Support full-text search in Spark SQL

Full-text search is a kind of text retrieval method that matches all the text in the document with the search term. Many application programs or systems provide full-text search capabilities.

**1.Full-text search in relational database**

Full-text search is provided in Database management system. For example, in MySQL, we can create index on designated columns and do full-text search on them. There are two steps: Creating index and Searching from index.

1. Build index

There are two ways to build index: create table or alter table.

CREATE TABLE tableName(

column1 TYPE,

column2 TYPE,

FULLTEXT (column1,column2)

) ENGIN=InnoDB;

ALTER TABLE tableName ADD FULLTEXT(column1, column2);

FULLTEXT (column1,column2) means creating index on two columns column1 and column2.

(2)Query from index

SELECT columnList, MATCH(indexedColumLlist)

AGAINST (‘query string’ [search\_modifier])

FROM tableName

WHERE MATCH(indexedColumLlist)

AGAINST (‘query string’ [search\_modifier])

search\_modifier:

{

IN NATURAL LANGUAGE MODE|

IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION|

IN BOOLEAN MODE|

WITH QUERY EXPANSION

}

The first MATCH AGAINST means rows returned will be added column ’score’ besides columnList. The second MATCH AGAINST is a filter that searchs indexedColums in search\_modifier(define search mode) to find most suitable rows and return rows sorted by score

Examples can be seen at <http://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html>

This function is commonly used in many situations but not currently supported in SparkSQL.

1. **Our design:Full-text search in Spark**

2.1 Grammar Design:

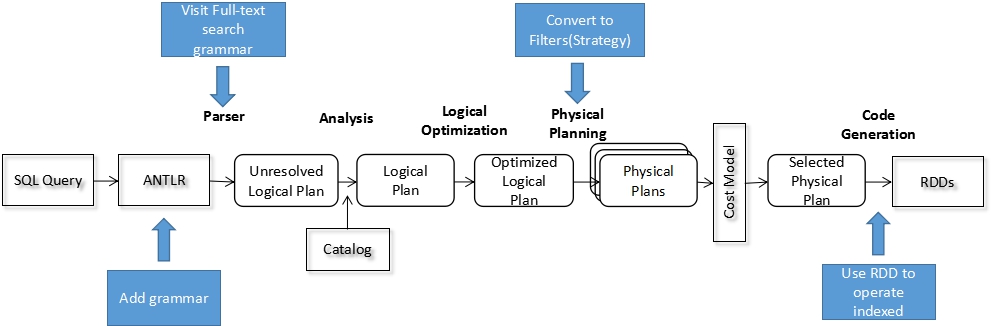
Like MySQL full-text search syntax, we should consider several factors:

1. Designate table
2. Designate columns to be indexed
3. Search\_modifier which means how to query based on indexes

2.2 Logical Plan Design:

Three steps need to be considered:

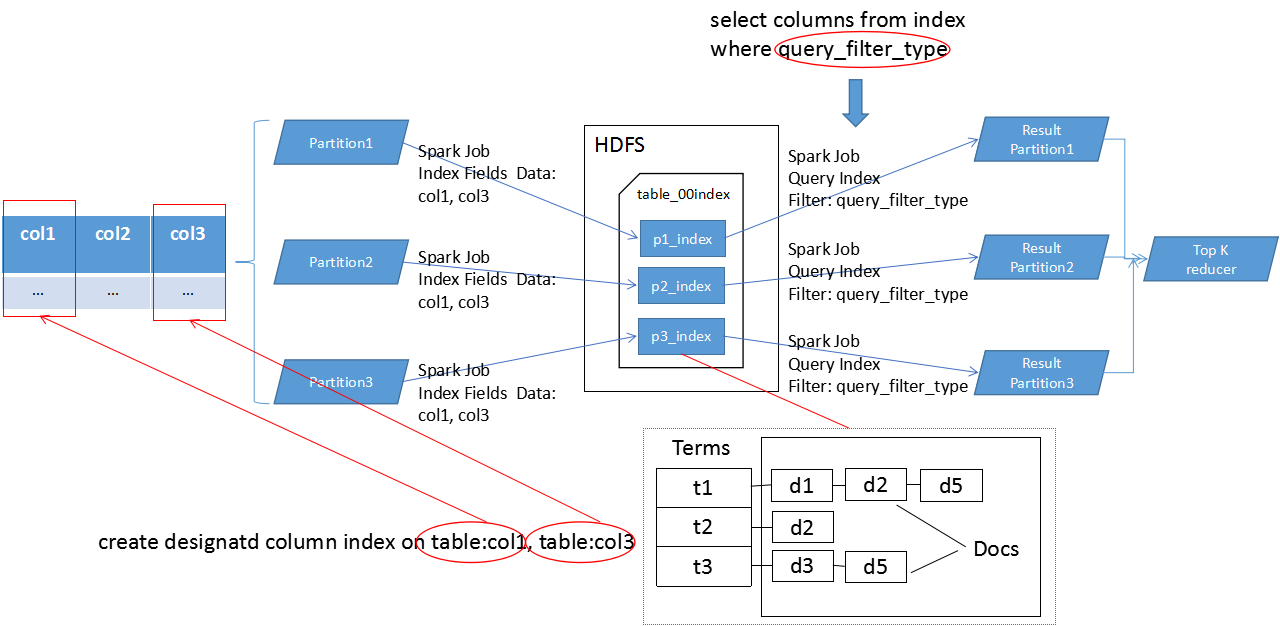
1. Add grammar to SqlBase.g4
2. Change full-text search structure to Catalyst TreeNode
3. Push down full-text search filters to RDD Operation



