Support Full-Text Search in Spark SQL

Guangfan Cui, Lijie Xu
cuiguangfan@xiaomi.com, xulijie@iscas.ac.cn

Full-text search (i.e., keyword search) is widely used in search engines and relational databases like MySQL. However, it is not natively supported in Spark SQL. We first introduce the full text search in RDBMS and then argue how to support full text in Spark SQL.

1. Full-text search in relational databases

Full-text search is supported in RDBMS. For example in MySQL, to search keywords in some columns, we can (1) create indexes on these columns, and (2) perform full-text search on them using MATCH() … AGAINST.

(1) Create indexes

A FULLTEXT index definition can be given in the CREATE TABLE statement when a table is created, or added later using ALTER TABLE or CREATE INDEX.

For example we create fulltext index on 'column1' and 'column2' on a table named 'table_name' with three columns:'column1','column2','column3'.

```
1) CREATE TABLE table_name (
column1 TYPE,
column2 TYPE,
column3 TYPE,
FULLTEXT [index_name](column1, column2)
) ENGINE=InnoDB|MyISAM
2) ALTER TABLE table_name
   ADD FULLTEXT INDEX
   [INDEX] [index_name](column1, column2)
3) CREATE FULLTEXT INDEX index_name
   ON table_name (column1, column2)
```

A full-text index in MySQL is an index of type FULLTEXT.

Full-text indexes can be used only with InnoDB or MyISAM tables, and can be created only for CHAR, VARCHAR, or TEXT columns.

