Text Search Support in Pinot

Author: Siddharth Teotia, Nov 2019

Introduction

This is a design document for supporting text search in Pinot. Text search is generally similar to other regular database search operations both in what user is looking to do (filtering) and how the database implements it efficiently (indexing). The TEXT column data is likely to be different than a regular STRING data. The text column could have log info (each column value a line from server log), resume text, yelp review (sentences) etc. The user should be able to search for phrases, terms/words, prefixes, regular expressions etc.

The initial POC (source code) was done with Lucene. This document will attempt to capture all the POC details along with complete design details and results from some more experiments.

Primer on Lucene

The central data structure in Lucene is an inverted index from terms/words to documents (essentially document IDs) in which they occur. For each term in the index, it also maintains additional data like position(s) of the term in a given document.

Document/Row is the unit of indexing (logically). Physically, the textual data in each row is parsed/tokenized into indexable terms and these are used to build an inverted index.

When we add Document to Lucene Index, we specify which fields are indexable fields. We can also specify if the indexable field should be physically stored by Lucene or not. In other words, if we just want Lucene to build the index for a text column (say skills or log data) in Pinot table, we should just wrap around that column in a Lucene document object, tell it to index but not store the field since it will anyway be stored in the forward index of Pinot.

Analyzer

Lucene comes with inbuilt Standard analyzer (essentially the text parser) that tokenizes the input text data into indexable tokens, removes stop words etc. The same analyzer is later used during query time to interpret the user written query and get the tokens which will be used to query the index. This process is also called as **Lexical analysis** and used during indexing and searching time.

- Remove stop words like ‘and’, ‘the’, ‘or’ etc -- these are not indexed.
- Break composite words into two individual words like publish-subscribe would be broken into ‘publish’ and ‘subscribe’
- Lowercasing
- Everything separated by white-space is by default an individual token.

- Word stemming (as per porter-stemming algorithm) such that if in the original text, there is token' bikes', then the user should be able to use query term as both 'bike' and 'bikes'.
- Removing trailing ('s) from words.
- Strip off commas and other punctuation.

Lucene's standard analyzer is built on Unicode text segmentation algorithm for tokenization (determining word boundaries). By default, this does not support stop words (we can provide a list though), stemming and removal of trailing 's. However, there is English Analyzer which uses the exact same tokenization mechanism for breaking text but uses a default set of english stop words (and, or, the, on etc), does stemming and removes trailing 's.

In addition to standard and english analyzer, Lucene has a ton of support for analysis in multiple different languages -- Chinese, Danish, Polish etc. This means that if a Pinot table has two text columns (one with English text and one with Chinese text), we can instruct Lucene to use a per column analyzer -- standard/english for first column, and Chinese analyzer for second column.

Azure search service which is based on Lucene provides 35 Lucene based analyzers for multiple different languages and several more proprietary analyzers based on their natural language processing technology.

NOTE: Only the term and phrase queries are analyzed. Prefix query, regex query aren't analyzed. They are just lower-cased and directly applied to index. The reason is that prefix and regex query support in Lucene is on a term basis. More on regex further in the document.

NOTE: Same analyzer should be used during (1) index building and (2) querying

Queries

Lucene supports different kinds of queries like (1) phrase query, (2) term query, (3) combination of these two using boolean operator(s), (4) prefix query and (5) regex query

Query syntax Examples: Say there is a text column named "col on which we have built a Lucene index. The following table gives some insight into how the lucene search queries look like.

col: "P1"	Look for documents where each document MUST contain the phrase P1 "blah blah blah" as is in column col. A phrase is **always specified in double quotes**
col: "P1" AND col:T1	Look for documents where each document MUST contain the phrase P1 "blah blah blah" as is and term T1 in column col

col: “P1” AND col: ”P2”	Look for documents where each document MUST contain two independent phrases “P1” and “P2” in column col
col: “P1” AND col:T1 AND col:T2	Look for documents where each document MUST contain the phrase “P1” as is along with terms ‘T1’ and ‘T2’ in column col
col: “P1” AND (col:T1 OR col:T2)	Look for documents where each document MUST contain the phrase “P1” as is and any of the terms ‘T1’’ or ‘T2’ in col

Lucene and Regex

Lucene query syntax supports regular expression searches and it can allow us to use text search instead of regexp_like at least for some common queries and getter better performance.