2.3 RDD Operation Design

There are operations need to be considered:

1. Read and index existing data(From an existing table)
2. Where to store index files
3. Do parallel full-text search on indexes
4. Data structure return to client
5. Global Top K



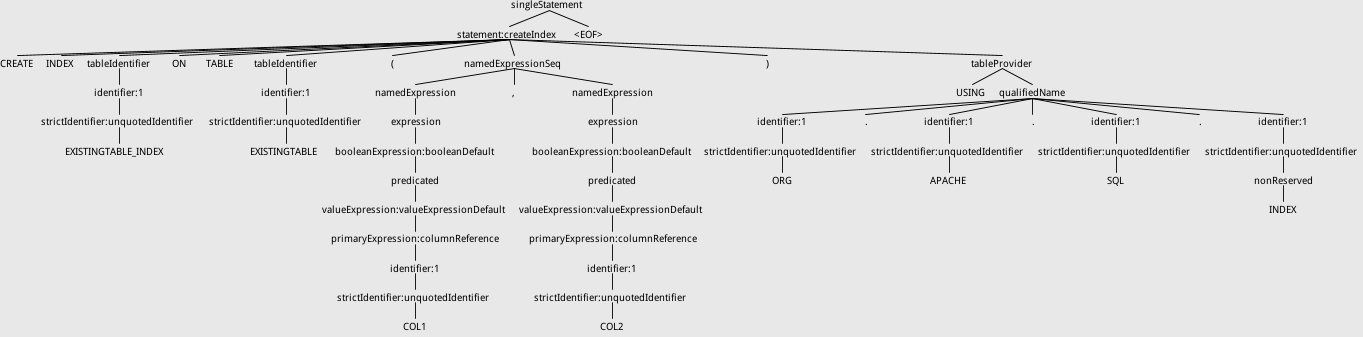
1. **Implement details**

3.1 Grammar

1. Create index from existing table

|  |  |
| --- | --- |
| **Spark SQL** | **Comparison(MySQL)** |
| CREATE INDEX existingTable\_index  ON TABLE existingTable (needIndexColumnList)  USING tableProvider | CREATE TABLE tableName(  column1 TYPE,  column2 TYPE,  FULLTEXT (column1,column2)  ) ENGIN=InnoDB;  ALTER TABLE tableName ADD FULLTEXT(column1, column2); |

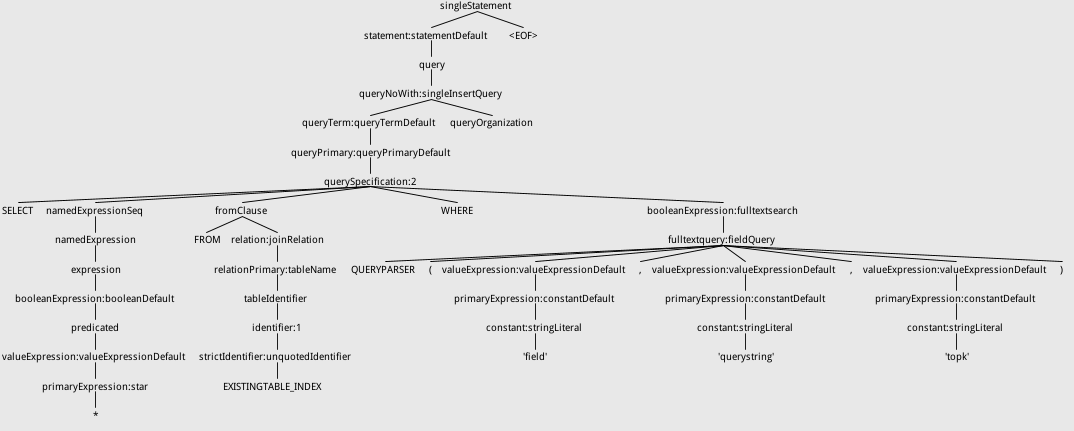
Grammar tree is:



1. Do full-text search from index

|  |  |
| --- | --- |
| **Spark SQL** | **Comparison(MySQL)** |
| SELECT columnList  FROM existingTable\_index  WHERE TERMQUERY|FUZZYQUERY|PHRASEQUERY|  PREFIXQUERY|QUERYPARSER  (‘field’, ’queryString’,[ ‘maxEdits’,] ‘topK’) | SELECT columnList, MATCH(indexedColumLlist)  AGAINST (‘query string’ [search\_modifier])  FROM tableName  WHERE MATCH(indexedColumLlist)  AGAINST (‘query string’ [search\_modifier]) |

Grammar tree is:



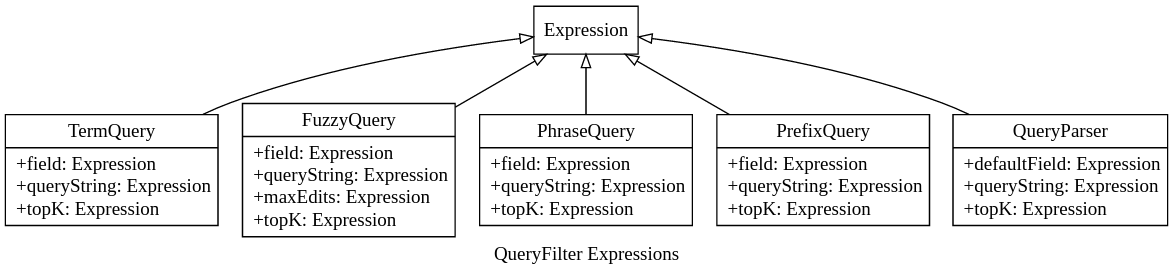
3.2Query Plan

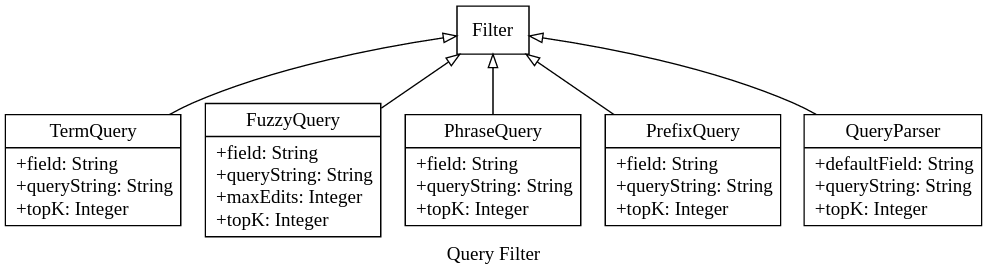
1. Create Index Command

We create a new Command named CreateIndexTable which ‘create index’ will be translated to in ddl.scala, and it will be converted to CreateIndexTableCommand to create a table using org.spark.sql.index in MANAGED way. So tableIndexName will be a table like others but we use org.spark.sql.index to manage its underlying storage. All columns data will store but only needIndexColumnList can be tokenized,indexed and queried.

1. Query Filter

TERMQUERY,FUZZYQUERY,PHRASEQUERY,PREFIXQUERY,QUERYPARSER are translated to Expressions with the same name, and in DataSourceStrategy, they are translated to Filters with the same name.These filters will be pushed down to IndexRelation.





1. IndexRelationProvider and IndexRelation

Underlying storage and access are all supported by LuceneRDD. We use RelationProvider to support low-level read/write operation. IndexRelation support scan(read) from LuceneRDD and insert data to LuceneRDD. Scan operation implements PrunedFilteredScana to receive filters(TERMQUERY,FUZZYQUERY,PHRASEQUERY,PREFIXQUERY,QUERYPARSER) from high level.

1. **RDD Operation**

4.1 Existing Problems

Our RDD Operation are based on the spark-lucenerdd developed by zouzias where all index operations are based on RDD operations which has a better performance. But There are some limits:

1. LuceneRDD can just build from existing rdd and store index to memory or physical machine disk current executors belong to , after we close current app, we can’t reuse previous index build by LuceneRDD because of the distributed environment.
2. LuceneRDD lacks the ability to build index on an designated table.
3. In Realation, results LuceneRDD returns are locally sorted, not globally.

4.2 Improvements

Based on the above problems and our previous design, we have the following implements:

1. Index build, write ,read and storage

DataSet or RDD is distributed which means that its data is distributed on multiple machines, so we need build lucene index locally. Every partition has one lucene index represented by LuceneRDDPartition.

LuceneRDDPartition is responsible for index build, write, read and query. In one partition of original data, LuceneRDDPartition processes data from this partition and store index to HDFS. When reading index, LuceneRDDPartition reads from HDFS, does search based on index and return LuceneRDDResponsePartition(Iterator[SparkScoreDoc]).