(2) Perform full-text search using MATCH() … AGAINST

For example, We search 'query string' in 'column1' and 'column2', and we need rows returned are sorted by relevance, in the meantime, 'column3' and 'score' will re added to rows returned.

```
SELECT column3, MATCH(column1, column2)
AGAINST ('query string' [search_modifier]) as score
```

```
FROM table_name
WHERE MATCH(column1, column2)
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 query returns the relevance values when specifying MATCH() in the SELECT list. Relevance values are nonnegative floating-point numbers.

When MATCH() is used in a WHERE clause, the rows returned are automatically sorted with the highest relevance first.

There are three types of full-text searches(search_modifier):

1) Full-text searches are natural language searches if the IN NATURAL LANGUAGE MODE modifier is given or if no modifier is given.A natural language search interprets the search string as a phrase in natural human language (a phrase in free text)

2) The IN BOOLEAN MODE modifier specifies a boolean search.A boolean search interprets the search string using the rules of a special query language. The string contains the words to search for. It can also contain operators that specify requirements such that a word must be present or absent in matching rows, or that it should be weighted higher or lower than usual.

3) The IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION or WITH QUERY EXPANSION modifier specifies a query expansion search. A query expansion search is a modification of a natural language search. The search string is used to perform a natural language search. Then words from the most relevant rows returned by the search are added to the search string and the search is done again. The query returns the rows from the second search.

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.

2. Approach for Full-Text Search in Spark SQL

As far as user interactions are concerned, full-text search function in MySQL is easy to use, we can refer to it to design the approach. Our discussion will be divided into two aspects:(1)Grammar (2)Translation (3)Data Flow (4)RDD Operation (5)Storage Design

2.1 Grammar

(1) Create index

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

1) Designate table

2) Designate columns to be indexed

In MySQL, there are three ways to create index, every way is feasible, but we choos e the third one because CREATE operation is opposite to DROP operation , and we want indexes we created can be stored as a table so that we can drop it anytime we want. S o, the third one is a more concise and clear way.

Spark SQL	**MySQL**
CREATE INDEX index_name ON TABLE table_name (column1, column2) USING tableProvider	CREATE TABLE table_name(column1 TYPE, column2 TYPE, FULLTEXT (column1,column2)) ENGIN=InnoDB
	ALTER TABLE table_name ADD FULLTE XT(column1, column2)
	CREATE FULLTEXT INDEX index_name ON table_name (column1, column2)

Parameter interpretation:
1) index_name is the name of indexes.
2) table_name is the table we create index on.
3) column1, column2 are columns we create index on.
4) USING tableProvider means which kind of datasource will be used.

TableProvider here can give us better scalability if we want to use our own way to create index.

(2) Do full-text search

In MySQL, search_modifier means how to query based on indexes, so it's important to supply richly functional interface.

As we can see from search_modifier in MySQL, full-text search in MySQL support several query mode , so FILTER here can achive the same effect. Select from index_name means we can do query separated from original table.

Spark SQL	**MySQL**

SELECT *
FROM index_name