The combination of following for a single column text search query helps us to build a variety of single column search expressions

- One or more phrases,
- One or more terms
- Using AND, OR, NOT, parentheses to combine and group phrases and terms
- Single character wildcard for terms (te?t)
- Multi-character wildcard for terms (tes*t, test*)
- /regular expression/ - regex on a term **enclosed within forward slashes.**

Important thing to note here is that as a user if I am aware of the data and to some extent how Lucene breaks down the text into tokens, it may not be entirely necessary to write a Lucene regex query.

Let’s say you have a log file and each log line is a column value in Pinot text column. Take the example of pql query log file. Let’s say we want to extract all queries that have ‘GROUP BY’. We are aware that “GROUP BY” as a phrase will be delimited by whitespace and so I can do a phrase search for these two terms instructing

WHERE REGEXP_LIKE(column_name, 'GROUP BY')
WHERE TEXT_SEARCH(column_name, ‘\”GROUP BY\”’) -- **(regex lucene syntax not needed)**

Similarly let’s take the example of server log file. Say we want to find all documents/rows with exception. Now since exception in Java is something like “BlahBlahException”, ‘exception’ by itself can’t be tokenized and indexed as an individual term unless we preprocess the text and add delimiter. So we can use regular expression syntax with Lucene.

WHERE REGEXP_LIKE(column_name, 'Exception')
WHERE TEXT_SEARCH(column_name, '/.*Exception/') - **(using regex lucene syntax)**

Let's take a generic example. Say we have to look for documents/rows in a text column that MUST have two phrases "p1" and "p2". Take the example of searching the Apache access log for all GET requests that went to Firefox browser. So essentially we need to look for 'GET' and 'Firefox' and these could be in any order in the document. The two queries will look like following:

WHERE REGEXP_LIKE(column_name, 'get.*firefox|firefox.*get')
WHERE TEXT_SEARCH(column_name, 'get' AND 'firefox') (**regex lucene syntax not needed if we know how a typical log line in apache log generally looks like)**

Another example would be to search for documents/rows with a combination of multiple phrases and terms in any order. Say phrase p1 and term t1 are a MUST along with any one of term t2 or t3

WHERE REGEXP_LIKE(column_name, 'some complex regex to take care of all orderings in which p1, t1, t2, t3 can occur in a document'')

WHERE TEXT_SEARCH(column_name, '\"p1\" AND t1 AND (t2 OR t3)') -- **regex lucene syntax not needed**

Specifying the same query in text_search syntax is much cleaner and simpler as opposed to coming up with complex regex.

The documented limitations I see are that

- Wildcard can't be used in the beginning of search query or within phrases.
    - This should be fine as we are unlikely to have phrase queries like "mach*learning". Whoever is looking for phrase is ideally looking for a strong match of the exact phrase as is.
- Regex query (enclosed in forward slashes) can only be used for terms.
    - Ideally, we should use regex query in Lucene only when we aren't sure about the nature of our data. If we know what we are looking for can be met through a **phrase, term, prefix or any arbitrary combination** of these using AND,OR, NOT then we don't have to use Lucene regex query.
    - On the other hand, if we aren't sure if the data we are looking for would have actually been tokenized (and put into index), then we should use regex.

See the performance evaluation section for examples on each of these queries. Also, go through the functional tests written as part of POC.

Input data

At a high level, the input data can be generally divided into three broad categories:

Unclean data (e.g log search)

In this case, the underlying data is not homogeneous. The data is generally alphanumeric, could have IP addresses, URLs and the characters are likely to include pretty much anything. There could be a range of characters here (in addition to standard alphabet and numbers) like comma, quotes, hyphen, slash, parentheses, semicolon etc.

On such data, the user is likely to use a combination of phrase, term, prefix and regex queries. Unless we preprocess the data, regex might be needed since the data is not clean and it is not possible to predict what content will be tokenized (and thus available in the index) or not tokenized (and thus available as part of some bigger token, e.g 'exception' as mentioned in previous section)

Nearly clean data (e.g search on yelp review, resume text)

In this case, the underlying data is generally a set of words and/or sentences where there is whitespace (or some delimiter) between words. The data can still comprise of punctuation like comma but the range of such characters is generally small.

In this case, the most appropriate set of queries would be phrase, term and prefix. User is unlikely to use regex queries since the nature of data is simple, is amenable to full tokenization resulting in almost every searchable word (barring stop words) available in the index.

Arbitrary language data

The grammar of each language is unique and so the word boundaries will be different. Although, Lucene supports analyzing multiple different languages, Pinot should initially support only English.

Design

Parser changes

A new built-in function (similar to regexp_like) will be introduced for the user to specify the column name and search expression. This function can then be used as part of the WHERE clause in the queries just like how other filter expressions are written by users.

The grammar change will be made in PQL2.g4

TEXT_MATCH(column_name, search expression, options)

TEXT_MATCH() will essentially be a new predicate type addition to the existing predicate types we already have in Predicate enum — IN, EQ, RANGE, IN etc.

Few queries to give an idea of how the queries will look like

- SELECT int_col, skills_text_col FROM MyTable WHERE TEXT_MATCH(skills_text_col, '\"Machine learning\" AND \"Tensor flow\"', options)

- SELECT COUNT(*) FROM MyTable WHERE INT_COL >= 1010 AND TEXT_MATCH(skills_text_col, '\"Distributed systems\"', options)

- SELECT int_col, skills_text_col FROM MyTable WHERE TEXT_MATCH(skills_text_col, '\"Machine learning\" AND gpu AND python', options)

The purpose of third argument “options” is to pass down some user specified information. When we build the AST, Pql2AstListener will create TextMatchPredicateAstNode where the information about the column name (identifier), search expression (string literal), options (string literal) will be stored as part of calls to addChild(AstNode node).

TextMatchPredicateAstNode is a new concrete implementation of PredicateAstNode similar to how we have for IN, BETWEEN, REGEX, comparison etc.

Once the AST is built, we build the broker request. The details for that (and how the broker request will be used to execute the query) are mentioned further in querying text index section

Data Type

As mentioned earlier in the input data section, the candidate data for text search is not similar to regular STRING column data. It implies that users probably won’t be doing other standard filter operations (like equality) which are generally done on STRING columns and made efficient through dictionary, inverted index etc.

For this reason, we have decided to introduce a new data type “TEXT”. If the user has one or more columns of this type in their schema, then segment generation code will automatically create a Lucene text index on such columns (per column). With new type, users won’t have to use any knob in SegmentGeneratorConfig or TableConfig or IndexLoadingConfig to specifically tell Pinot to create a text index.

The underlying reader/writer for this could simply piggy-back on reader/writer for BYTES type.

Another alternative could be to keep the types as BYTES (which we already support in Pinot). In this case, we can control the creation of text index through knobs in SegmentGeneratorConfig or TableConfig — something like setTextIndexColumns().

Rest of the document will refer to such columns as “text columns” for the purpose of discussion.

Storage

There will be a **Lucene index per text column**. These columns will not be dictionary encoded. Creating a dictionary for such kind of data is going to be extremely heap heavy and not useful since users are never going to run equality operations on these columns. So we will have raw data in Pinot's forward index and a Lucene text index — Lucene will only index and not store the raw data.

NOTE: During initial discussions for requirements, it was brought up that users are unlikely to project the text column. In other words, they will only be used as part of TEXT_MATCH() predicate for filtering. This allows us to not store the raw data at all and save storage. However, we decided to have a raw data in forward index for two reasons:

- **Migration** - In future, if/when we implement an in-house text search solution without Lucene, then we need to migrate existing indexes which will essentially require to re-index.
- **Real-time segment conversion** - When we finish building a MutableSegment and about to convert it into ImmutableSegment, we need to have the raw data for re-indexing.
    - We discussed that for text columns, realtime segment conversion could purely be a directory move to avoid indexing twice. However, this is unlikely to work since the docIDs, dictionary IDs for other columns will change when we convert the segment.

Lucene index directory and files

Lucene index is stored in a directory. Since there will be a Lucene index per text column, there will be as many directories under the Pinot segment directory. The directory structure will look something like this:

```
/Path/To/Pinot/SegmentDirectory
        /text_col1_lucene_index
                — index files (no subdirectories)
        /text_col2_lucene_index
                — index files (no subdirectories)
        /v3
                — all existing segment files (.psf, index_map etc)
```

Under lucene index directory, number of files during segment generation are as follows:

- write_lock() — the index writer does fcntl() on this file to get exclusive access.
- Data/metadata files where Lucene writes its dictionary and index info.

As part of POC, we did experiments to understand the number of files etc. For a single segment, we tried with 5million documents and 3 TEXT columns. So during segment

generation, 4 files (3 from Lucene) and 1 for Pinot's raw forward index were opened per text column.

Once the writer finishes, Lucene also builds a compound file (.CFS), so during querying, there is one file opened per text column. **Note that it is not opened for each query**. When we do ImmutableSegment.load() to mmap the segment, at that time we also load the Lucene index (which also mmaps the compound file) **and the file descriptor remains open thereafter**. More on the query execution further in the document.

