
   During building filter executor, if the expression is of **CDCBlockImplicitExpression**, we return **CDCBlockImplictExecutorImpl** which takes the block paths from the expression object.

This class implements the method **isFilterValuesPresentInBlockOrBlocklet**, which helps to prune only the files present in the list.

8. When the drier pruning is finished, when the format is prepared for executor, the **CDCBlockImplicitExpression** is replaced with the True expression, so that **CDCBlockImplictExecutorImpl** doesn't need to implement the methods prunePages, applyFilter etc like the SI flow does.

NOTE: Handling pruning when the source table or data source is partition table:
When the source dataset is a partition table and the partition columns are the same or subset of target table partition columns, then we should get only the selected partitions from the source table and go ahead.

NOTE:

1. **When the join type is "full_outer", there is no use in applying this optimization as we will be needing all the keys from the left table (target table)**
2. **The pruning happens for the equijoin join conditions itself, when there are other filter conditions, not to handle here, let spark apply its optimizations and not to make carbon's code complex.**