WHERE FILTER(parameters_list)
	SELECT * MATCH(column1,column2)
	AGAINST ('query string' [search_modifier])
	FROM table_name
	WHERE MATCH(column1,column2)
	AGAINST ('query string' [search_modifier])

Parameter interpretation:
1) *(or other column_name) are those need to be printed.
2) index_name is the name of table storing indexes.
3) FILTER is the interface to do full-text query.

2.2 Translation

Translation is the base of full-text search system architecture.

In this part, we will discuss translation process.

In Catalyst, the translation converts input sql string to RDD operation.

Three things need to be considered:

1) New grammar

2) Recognize full-text search grammar in Catalyst

3) Convert full-text search structure(TreeNode) to RDD operation

We use Catalyst's general tree transformation framework to explain our approach, as show in figure 1:

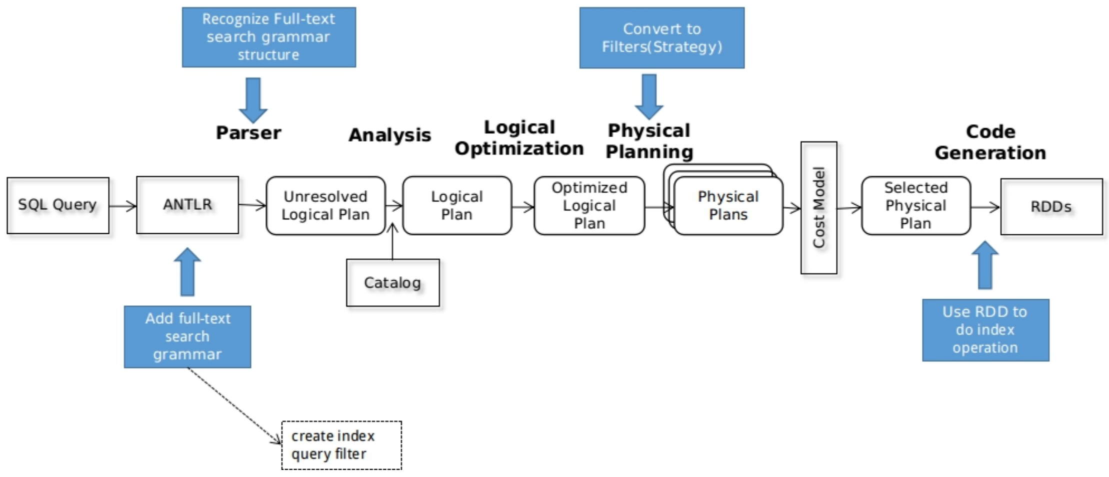


Figure1. Catalyst transformation framework

Every sql string will be processed in ANTLR, Parser, Analyzer, Optimizer and Spark Planner.

For our goals, we need to do the following things:

1) Spark SQL use ANTLR to define grammar, so we need to add grammar to ANTLR. Here we will add CREATE INDEX and query FILTER as we design in previous section.

2) Full-text search structure should be transformed to tree node in Parser which can be recognized by Catalyst.

3) Convert to Filters which can be pushed down.

4) New RDD implement to do index operations according to the filters pushed down.

2.3 Data Flow chart of full-text search

There are two data flow paths:

1) Create index on existing table and next, do full-text search on index. This path corresponds to the scene that creating and querying operations are in the same process.

2) Do full-text search on existing index. This path corresponds to the scene that query from existing index separately.

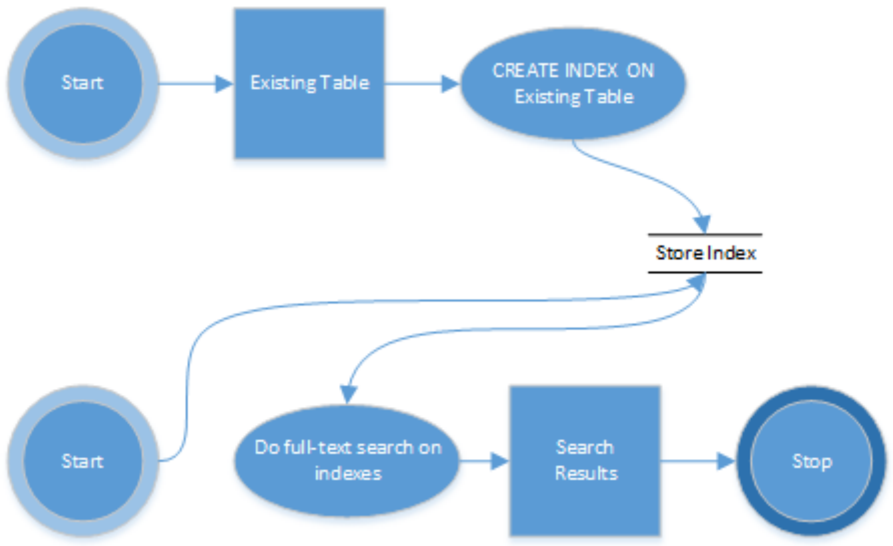

Start
Existing Table
CREATE INDEX ON Existing Table
Store Index
Start
Do full-text search on indexes
Search Results
Stop

Figure2. Data flow in full-text search

2.4 RDD Design

We can support two paths by RDD in Spark.

RDD is the bottom layer which will be executed by Spark Core, and we can define our operations by implementing RDD.

To support two paths of data flow, five operations need to be considered:

1) Create indexes based on existing data(table)

2) Reconstruct RDD from existing indexes

3) Do parallel full-text search on indexes

4) Data structure return to client

5) Global Top K

2.5 Storage Design

RDD operates on the index, and three things need to be considered:

1) Column Storage

2) Query Performance

3) Index sharing

Full-text search create indexes on specified fields which mean specified columns of a table, so we need to design column data storage strategies to minimize storage cost.

Query performance associates with column storage and how a task is actually executed.

Index sharing means we can support RDD reconstruction from existing indexes.

3. Implement details

3.1 Grammar

(1) Create index from existing table

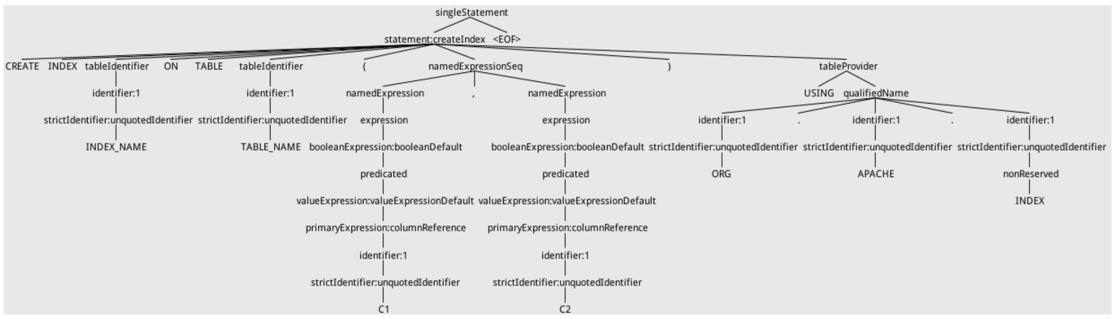
Spark SQL	**MySQL**

CREATE INDEX index_name ON TABLE table_name (column1,column2) USING tableProvider	CREATE TABLE table_name(column1 TYPE, column2 TYPE, FULLTEXT (column1,column2)) ENGIN=InnoDB;
	ALTER TABLE table_name ADD FULLTE XT(column1, column2)
	CREATE FULLTEXT INDEX index_name ON table_name (column1, column2)

Parameter interpretation can be found In the previous section.

Here we add CREATE INDEX statement in SqlBase.g4.

Grammar tree is:



singleStatement
statement:createIndex <EOF>
CREATE INDEX tableIdentifier ON TABLE tableIdentifier (namedExpressionSeq) tableProvider
identifier:1
