Full-text search issue

Full-text search is is a kind of text retrieval method witch can match all the text in the document with the search item. In a full-text search, a search engine examines all of the words in every stored document as it tries to match search criteria.Many application programs or systems provide full-text search capabilities.

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

Full-text search is also provided in Database management system.F or example, in MySQL, there are two steps: Creating index and Searching from index.

1. Create index when creating table

CREATE TABLE tablename(

column1 TYPE,

column2 TYPE,

...

FULLTEXT (column1,column2)

) ENGIN=..

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

(2)Query from index

After we insert data into indexed table, we can do full-text search

SELECT columnList, MATCH(indexedColumLlist)

AGAINST (‘query string’

QUERYMODE)

FROM articles

WHERE MATCH(indexedColumLlist)

AGAINST (‘query string’

QUERYMODE)

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 QUERYMODE 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.

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 are only supplied with RDD Operation, people without RDD or API programming knowledge can’t use pure sql to do full-text search.So it’s necessary to support full-text search in native way.

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

2.1Grammar

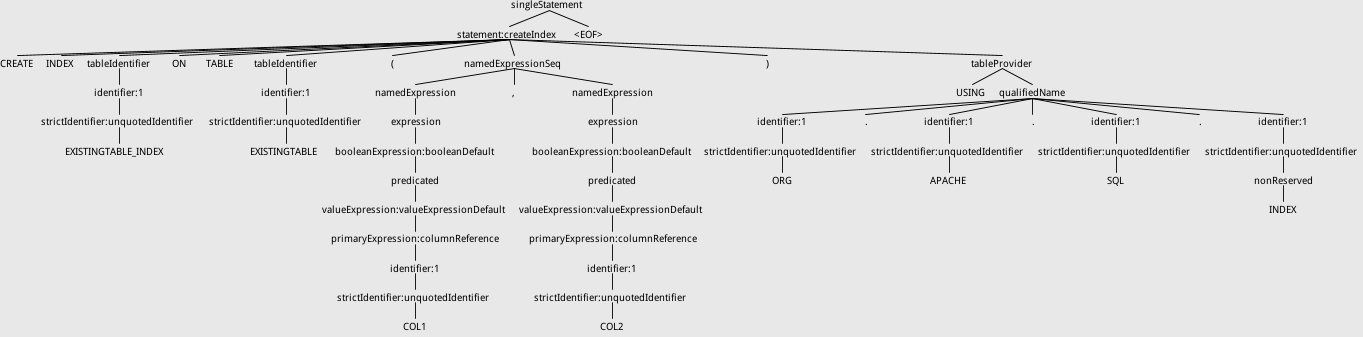
1. Create index from existing table

CREATE INDEX existingTable\_index

ON TABLE existingTable (needIndexColumnList)

USING tableProvider

Grammar tree is:



1. Do full-text search from index

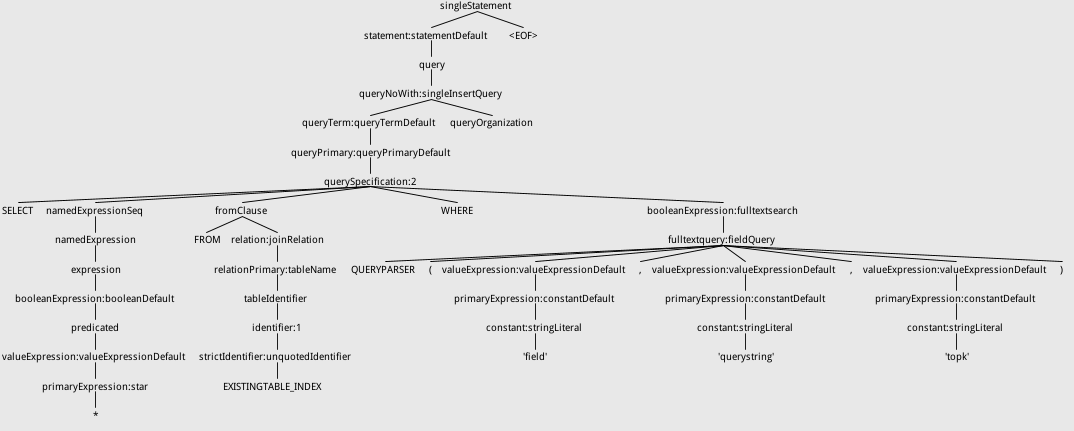
SELECT columnList

FROM existingTable\_index

WHERE TERMQUERY|FUZZYQUERY|PHRASEQUERY|PREFIXQUERY|QUERYPARSER

(‘field’, ’queryString’,[ ‘maxEdits’,] ‘topK’)

Grammar tree is:



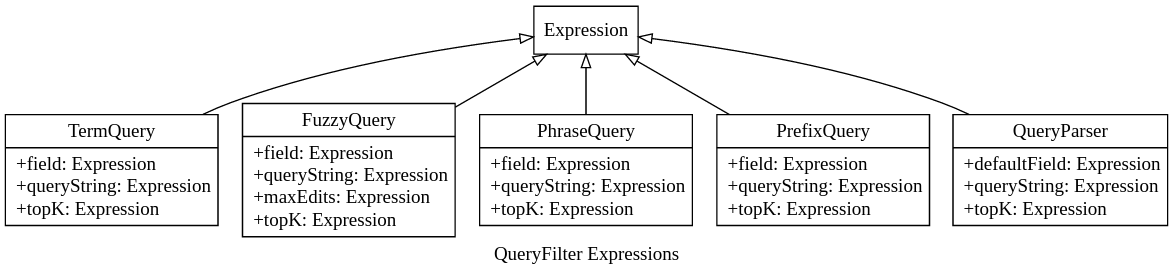
2.2Query Plan

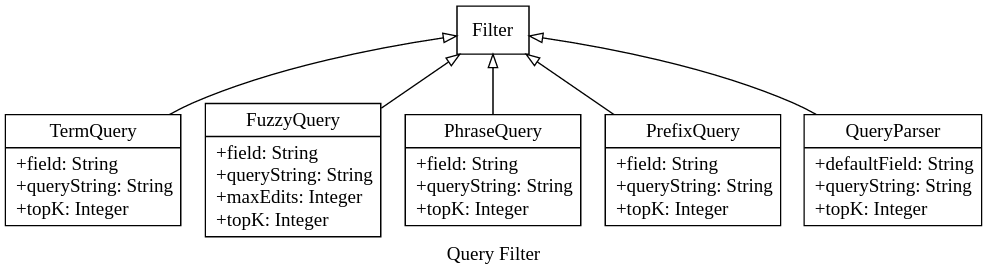
1. Create Index Command

We create a new Command named CreateIndexTable in ddl.scala, and it will be translated 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.

Query results will be RDD[Row] which just owns one partition for the global sort by score.

We use MySQL sample data(http://dev.mysql.com/doc/refman/5.7/en/fulltext-natural-language.html), and our sql is as follows:

create index:

// Just conver sample data to json format for easy usage

val df = sparkSession.read.json(“resources4/data.json”)

df.createOrReplaceTempView(“articles”)

df.printSchema()

sparkSession.sql(“SELECT \* FROM articles”).show()

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)

val df\_query = sparkSession.sql(“SELECT \* FROM articles\_index WHERE QUERYPARSER(‘nothisfield’, ’body:(Security implications of running MySQL as root) AND title:(security)’, ‘3’)”)

// Explain query index logical plan

df\_query.explain(true)

// Show query results

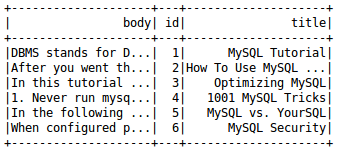
df.show()

Screen shots are:

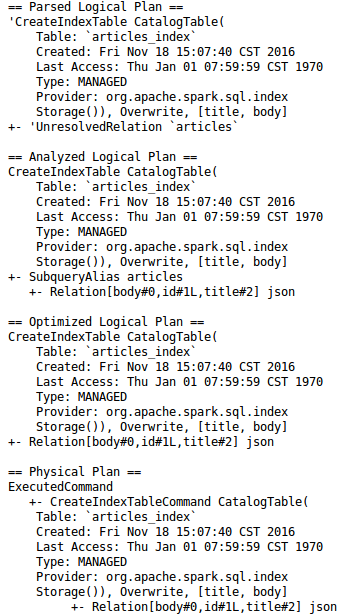
data.json schema

2016-11-18 15-08-39屏幕截图

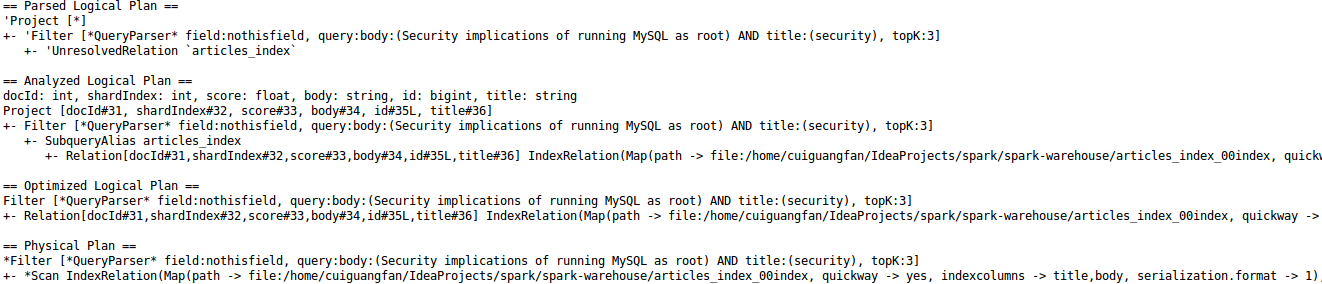
data.json inner data



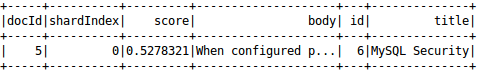
create index logical plan



query index logical plan



query results



**3.RDD Operation**

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: A.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.

Based on the above problems, we has the following design:

1. Index build, write ,read and storage

DataSet or RDD is distributed which means that it’s 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 and read. 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 with 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) because of global ranking by score in descending order.

Global data processing is shown in figure:

