

BloomFilter Design Doc

When scanning Kudu table user can specify "KuduPredicate", by which to filter rows based on the value of a column. The types of predicate can be "Equality", "Range", "Is_not_null", "Is_null", "In_list". This doc proposes to add a new type of predicate -- "InBloomFilter", which contains a bloom filter(https://en.wikipedia.org/wiki/Bloom_filter) inside. When a Kudu scan is tagged with predicate of "InBloomFilter", TServer should only return rows which can pass the bloom filter.

Rationality

I will take a Spark operator -- "Join" as an example and reason out "InBloomFilter" is a killing feature to improve performance.

BroadcastJoin(aka Map Join) is really common/popular in Spark SQL. It runs by below steps:

1. When do broadcast join, we have a small table and a big table; Spark will read all data from small table to one worker and build a hash table;
2. The generated hash table from step 1 is broadcasted to the other workers, which will read their corresponding splits from big table;
3. Workers start fetching and iterating all the splits of big table and see if the joining keys exists in the hash table; Only matched joining keys is retained.

From above, step 3 is the heaviest, especially when the worker and split storage is not on the same host and bandwidth is limited. Actually the cost brought by step 3 is not always necessary. Think about below scenario.

Small table A

id	name
----	------

1	Jin
---	-----

6	Xing
---	------

Big table B

id	age
1	10
2	21
3	33
4	65
5	32
6	23
7	18
8	20
9	22

Run query with SQL: **select * from A inner join B on A.id=B.id**

It's pretty easy to figure out that we don't need to fetch all the data from Table B, because the number of matched keys is really small;

I propose to use the small table to build a bloom filter(BF) and wrap the generated BF in predicate of "InBloomFilter", then use this Kudu predicate to fetch data from big table, thus:

1. Much traffic/bandwidth is saved.
2. Less data to process by worker

Broadcast join is just an example, other types of join will also benefit if we scan with a BF

Approach

1. A Java Implementation for "BloomFilter"

Kudu should expose a standard Java implementation for bloom filter to different calculation engines.

This is already merged by (<https://gerrit.cloudera.org/#/c/11333/>)

This implementation should have below features:

1. User can create a bloom filter by indicating (size, number of rows, false positive rate)

2. User can put/add keys into a bloom filter. When put a record into bloom filter, it means you expect the TServer to return records with the same value in a scan.
3. Support all kinds of Kudu data types.
4. User can not customize the hashing algorithm used in bloom filter, because the self defined hashing may not be identified by TServer.
5. Kudu provides an enumeration of hashing algorithms for user to choose.

This is a simple use case

```
* {  
*  BloomFilter bf = BloomFilter.bySize(nBytes);  
*  bf.put(1);  
*  bf.put(3);  
*  bf.put(4);  
*  byte[] bitSet = bf.getBitSet();  
*  byte[] nHashes = bf.getNHashes();  
*  String hashFunctionName = bf.getHashFunctionName();  
*  // Serialize and wrap (bitSet, nHashes, hashFunctionName)  
*  // in "InBloomFilter" and send to TServer.  
* }
```

2. New added protocol

Add below new messages inside "ColumnPredicatePB". User sends the predicate from "KuduClient" to TServer

```
enum HashAlgorithmInBloomFilter {  
    MURMUR_HASH_2 = 0;  
}
```

```

message BloomFilter {
  // The hash times for bloom filter.
  optional int32 nhash = 1;
  // The bloom filter bitmap.
  optional bytes bloom_data = 2 [(kudu.REDACT) = true];
  // Hashing algorithm
  optional HashAlgorithmInBloomFilter hash_algorithm = 3;
}

message InBloomFilter {
  repeated BloomFilter bloom_filters = 1;
  // Lower and Upper is optional for InBloomFilter.
  // The inclusive lower bound.
  optional bytes lower = 3 [(kudu.REDACT) = true];
  // The exclusive upper bound.
  optional bytes upper = 4 [(kudu.REDACT) = true];
}

```

The reason to add "lower" and "upper" inside "InBloomFilter" is to provide a way to merge predicates of "InBloomFilter" and "Range" into one "ColumnPredicate"

3. ColumnPredicate

1. Add a new type of predicate -- "InBloomFilter" in "PredicateType"
2. Add a new element -- "std::vector<BloomFilterInner> bloom_filters_" to "ColumnPredicate". "BloomFilterInner" will be used to construct "BloomFilter"(bloom_filter.cc) to check cells.
3. Add a new method "void MergeIntoBloomFilter(const ColumnPredicate &other);" to "ColumnPredicate" to merge another predicate into a predicate of bloom filter.
4. Add a new method "EvaluateCellForBloomFilter" to "ColumnPredicate". If a "ColumnPredicate" is of type "InBloomFilter", evaluate by "EvaluateCellForBloomFilter"

Basically we can see that the main work to do on TServer side is the extension of "ColumnPredicate".

Statistics

Here I want show some statistics for Spark-Kudu integration when pushing down BloomFilter.

We do inner join with two tables – one is large and another one is comparatively small.

In Spark, inner join can be implemented as SortMergeJoin or BroadcastHashJoin, we implemented the corresponding operators with BloomFilter as SortMergeBloomFilterJoin and BroadcastBloomFilterJoin.

The hash table of BloomFilter is configured as 32M.

I show statistics as below:

Records of Table A	Records of Table B	Join Operator	Executor Time
400 thousands	14 billions	SortMergeJoin	760 seconds
400 thousands	14 billions	BroadcastHashJoin	376s
400 thousands	14 billions	BroadcastBloomFilterJoin	21s
2 millions	14 billions	SortMergeJoin	707s
2 millions	14 billions	BroadcastHashJoin	329s
2 millions	14 billions	SortMergeBloomFilterJoin	75s
2 millions	14 billions	BroadcastBloomFilterJoin	35s

As we can see, it benefit a lot from BloomFilter-PushDown.