We change lucene index storage to a HDFS directory we defined so it can be found by for usage next time. When we build index on a table, indexes of this table will be stored in a HDFS directory named tablename\_00index(00index is just a suffix).

(2)LuceneRDD recovery

We provide the function to reconstruct a new LuceneRDD by reading indexes from HDFS. It’s useful when we want to reuse these indexes.

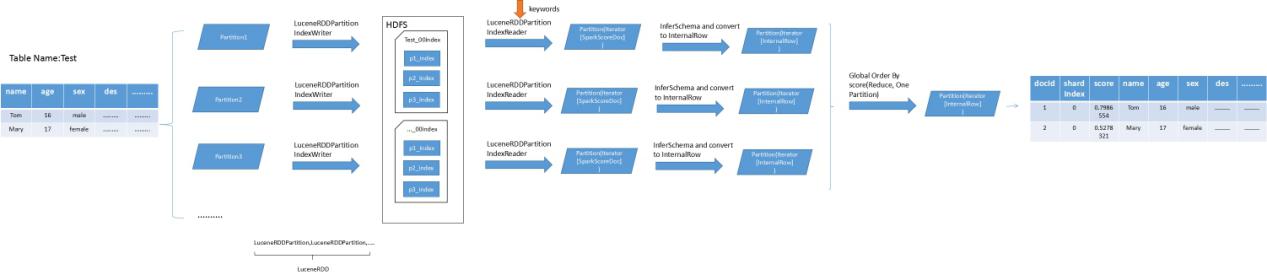
(3)Infer Schema

Query results will be converted to InternalRow based on the schema inferred from index’s fields’ data type, and three additional StructField will be added to schema: docId(Int), shardIndex(Int), score(Float).

(4)Global ranking

Final query results are all in one partition(Iterator size is topK) for global ranking by score in descending order.

1. **Global data processing**



There are steps to index a table and do full-text search on this table:

1. Partitions of this table will be parallel processed
2. Different partitions’ index data will be written in a HDFS directory named indexTableName\_00index(Executed by Spark Job)
3. Query will be mapped to Spark jobs, and job count is equal to original table data partitions. In another word, every index from original table data partition will be queried as a Spark job.
4. Reduce results from different jobs to one partitions for global top K ordered by score field.
5. **Experimental Results**

We use MySQL sample data(http://dev.mysql.com/doc/refman/5.7/en/fulltext-natural-language.html). Our code and screenshots of running are as follows:

|  |  |
| --- | --- |
| **Code** | **Screenshots** |
| // Just convert sample data to json format for easy usage  val df = sparkSession.read.json(“resources4/data.json”)  df.createOrReplaceTempView(“articles”)  df.printSchema() | root  |-- body: string (nullable = true)  |-- id: long (nullable = true)  |-- title: string (nullable = true) |
| sparkSession.sql(“SELECT \* FROM articles”).show() | IMG_256 |
| sparkSession.sql(“DROP TABLE IF EXISTS articles\_index”)  val df\_index = sparkSession.sql(“  CREATE INDEX articles\_index ON TABLE articles (title, body) USING org.apache.spark.sql.index”)  // Explain build index query plan  df\_index.explain(true) | IMG_256 |
| val df\_query = sparkSession.sql(“  SELECT \* FROM articles\_index WHERE QUERYPARSER(‘nothisfield’, ’body:database’, ‘3’)”)  // Explain query index logical plan  df\_query.explain(true) | IMG_256 |
| df.show() | IMG_256 |

As we can see, the query results are the same as in MySQL.

1. **Comparison with ES and Solr**

Existing solutions are ElasticSearch+Spark or Solr+Spark, but these solutions require ES Cluster or Solr Cluster which means extra overhead(Install, limited resources and so on). And these extensions is not straightforward for users, here are some comparisons:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Our Design** | **Elastic Search** | **Solr** |
| Usage | Conformance with RDBMS | 1.Add jar  2.Create Table  Using es Options | 1.Add jar  2.Create Table  Using solr Options |
| Create Index on designated columns of an existing table | Straightforward:  Create index index\_table\_name on table (indexedColumnList) using index | Not Supported, but can be done in another way:   1. Create a new es\_table using es Options 2. Insert into es\_table from existing\_table | Not Supported, but can be done in another way:  1.Create a new es\_table using solr Options  2.Insert into es\_table from existing\_table |
| Query from Index | Straightforward:  Select columnList from index\_table\_name where query\_filter | Hard to use:   1. Create a new es\_table using es Options(query ‘\*:\*’) 2. Select \* from es\_table | Hard to use:  1.Create a new es\_table using es Options(query ‘\*:\*’)  2.Select \* from es\_table |
| System overhead | Our Design < Elastic Search < Solr  Our Design is a native implement, so it doesn’t need other cluster(ES or Solr) installation and reduce unnecessary overhead | | |