strictIdentifier:unquotedIdentifier
INDEX_NAME
identifier:1
strictIdentifier:unquotedIdentifier
TABLE_NAME
namedExpression , namedExpression
expression
booleanExpression:booleanDefault
predicated
valueExpression:valueExpressionDefault
primaryExpression:columnReference
identifier:1
strictIdentifier:unquotedIdentifier
C1
C2
USING qualifiedName
identifier:1 . identifier:1 . identifier:1
strictIdentifier:unquotedIdentifier
ORG
APACHE
nonReserved
INDEX

Figure3. CREATE INDEX grammar

(2)Do full-text search from index

Spark SQL	MySQL

SELECT *
FROM index_name
WHERE TERMQUERY|FUZZYQUERY|PH RASEQUERY|
PREFIXQUERY|QUERYPARSER
(‘field, ’queryString’,[‘maxEdits’,] ‘topK’)
	SELECT * MATCH(column1,column2)
	AGAINST (‘query string’ [search_modifier])
	FROM table_name
	WHERE MATCH(column1,column2)
	AGAINST (‘query string’ [search_modifier])

Besides parameter interpretation in design, here are details of our implement.

Parameter interpretation:

1) TERMQUERY|FUZZYQUERY|PHRASEQUERY|PREFIXQUERY|QUERYPARSER are different filters which are one-to-one match with query type in lucene.

2) maxEdits is only valid for FUZZYQUERY.

3) field means which column to be searched, field in QUERYPARSER is a defaultField, if we want to do complex search, just set it any string.

4) topK means the size of results returned finally.

Grammar tree is:

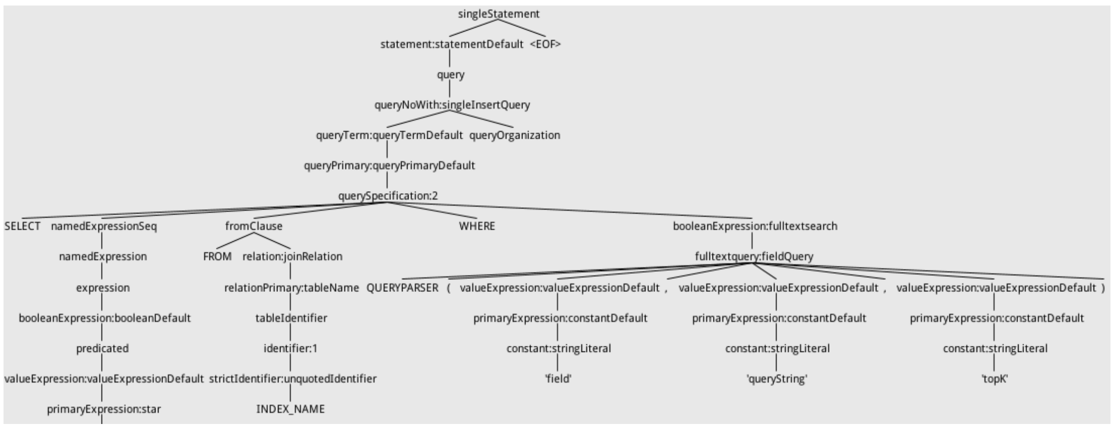

singleStatement
statement:statementDefault <EOF>
query
queryNoWith:singleInsertQuery
queryTerm:queryTermDefault queryOrganization
queryPrimary:queryPrimaryDefault
querySpecification:2
SELECT namedExpressionSeq
fromClause
WHERE
booleanExpression:fulltextsearch
namedExpression
FROM relation:joinRelation
fulltextquery:fieldQuery
expression
relationPrimary:tableName QUERYPARSER (valueExpression:valueExpressionDefault , valueExpression:valueExpressionDefault , valueExpression:valueExpressionDefault)
booleanExpression:booleanDefault
tableIdentifier
primaryExpression:constantDefault
predicated
identifier:1
constant:stringLiteral
valueExpression:valueExpressionDefault
strictIdentifier:unquotedIdentifier
'field'
'queryString'
'topK'
primaryExpression:star
INDEX_NAME

Figure4. Full-text search grammar

3.2 Translation

(1) Create Index Command

We create a new Command named CreateIndexTable in ddl.scala which 'create index' will be translated to , and it will be converted to CreateIndexTableCommand to create a table using org.spark.sql.index in MANAGED way. So index_name will be a table like others which means we can drop it if we need, and we use org.spark.sql.index to manage its underlying storage.

(2) 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.

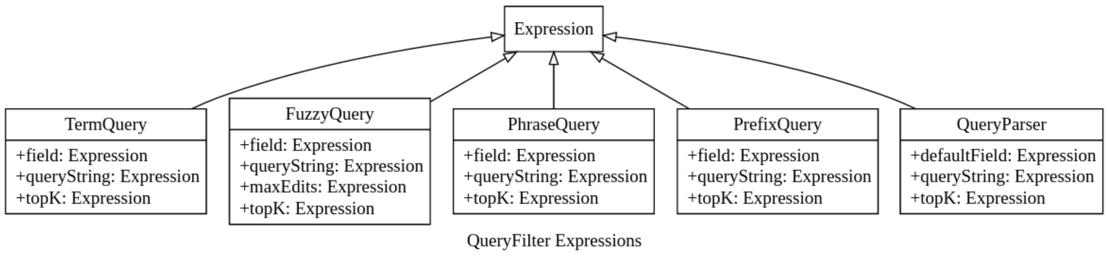


Figure5. QueryFilter Expressions

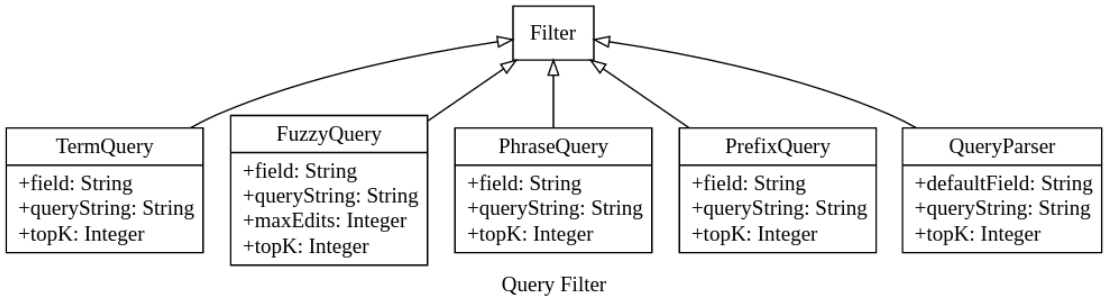


Figure6. Query Filter

(3) IndexRelationProvider and IndexRelation

We use IndexRelation to support low-level read/write operation by RelationProvider. I ndexRelation supports inserting data to LuceneRDD and reading query results from Lucene RDD with path parameter which points out index share directory. Underlying storage and access are all supported by LuceneRDD. Details will be shown in storage implement secti on.

Insert operation implements InsertableRelation to use data to reconstruct a new Lucen eRDD with specified table name or using existing RDD.

Scan operation implements PrunedFilteredScan to receive filters(TERMQUERY,FUZZY QUERY,PHRASEQUERY,PREFIXQUERY,QUERYPARSER) from previous steps. Operatio n details and processing of query results will be shown in next section.

3.3 RDD Implement

3.3.1 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 physi cal machine disk current task belong to , after we close current app, we can't reuse previ ous index build by LuceneRDD because of the distributed environment.

2)LuceneRDD lacks the ability to build index on an designated table.

3)Query results LuceneRDD returns have multiple partitions where data is locally sort ed, but final results higher layer received are not globally sorted.

3.3.2 Improvements

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

(1) Index build, write, read and storage

Data of original RDD we want do full-text search on is distributed which has multipl e partitions. Every partition will generate lucene index individually, Index build, write, rea d and storage are all manged by LuceneRDDPartition. LuceneRDD consists of serveral Lu ceneRDDPartition.

One LuceneRDDPartition processes data from one partition of original RDD and store index to directories(Every partition has two directories <indexDirectory,taxonomyDirector y> which will be used for general query and facet query) under the table_name_00index(t able_name is the table we create index on, 00index is just a suffix convention) on HDFS. LuceneRDDPartition will read indexes to do search from HDFS, and finally every partiti on returns a LuceneRDDResponsePartition(Iterator[SparkScoreDoc]).

We change lucene index storage to a HDFS directory we defined so it can be found for usage next time. When we build index on a table, indexes of this table will be store d in a HDFS directory named table_name_00index.

(2)LuceneRDD reconstruction

We provide the function to reconstruct a new LuceneRDD by reading indexes from

HDFS. It’s useful when we want to reuse these indexes.

All folders under the table_name_00index directory will be processed through for reco nstruction. According to reconstruction rules, every <indexDirectory,taxonomyDirectory> wil l be used to build a new LuceneRDDPartition. Reconstruction rules are closely related to storage strategies. Details will be explained in storage strategies.

(3) Infer Schema

Only providing schema of query results can Spark SQL recognize data structure and print data correctly. According to the storage strategies which will be explained in next se ction, query results will be converted to InternalRow based on the schema inferred from i ndex’s fields’ data type or original data type, and additional StructFields will be added to schema.

(4) Global ranking

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

(5) Rich interface

Besides interfaces supplied by original implement, we support additional interfaces:

Interpretation	**Interface**
Build index on existing_data, store index to ‘path’ directory	LuceneRDD(existing_data,"path")
Build index on ‘field1’ and ‘field2’ from e xisting_data, store index to ‘table_name’ dir ectory. True means store all data, false mea ns store part of data.	LuceneRDD(df,"path",Seq[String]("field1",”fie ld2”), true)
Reconstruct LuceneRDD from existing path ‘path’	LuceneRDD(sparkSession, “path”)

3.4 Storage

3.4.1 Index Sharing

For sharing, we can store index to HDFS in designated way. For example, we build index named ‘test_index’ on table ‘table_name’, and actual storage in HDFS is shown in figure 7:

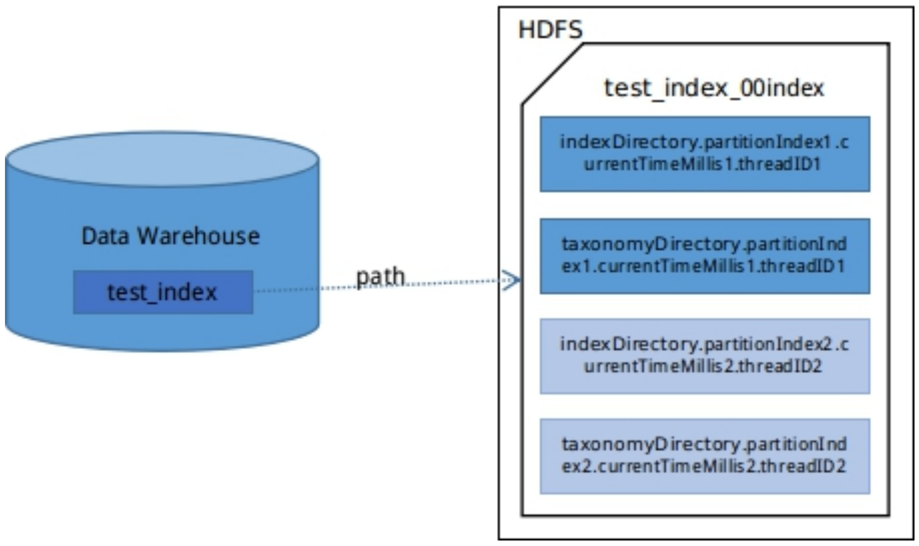

HDFS
test_index_00index
indexDirectory.partitionIndex1.currentTimeMillis1.threadID1
taxonomyDirectory.partitionIndex1.currentTimeMillis1.threadID1
indexDirectory.partitionIndex2.currentTimeMillis2.threadID2
taxonomyDirectory.partitionIndex2.currentTimeMillis2.threadID2
Data Warehouse
test_index
path

Figure7. Index sharing

test_index is a table like others which has been recorded in warehouse, its metadata contains path parameter ‘test_index_00index’ which will point to test_index_00index directo ry on HDFS. Path parameter is important because it points out share directory.

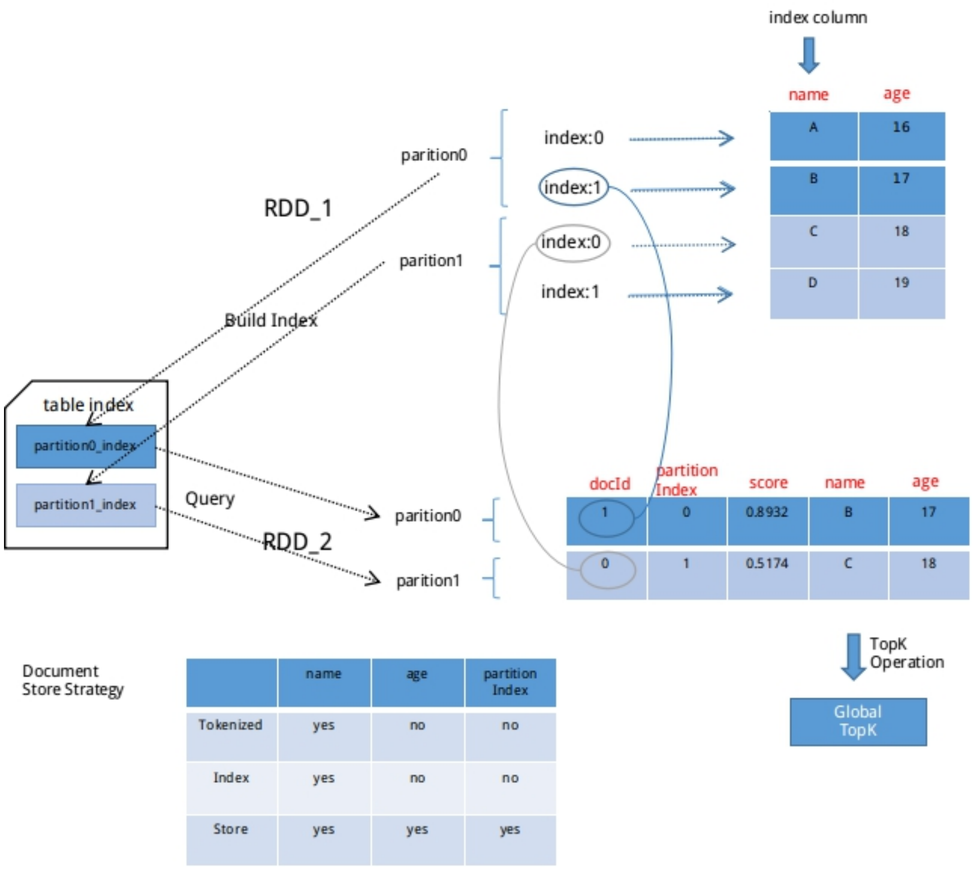
Every partition creates two directories called indexDirectory and taxonomyDirectory(‘ta ble_name’ has two partitions so it has two <indexDirectory,taxonomyDirectory> pairs). part itionIndex means which partition created current index, currentTimeMills means creation ti me, and threadID means task thread id.

3.4.2 Storage Strategies

Storage Strategies are closely related to the format of query results because that quer y results need to show not only score column but also selected columns(SELECT stateme nt) of original data which means we need build connection between query results and orig inal data. There are two solutions:

(1) Store all data to index

1) Field(Column) storage strategies: All columns data will be stored but only column1, column2,...(columns which need to be built indexes on) can be tokenized, indexed and que ried.Tokenization means tokenize data of column designated, index means can be queried. Field(Column) stored in Lucene means we can obtain field’s value if fields(columns) index ed searched relate to our query string. This method will store whole row data to a docum ent, and original data can be easily extracted and reconstructed from query results.



	name	age	partition Index
Tokenized	yes	no	no
Index	yes	no	no
Store	yes	yes	yes

Figure8. Store all data to index

2) Schema inferred

①Basic schema: Basic schema is inferred from index. Conversion rules are shown b elow:

Original Type	Target Type
Integer	StructField(IntegerType)
Long	StructField(LongType)
Float	StructField(FloatType)
Double	StructField(DoubleType)
String	StructField(StringType)

Figure9. Conversion rules

In our case, fields conversion in document is:

	name	age	partition Index
Original Type	String	Integer	Integer
Target Type	StructField(StringType)	StructField(IntegerType)	StructField(IntegerType)

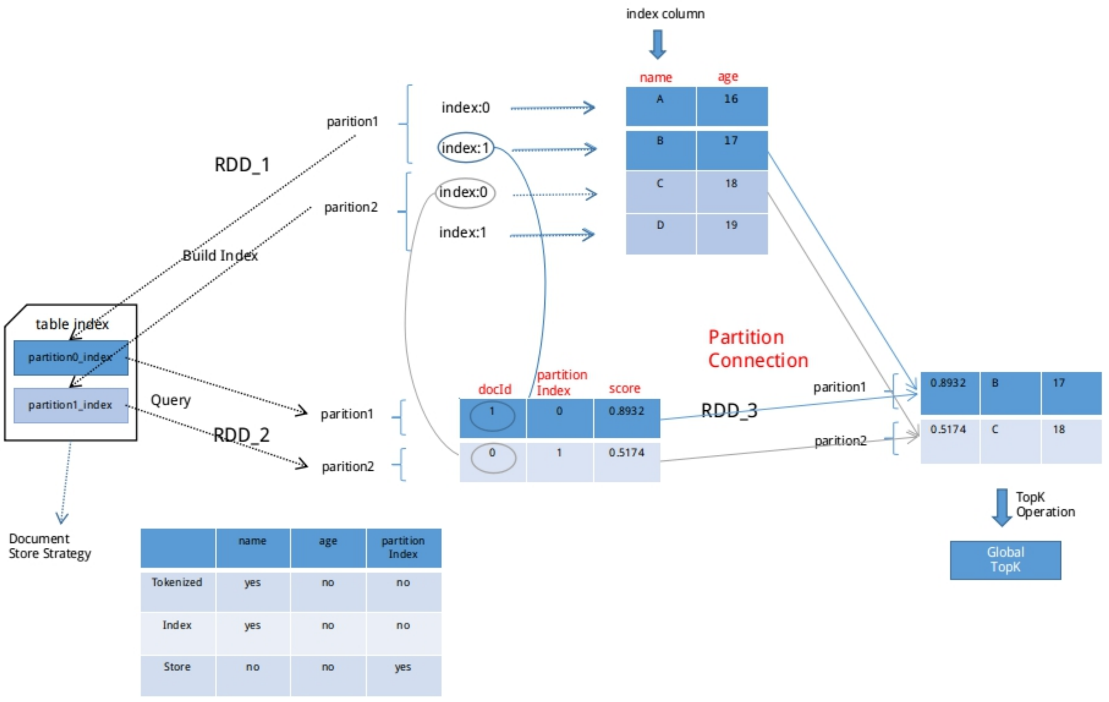
Figure10. Example fields conversion

②Additional StructField: Two StructFields will be added: docId(StructField(IntegerType)), score(StructField(FloatType)). docId is equal to the index(offset) of original data.

In order to give sufficient information, docId and partitionIndex can tell us where current query result row comes.

(2) Store part of data to index

1) Field(Column) storage strategies: Only column1,column2,...(columns which need to be built indexes on) can be tokenized, indexed and queried. Pay attention that only partitonIndex is stored. Other fields(columns) are not stored which means we can't reconstruct original data just from indexed. So, we have to build connection between original table(or RDD) and current row queried from index so that we can reconstruct every row data and show them in console. We need to build connection between every partition of query results and every partition of original data.



name	age
A	16
B	17
C	18
D	19

docId	partition Index	score
1	0	0.8932
0	1	0.5174

0.8932	B	17
0.5174	C	18

	name	age	partition Index
Tokenized	yes	no	no
Index	yes	no	no
Store	no	no	yes

Figure11. Store part of data to index

2)Schema inferred

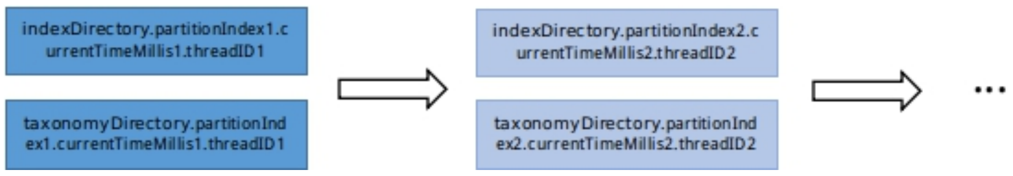
①Basic schema: Basic schema here is equal to the schema of original data. We just need to extract it using existing interface.

②Additional StructField: Only score StructField is added to basic schema.

3.4.3 RDD reconstruction rules

Reconstruction rules are closely related to storage strategies because of the partition connection. We need to ensure that current original data partitions are the same as when we create indexes so that we can use alignment operation to do partition connection. Partition changes will only influence dodId and partitionIndex columns of query results in "store all data to index" strategy but in "store part of data to index" strategy, it will lead two wro

