标签页 1

Iceberg supports full-text and vector indexing solutions

Background

With the development of LLM technology, Data lakes are increasingly being used in Machine Learning scenarios such as Model Training and feature storage. The processing requirements for data in ML scenarios are diverse, including batch scanning, vector retrieval, or full-text retrieval. Users hope that one copy of data can meet both of these requirements to simplify the data processing chain and reduce storage costs. Therefore, we need to build full-text and vector retrieval capabilities on the data lake to support the processing requirements of MultiModal Machine Learning in ML scenarios.
Currently, there are some open source projects in the industry, such as lancedb, that support the ability to scan, full-text, and vector indexing on a dataset, which has received increasing attention. We hope to introduce full-text and vector retrieval support on Iceberg to make up for Iceberg's shortcomings in MultiModal data processing and further expand Iceberg's usage scenarios.

Design goals

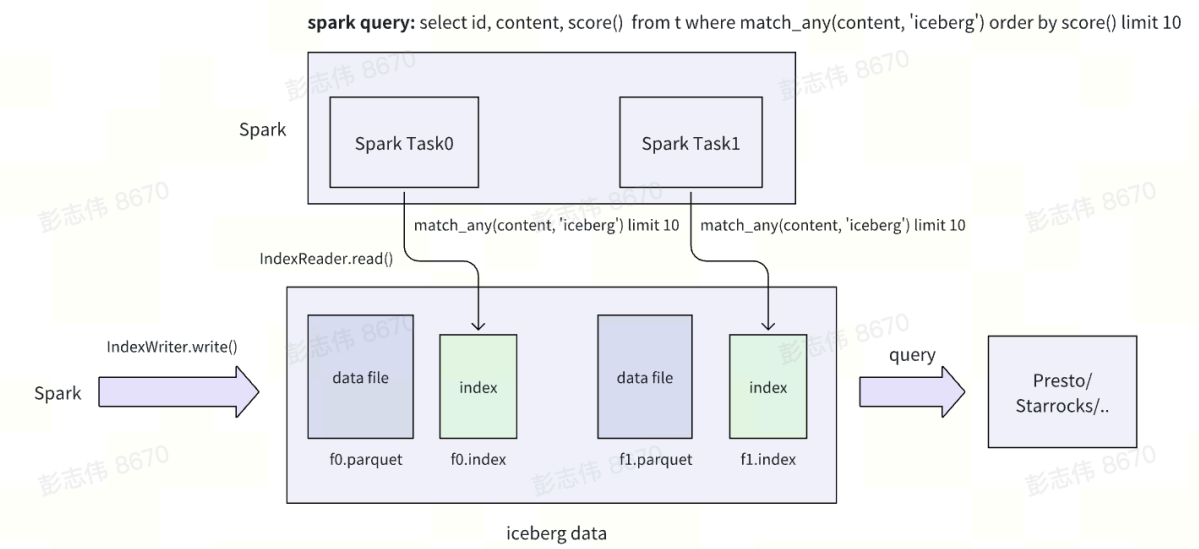
Based on open-source full-text search and vector search libraries, Iceberg's full-text search and vector indexing capabilities mainly include the following functions:

- Index building: Build full-text index and vector index on top of Iceberg based on spark or flink.
- Spark index query: spark sql query utilizes index information to accelerate query performance, achieving full-text and vector retrieval query capabilities on top of Iceberg.
- Other analysis engines support: abstract index query interface, support docking with other query engines, such as Presto, Starrocks, etc.

Scheme design

Overall architecture

The overall architecture is shown in the following figure. We will use Lucene as the library for full-text and vector retrieval. Lucence is a high-performance full-text retrieval library based on Java language, and also supports vector



indexing capabilities. It is currently widely used in the search field. We will also build full-text and vector retrieval capabilities based on this library.

For the query phase, Spark SQL pushes the search conditions and limit values down to the scan task node. The scan task queries the row_id (line number) list of matching records in the index file through the push-down conditions, and then queries the corresponding records in the corresponding parquet file through row_id. Other analysis engines such as presto/starrocks have similar query processes.

The specific details of the plan are as follows:

Index building

DDL syntax

The syntax is shown below, which supports building indexes and adding indexes to existing data when creating tables.

```
-- create table with index
CREATE TABLE t0 (
    id bigint,
    c1 string,
    c2 string,
    c3 array<float>,
    INDEX c3 USING VECTOR with (ann.algo = 'hsnw')
) using iceberg;

-- add index for exists table
ALTER TABLE t1
ADD INDEX text_idx(text INVERTED);
```

Index layout

Index files correspond one-to-one with data files. A data file corresponds to an index instance of Lucence. The index file name has the same prefix as the data file, which helps us quickly find the corresponding index file through the data file. The advantage of doing this is to simplify the version management of the index file. The version of the index file evolves with the data file. When a new snapshot of the data file is generated, a corresponding index file will also be generated.

Index write

During the data writing phase, index fields and row_id (line numbers) are written to lucence through the lucence IndexWriter interface.

```
// build index writer
FSDirectory directory = FSDirectory.open(Paths.get("index_directory"));
IndexWriterConfig config = new IndexWriterConfig(new StandardAnalyzer());
IndexWriter indexWriter = new IndexWriter(directory, config);

// add doc to index writer
Document doc = new Document();
doc.add(new TextField("row_id", 0L);
doc.add(new TextField("content", "This is a book about Lucene.", Field.Store.YES));
writer.addDocument(doc);

// commit and close
writer.commit();
writer.close();
```

First write the index data to the local disk, and then upload it to the HDFS directory.

Index query

Query statement

The score () function represents the full-text index score or vector retrieval relevance distance.

```
select id, content, score() from t
where match_any(content, 'iceberg') order by score() limit 10
```

Query process

- Spark Driver or starrocks FE node calls planTask () to generate an execution task list, and pushes the query to the scan task node under the push-down condition, and schedules it to the Executor node for execution.

- The Executor node receives the ScanTask task. First, it uses lucence to query the list of row IDs and score values corresponding to the search conditions in the index file. Then, it searches for the corresponding records in the data file using the row IDs. The implementation of Pseudocode is as follows:

```
IndexReader indexReader = new IndexReader(...);
ScoreDoc[] scoreDocs = indexReader.search(query, limit);
List<Long> rowIds = new ArrayList();
for (ScoreDoc doc: scoreDocs) {
    int docId = scoreDoc.doc;
    Document doc = searcher.doc(docId);
    long rowId = doc.get("row_id");
    rowIds.add(rowId):
}

ParquetReader reader = new ParquetReader(...);
List<Row> rows = reader.readRows(rowIds);
```

  We need to provide an interface for reading by line number on top of Parquet reader, which can use Parquet's page index indexing capability to accelerate the lookup performance.

- The GlobalSort operator receives the < row, score > list returned by each Task, summarizes the global TopN row list by score, and returns it to the Driver.

Read Index from Hdfs

Query needs to support direct reading of index data on HDFS. Lucence provides a Directory interface to abstract IO operations. We can implement Hdfs Directory to achieve the ability to read indexes on HDFS.

```
public class HdfsDirectory implements Directory {

    public abstract long fileLength(String name) throws IOException {
    }

    public abstract IndexInput openInput(String name, IOContext context)
    throws IOException {
    }
}
```

Object storage such as S3 can also be achieved by implementing corresponding directories.

Other matters

When rewriting or cleaning up a data file, it is necessary to synchronously rewrite or clean up the corresponding index file.

Implementation plan

- Step 1: Complete the Spark index building function, including syntax extension and index building function development.
- Step 2: Complete the development of Spark query index function, including index interface encapsulation, parquet reader support for searching by row number, etc.
- Step 3: Data file Rewrite and clean up support the processing of index files.
- Step 4: Connect to the presto connector to implement presto lookup iceberg index.