I don't think we need to worry about the number of open files during segment generation. However, during querying we may run into problems for tables with several thousands of segments on a single server. For example, if there are 3 TEXT columns and 20k segments on a server we are going to have 60k open file descriptors. This is something that users of this features should be aware of.

NOTE: If users look into the Lucene index directory, they should see some additional metadata files which are quickly opened and closed. I don't see them being opened as part of any long running operation. For example, the write_lock file doesn't disappear once the writer finishes building/committing the index. It will be there but not opened. The CFS file is the only one that is kept open.

NOTE: There are some interesting relations between the number of files and heap overhead. Further information on this is captured in heap overhead section where we did experiments to understand the number of files under different situations.

Offline

We look at the design from both realtime and offline side. This section talks about building, loading and execution from offline perspective. Realtime is covered separately.

Building Text Index

SegmentColumnarIndexCreator will drive the index creation just like it does today as part of indexRow(GenericRow row) method as and when each row arrives.

Pre-index setup

- For each TEXT column, we need to instantiate the Lucene index creators/writers just like we do for forward index, inverted index.
    - As part of POC, a new interface "TextIndexCreator" was introduced. However, since Lucene index is just another inverted index we should leverage (and enhance) the existing InvertedIndexCreator interface and subclass it to write a new concrete implementation for building Lucene index.
- SegmentColumnarIndexCreator already knows the segment index directory. We create a lucene index directory under that for each text column as shown earlier. LuceneTextIndexCreator should handle all of that internally

indexRow(GenericRow row)

- For each TEXT column, call its textIndexCreator.add(columnValue) to add the column value in Lucene's inverted index.
- Add it to Pinot's forward index.

Lucene Text Index Creator

The text index creator is initialized per TEXT column by SegmentColumnarIndexCreator. It uses Lucene IndexWriter to add documents to the index. The following code snippet shows how the index creator adds information to the Lucene index.

```
docToIndex.add(new TextField(_textColumn, columnValue.toString(),
Field.Store.NO));
docToIndex.add(new StringField(DOC_ID_COLUMN_NAME,
String.valueOf(docIdCounter), Field.Store.YES));
```

_docToIndex is the custom document/row object that we are adding to Lucene with two fields

- TextField with name _textColumn (this is a Pinot TEXT column)
  - columnValue is the text data that will be **tokenized** and **indexed**.
  - FIELD.STORE.NO instructs Lucene to not store the text data and just simply index it.
- StringField with name DOC_ID_COLUMN_NAME
  - A monotonically increasing docID counter controlled by SegmentColumnarIndexCreator.
  - The purpose of adding this field is to identify each document uniquely within Lucene index such that during filter processing we can use these docIDs.
  - Ideally there should be no need for this field as Lucene internally maintains docID (as part of it's inverted index and this is based on insertion order) and this works perfectly for us. However, some Lucene documentation indicates that internal docID could change and users should not rely on it externally.

Once all the calls to indexRow() are over, for each text search column, we would have added as many documents into its respective lucene text index as there are rows in Pinot segment. Finally, when the segment generation finishes, we release the resources associated with index writer as part of SegmentColumnarIndexCreator.close() which closes (and commits) each text index writer. Commit essentially fsync's the lucene index directory and makes the changes available to reader.

The key thing here is the buffering threshold that we need to control. This threshold is used by Lucene to flush the buffered index data to index directory. The default value (Lucene's) is 16MB. In the POC code, we started with 500MB and did some experiments with this value to

understand the heap overhead and come up with a suitable value. The heap overhead section further in the document has more information on this.

Loading Text Index

When the server loads (mmaps) the ImmutableSegment as part of ImmutableSegmentLoader.load(), we create index containers for each column. The index container has pointers to readers for forward, inverted index, dictionary etc. All of this information is then available as part of ColumnDataSource.

For TEXT column, we have to do the exact same thing. During ImmutableSegmentLoader.load(), when we build the PhysicalColumnIndexContainer, for each TEXT column, we create a corresponding Lucene text index reader.

NOTE: As part of POC, a new interface "TextIndexReader" was introduced. However, since Lucene index is just another inverted index we should leverage (and enhance) the existing InvertedIndexReader interface and subclass it to write a new concrete implementation for reading Lucene index.

Lucene Text Index Reader

Created per TEXT column. Uses the Lucene IndexReader and IndexSearcher to mmap the index directory (compound file specifically). It provides a search(query_string) method which is invoked by the filter operator (see the next section). The query_string is the search expression as is that user specifies as part of TEXT_MATCH().

Per call to search(query_string), we parse the search query using Lucene query parser and search the index. The result of search is a set of matched documents which are returned back to the operator to iterate upon later.

Similar to LuceneTextIndexCreator, the **reader is also instantiated exactly once per column** during the segment load and the reference to it is available via ColumnDataSource for use by queries. This is exactly how other indexes are loaded and used in Pinot during execution.

Querying Text Index

This section talks about the query execution related changes to query the lucene index for a given query that has TEXT_MATCH(.....) as part of WHERE clause.

Building BrokerRequest

When the AST is walked to build the BrokerRequest, the information for WHERE clause is stored in an expression tree (FilterQueryTree) serialized into BrokerRequest. Just like how each PredicateAstNode is responsible for building it's FilterQueryTree (as a root node or as a child), TextMatchPredicateAstNode will do the same.

Example:

WHERE int_col = 2000 AND text_match(text_col, “\”machine learning\”)

The root of the FilterQueryTree will be the AND predicate (FilterOperator AND) with the following two children.

- int_col = 2000 for FilterOperator type EQUALITY
- text_match(....) for FilterOperator type TEXT_MATCH.

The FilterQueryTree will be then serialized into BrokerRequest.

Physical (execution planning) per segment

FilterPlanNode looks into the BrokerRequest, deserializes the FilterQueryTree and walks it to build the filter operator(s). For TEXT_MATCH, it does 3 new things:

- Read FilterQueryTree and get the information to build TextMatchPredicate. This will stash away the info on column name and search string.
- Use the predicate to build evaluator - raw value based text predicate evaluator
- Create a leaf level FilterOperator — TextMatchFilterOperator
    - TEXT_MATCH() unlike AND, OR is a leaf level filter operator

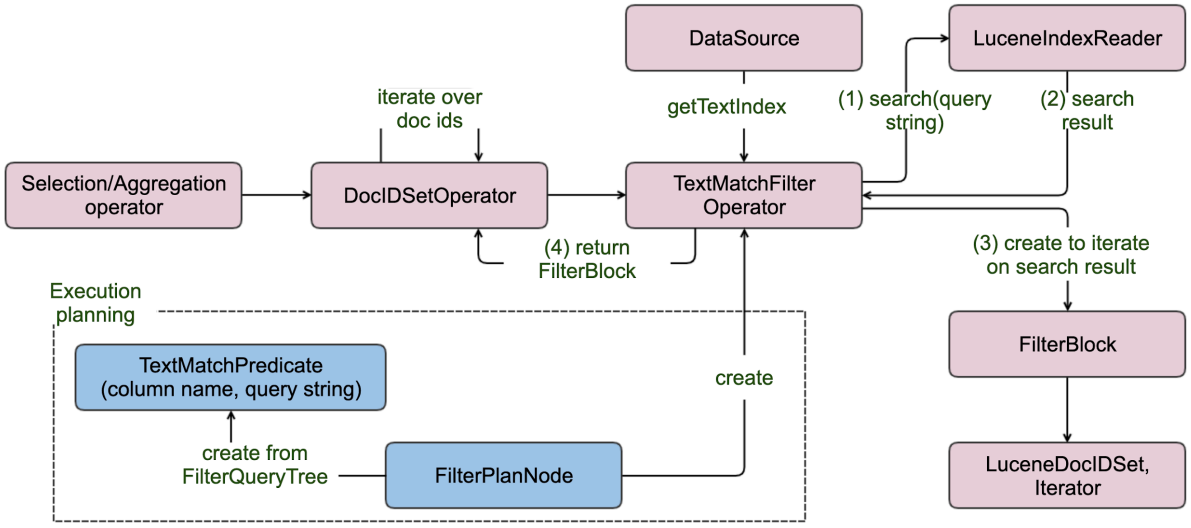
The TextMatchFilterOperator will contain the column data source, predicate evaluator and override the getNextBlock() method of the FilterOperator interface.

Filter processing

The filter processing for text_match will happen just like it happens for any other filter operator in Pinot. The below diagram shows how the execution will happen when we have a where clause as WHERE text_match(text_column_name, “search string”). Now whether we have one text_match() filter expression or multiple or a combination (using AND/OR) of this with others like EQUALITY, IN, BETWEEN etc, the execution flow will work as it does today.

The search query string is available to the filter operator through the predicate evaluator that was built earlier during execution planning. The text filter operator calls search() on lucene index reader exactly once for a query and gets the result. The result is then encapsulated inside the iterator because that is what is used by the rest of the execution code that works on docIDs.

Note that multiple queries will use the same LuceneTextIndexReader to search the text index with their respective query strings via TextMatchFilterOperator.



Server combine and broker reduce

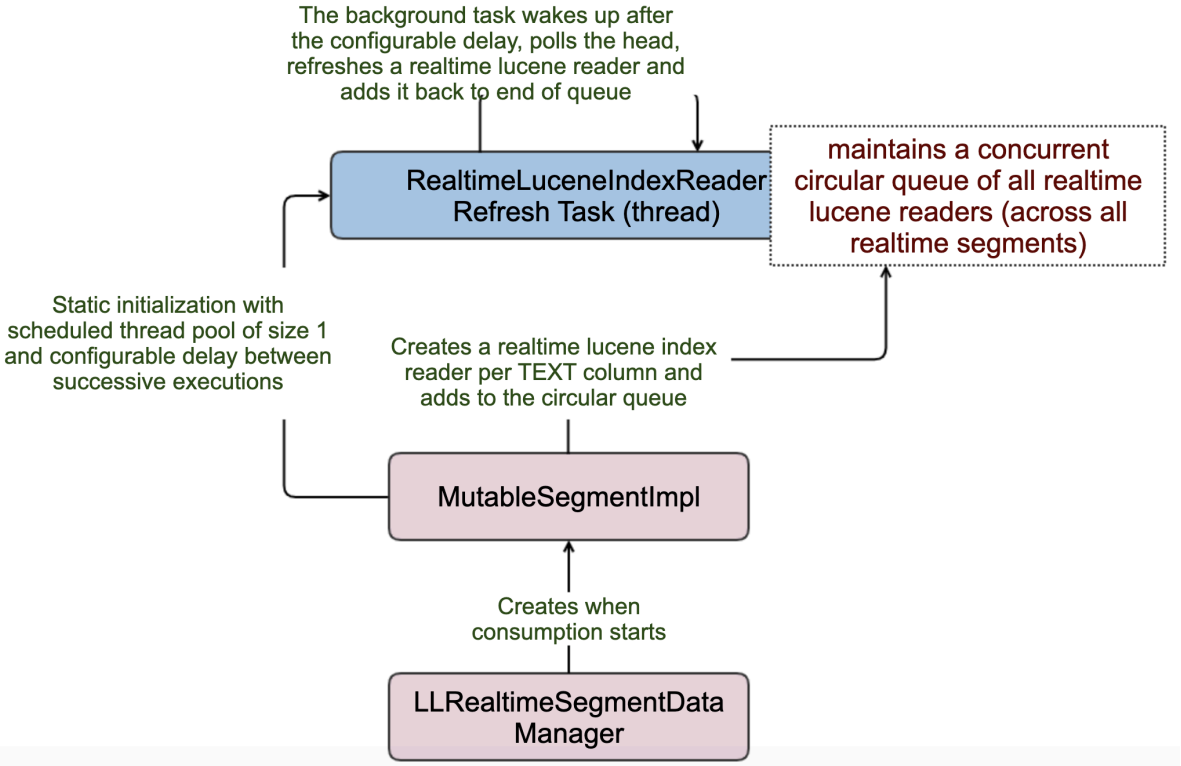
No changes are needed since the changes in filter processing code path are enough for this feature to fit into Pinot's execution engine.

Realtime

Lucene's IndexWriter which creates and maintains the index buffers the data (document's added to the index) and flushes them periodically to the index directory. However, the data is not visible to the IndexReader (and IndexSearcher) until the writer commits/closes which essentially fsync's the lucene index directory.

The index reader always looks at a point-in-time snapshot (of committed data) of the index as of the time reader was opened. As mentioned in previous section, for offline segments this works well since we load the index once during ImmutableSegment.load().
If we want to enable this feature for real-time (or hybrid tables), we need to be able to search the index (with some lag) as the writer is in progress.

Lucene supports NRT (near real-time) search by allowing to open a reader from a live writer and essentially allowing the reader to look at all the uncommitted index data from the writer. Again, just like any other index reader in Lucene, the NRT reader is also a point-in-time snapshot reader. This implies that the real-time reading code will have to periodically reopen the NRT reader (since the live writer would have made further progress since NRT reader was opened).



- There is a single background task (thread) on the server that maintains a concurrent circular queue across all the realtime lucene readers (across all realtime segments).
    - The task wakes up every X secs, polls the queue to get the realtime lucene reader at the head, refreshes it and adds it back to the queue.
    - We could also consider partitioning this queue at table level as a second level optimization. In that case, there will be as many background tasks as there are tables.
- Similar to OFFLINE (and similar to how we have today in Pinot), there is a Realtime Lucene reader per TEXT column.
- The reader is created by MutableSegmentImpl during its initialization. It is also added to the queue maintained by the refresh task.

Building and querying text Index in real-time

MutableSegmentImpl drives the consumption of rows for the in-memory real-time segment. As part of that, for TEXT columns, we will create raw forward index and Lucene based inverted index. This whole thing will be driven by existing index(GenericRow, RowMetadata) method in MutableSegmentImpl.

Note that currently realtime code supports no dictionary columns only for fixed width types. A separate change will be needed here to implement a variable width reader/writer that writes the raw forward index for variable width TEXT column.

Per TEXT column, an instance of RealtimeLuceneIndexReader is created and maintained by MutableSegmentImpl. This is similar to how today we have RealtimeInvertedIndexReader. As mentioned above in the offline section, the lucene reader and writer interfaces were

separate from InvertedIndexCreator and InvertedIndexReader in POC implementation. However, we should try to enhance the existing InvertedIndex{Creator|Reader} interfaces and write a new concrete implementation by extending them for Lucene.

Unlike the offline text index reader for Lucene, the real-time reader is both a reader and a writer. This is again similar to RealtimeInvertedIndex reader is implemented. It acts as both writer and reader while the segment is being consumed.

The reader will be available to the TextMatchOperator during query execution via the ColumnDataSource. So the query execution will happen as it happens today -- oblivious to the fact if the segment is offline or realtime.

How is the reader refreshed, reference counting and thread safety?

The realtime lucene reader maintains a SearcherManager -- a reference manager for thread safe management of IndexReader.

When the server query execution thread searches the text index through TextMatchFilterOperator, it acquires an IndexSearcher by bumping (atomically) up the reference count in SearcherManager. The searcher is then used to search the index for the query string and retrieve a set of matched documents. The IndexSearcher is then released through the searcher manager which decrements (atomically) the reference count. This use of searcher is similar to how it is done in offline except for acquire and release semantics.

In the beginning when SearcherManager is instantiated (that is during creation of Realtime lucene reader), it creates an IndexReader and a CURRENT searcher with reference count of 1. A call to acquire() in the foreground by the query thread will return the CURRENT reference of searcher. This reference could be of two kinds:

- A refreshed one (indicating different than the one returned by the previous call to acquire)
- Same as previous one

In both cases, the reference count would be bumped up by 1.

Concurrently in the background, the refresh task picks a realtime lucene reader from the queue and invokes a refresh() call on the reader's searcher manager. If the refresh is successful (implying index was really changed since the last time), the CURRENT searcher reference inside the searcher manager is swapped with the new one. If the refresh didn't do anything, then the CURRENT searcher reference inside the searcher manager will remain the same. The refresh() takes care of the fact that if index hasn't moved, nothing happens.

Any call to acquire the searcher in the foreground by the query thread will return this new searcher (if previous call to refresh wasn't a NOOP) and consequently will see the increased number of documents.

What happens if the call to refresh() by the background task is made at the same time as the call to acquire a CURRENT searcher in foreground by the query thread? The reference counting takes care of it.

Periodic reopening/refreshing of the NRT reader from the live writer is theoretically equivalent to flushing the writer followed by getting a new reader (at least this is what some Lucene documentation implies). But still for the initial iteration, the proposal is to go through the SearcherManager based refreshing instead of doing the flush ourselves (after every certain number of documents are added) in the real-time consumption code. It is yet to be determined conclusively if we can correctly abstract this out into two calls of flush() followed by reopen.

Please see the POC code to understand the current implementation of RealtimeLuceneIndexReader and refresh task. As part of POC, we had unit tests for testing the reference counting, multi threaded real-time and a cluster integration test to test the realtime functionality separately.

How often should the refresh happen?

This can be a configurable number and also depends on how the queue is managed. If we have a single queue across all realtime tables (and their segments), then the threshold should be lower (100ms between successive executions regardless of whether a single execution took more than 100ms). If the queue is managed on a per table basis (thus we have a refresh task to manage all the realtime segments of a table), then the threshold could be slightly higher (5secs between successive executions).