ng query results because of the wrong connection.
Index directories are read sorted by partitionIndex in ascending order which ensure Lucene RDD reconstructed has the same partition order as original data. Every <indexDirectory,taxonomyDirectory> pair will create a LuceneRDDPartition.



indexDirectory.partitionIndex1.currentTimeMillis1.threadID1
taxonomyDirectory.partitionIndex1.currentTimeMillis1.threadID1
indexDirectory.partitionIndex2.currentTimeMillis2.threadID2
taxonomyDirectory.partitionIndex2.currentTimeMillis2.threadID2
...

Figure12. Reconstruction rules

3.4.4 Comparison

Comparison are shown below:

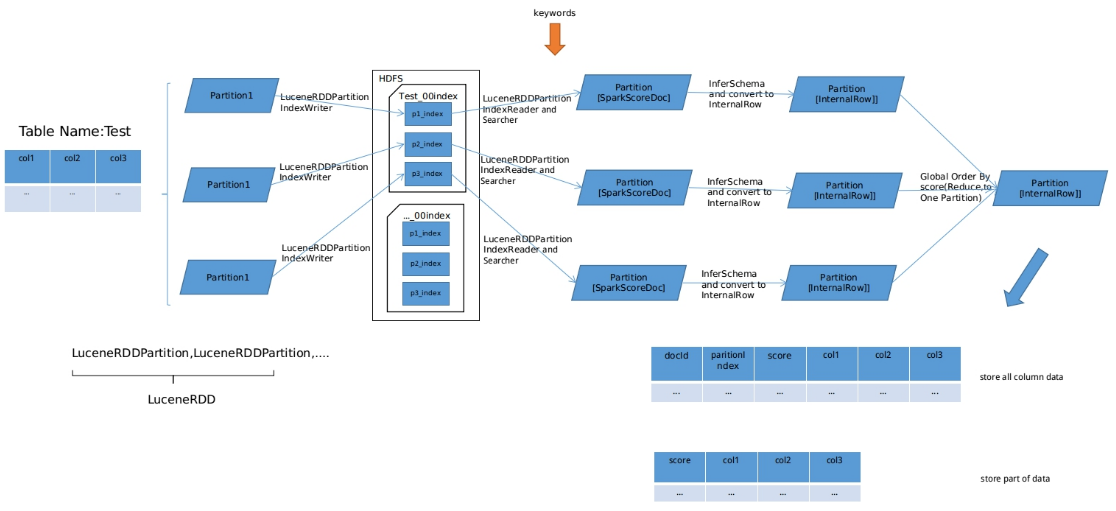
	Store all data	Store part of data
Column Storage Volume	all data of a table	only specified tokenized columns' data(minimal)
Query Performance	high(no need connection building)	low(connection building)

3.5 Data processing overview

Processing logic is shown in two perspectives below: Class perspective and Spark job perspective.

(1)Class perspective

All classes in this graph have been discussed before. So we don't get into details again.



keywords
Table Name:Test
col1
col2
col3
Partition1
LuceneRDDPartition IndexWriter
HDFS
Test_00index
p1_index
p2_index
p3_index
..._00index
LuceneRDDPartition IndexReader and Searcher
Partition [SparkScoreDoc]
InferSchema and convert to InternalRow
Partition [InternalRow]]
Global Order By score(Reduce to One Partition)
LuceneRDDPartition,LuceneRDDPartition,....
LuceneRDD
docId
paritionIndex
score
store all column data
store part of data

Figure13. Class perspective

(2)Spark job perspective

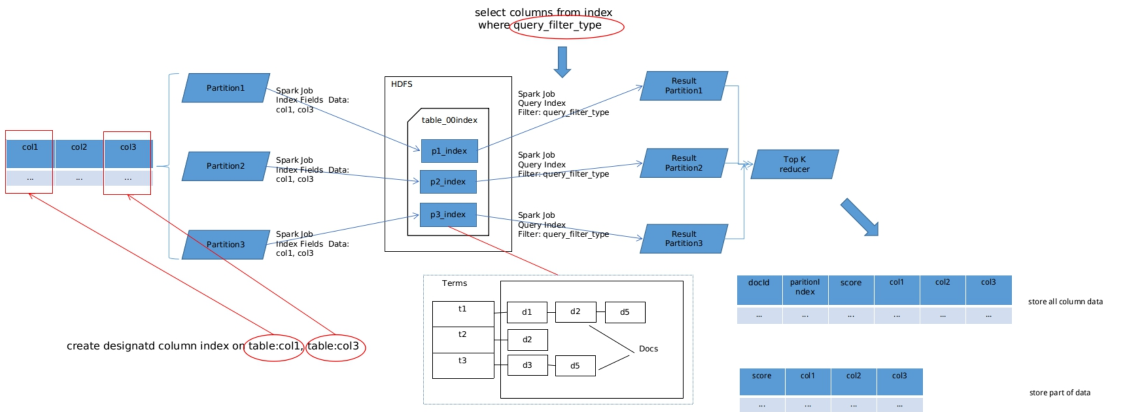

select columns from index where query_filter_type
col1
col2
col3
...
Partition1
Partition2
Partition3
Spark Job Index Fields Data: col1, col3
Spark Job Index Fields Data: col1, col3
Spark Job Index Fields Data: col1, col3
HDFS
table_00index
p1_index
p2_index
p3_index
Spark Job Query Index Filter: query_filter_type
Spark Job Query Index Filter: query_filter_type
Spark Job Query Index Filter: query_filter_type
Result Partition1
Result Partition2
Result Partition3
Top K reducer
Terms
t1
t2
t3
d1
d2
d5
d2
d3
d5
Docs
create designatd column index on table:col1, table:col3
docId
paritionIndex
score
col1
col2
col3
store all column data
score
col1
col2
col3
store part of data

Figure14. Spark job perspective

As we can see, there are four 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 to a HDFS directory named table_name_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.

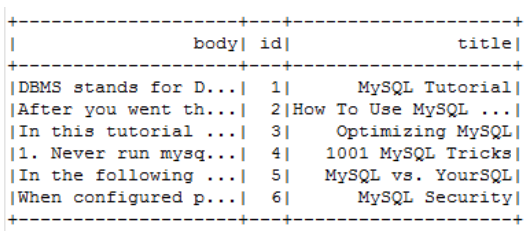
4)Reduce results from different jobs to one partitions for global top K ordered by score field.

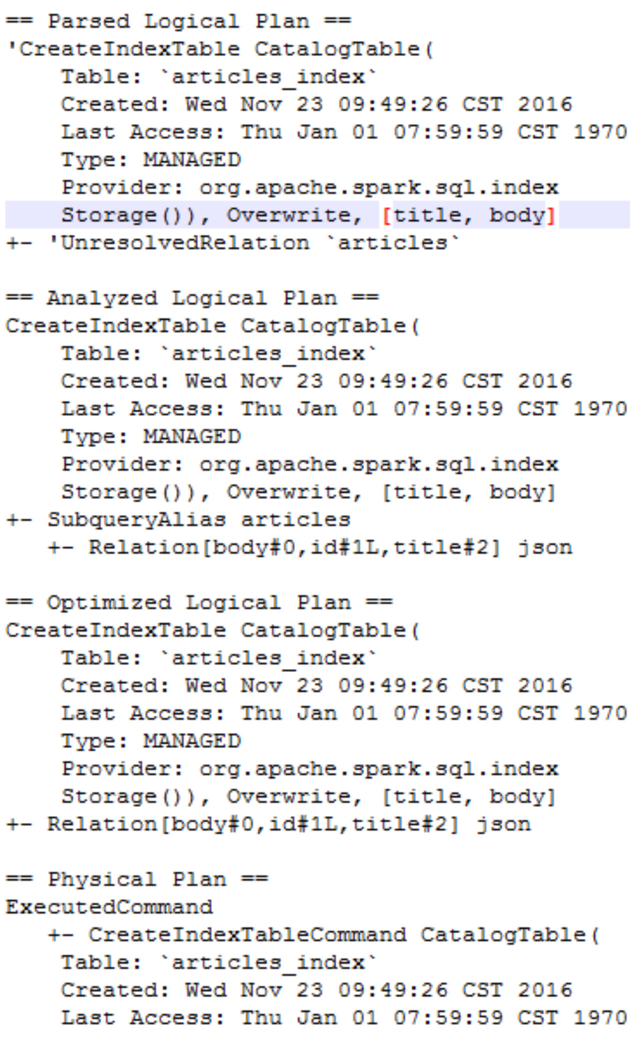
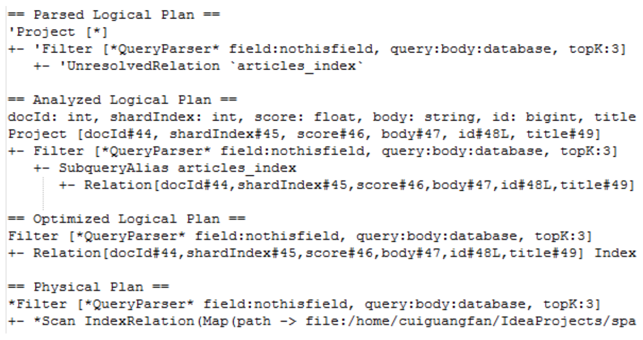
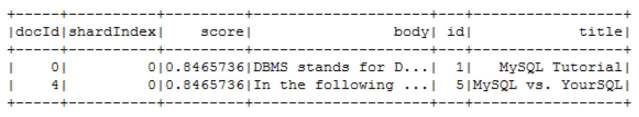
4. Experimental Results

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()	
	`+--------------------+---+--------------------+`
	`|                body| id|               title|`
	`+--------------------+---+--------------------+`
	`|DBMS stands for D...|  1|      MySQL Tutorial|`
	`|After you went th...|  2|How To Use MySQL ...|`
	`|In this tutorial ...|  3|    Optimizing MySQL|`
	`|1. Never run mysq...|  4|   1001 MySQL Tricks|`
	`|In the following ...|  5|   MySQL vs. YourSQL|`
	`|When configured p...|  6|      MySQL Security|`
	`+--------------------+---+--------------------+`

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 W HERE QUERYPARSER(‘nothisfield’, ’body:database’, ‘3’)”) // Explain query index logical plan df_query.explain(true)	
df.show()	

```
== Parsed Logical Plan ==
'CreateIndexTable CatalogTable(
    Table: `articles_index`
    Created: Wed Nov 23 09:49:26 CST 2016
    Last Access: Thu Jan 01 07:59:59 CST 1970
    Type: MANAGED
    Provider: org.apache.spark.sql.index
    Storage()), Overwrite, [title, body]
+- 'UnresolvedRelation `articles`

== Analyzed Logical Plan ==
CreateIndexTable CatalogTable(
    Table: `articles_index`
    Created: Wed Nov 23 09:49:26 CST 2016
    Last Access: Thu Jan 01 07:59:59 CST 1970
    Type: MANAGED
    Provider: org.apache.spark.sql.index
    Storage()), Overwrite, [title, body]
+- SubqueryAlias articles
   +- Relation[body#0,id#1L,title#2] json

== Optimized Logical Plan ==
CreateIndexTable CatalogTable(
    Table: `articles_index`
    Created: Wed Nov 23 09:49:26 CST 2016
    Last Access: Thu Jan 01 07:59:59 CST 1970
    Type: MANAGED
    Provider: org.apache.spark.sql.index
    Storage()), Overwrite, [title, body]
+- Relation[body#0,id#1L,title#2] json

== Physical Plan ==
ExecutedCommand
   +- CreateIndexTableCommand CatalogTable(
    Table: `articles_index`
    Created: Wed Nov 23 09:49:26 CST 2016
    Last Access: Thu Jan 01 07:59:59 CST 1970
```

```
== Parsed Logical Plan ==
'Project [*]
+- 'Filter [*QueryParser* field:nothisfield, query:body:database, topK:3]
   +- 'UnresolvedRelation `articles_index`

== Analyzed Logical Plan ==
docId: int, shardIndex: int, score: float, body: string, id: bigint, title
Project [docId#44, shardIndex#45, score#46, body#47, id#48L, title#49]
+- Filter [*QueryParser* field:nothisfield, query:body:database, topK:3]
   +- SubqueryAlias articles_index
      +- Relation[docId#44,shardIndex#45,score#46,body#47,id#48L,title#49]

== Optimized Logical Plan ==
Filter [*QueryParser* field:nothisfield, query:body:database, topK:3]
+- Relation[docId#44,shardIndex#45,score#46,body#47,id#48L,title#49] Index

== Physical Plan ==
*Filter [*QueryParser* field:nothisfield, query:body:database, topK:3]
+- *Scan IndexRelation(Map(path -> file:/home/cuiguangfan/IdeaProjects/spa
```

```
+-----+----------+---------+--------------------+---+-----------------+
|docId|shardIndex|    score|                body| id|            title|
+-----+----------+---------+--------------------+---+-----------------+
|    0|         0|0.8465736|DBMS stands for D...|  1|   MySQL Tutorial|
|    4|         0|0.8465736|In the following ...|  5|MySQL vs. YourSQL|
+-----+----------+---------+--------------------+---+-----------------+
```

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

5. 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 tableProvider	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		