Another important factor in choosing a suitable default value is the consistency and performance trade-off. Assuming realtime index is really moving forward constantly:

- If the threshold is low, we flush too often and bear potentially lots of small I/Os. However, the realtime reader queries will get to see the new documents quickly
- If the threshold is high, we flush less often which increases the lag.

As part of productionizing the POC code, we need to do some more performance tests to get to a reasonable default value. This is also something that will vary on each use case basis.

Realtime to offline segment conversion?

Since we maintain a raw forward index, we will do whatever is done today for other columns -- re-index. One of the better ways to do this would be to handle the conversion for TEXT columns as simple directory move. However, this will not work since during segment conversion, the dictionary IDs and docIDs will change and the docIDs already there in the lucene index will be incorrect. So reindexing is necessary.

Performance Evaluation

Please see the section in POC document.

Heap overhead, GC, CPU

Some experiments were done as part of POC to understand the heap overhead, CPU and GC. This was done for offline segments for both (1) write path -- index building and (2) query path

The experiment was done for a single segment with 5million documents and 3 TEXT columns. For each column, the lucene TEXT index size was 137MB, 150MB and 150MB respectively. The raw data size was 1.2GB, 1GB and 1GB respectively for each column.

Write Path

As mentioned earlier in building text index section, Lucene IndexWriter buffers data in memory until a configurable threshold and flushes it once the threshold is hit. This data is buffered in JVM heap. Lucene's internal default value for this is 16MB and we experimented with multiple values of this parameter to estimate the heap overhead during segment generation.

Segment generation heap overhead when the 3 TEXT columns are stored as regular STRING -- dictionary, forward index, inverted index but no lucene inverted index.

Heap overhead due to Lucene objects	Young gen (Eden, S0)	Old gen	GC time	CPU usage
1.2GB	1GB, 252MB	1.8GB	34 collections 8secs	10-25% Stayed at 10% with spiking to 20-25% during GC. No major spikes.

Segment generation heap overhead when we create Lucene index on 3 TEXT columns

Since the test code itself has a non-trivial overhead -- 5million GenericRow objects and the hashmap nodes inside each one of them, the best way to look at the comparison between this and the previous table is to look at the second column in table below. This was gathered directly after taking heap dump and extracting the heap size associated with objects referenced by Lucene.

IndexWriter Buffer Size	Additional Heap overhead due to Lucene objects	Young gen (Eden, S0)	Old gen	GC time	CPU usage
500MB	~1GB	728MB, 1.5MB	3GB	62 collections 18secs	10-25% Stayed at 10% with spiking to 20-25% during GC. Two major spikes showed CPU usage at 70% when there was high GC activity
256MB	~400MB	37MB, 295MB	2.7GB	63 collections 14secs	10-25% Stayed at 10% with spiking to 20-25% during GC. One major spikes showed CPU usage at 70% when there was high GC activity
128MB	~250MB	50MB, 0	1.8GB	61 collections 13secs	10-25% No major spike unlike the previous two.
16MB	< 100MB				

Considerations for choosing the threshold:

- Lower the value, higher the number of times flush is invoked by Lucene during segment generation to flush the in-memory index state to index directory.
- Note only this, a flush results in a file.
    - With 500MB: at the end of segment generation we had 1 compound file per TEXT column -- this is the file that will later be kept opened for querying when the segment is loaded.
    - With 256MB and 128MB, there were 2 and 4 files respectively per TEXT column. These numbers can obviously be different depending on the nature of data (index size).
    - With 16MB, the number of files were extremely high.
- As mentioned earlier in file handling section, we need to try and keep the number of files low. It looks like 256MB could be a reasonable default value for the buffering threshold. Pinot users using this feature should be able to configure it.
    - Secondly, since once the segment is built, index is immutable for us so we can merge the index files before commit to a single file -- Lucene index writer provides an API to do this.
    - So in case, we want to set a really low value to keep a tight bound on heap usage, we can merge the files later to still have a single Lucene index file (per

TEXT column since we have an index per TEXT column). We can do this once we finish generating the offline segment and do indexSegment.close().

Query Path

The test ran 5000 queries sequentially and monitored the heap overhead, garbage collection and CPU. The segment has 5million documents, 3 TEXT columns (same as mentioned above for write path experiment). For comparison, the same experiment was also done when text data is stored in a STRING column with all the native Pinot indexes -- dictionary encoded forward index and inverted index.

Test type	Number of classes loaded	Young generation (Eden, S0)	Old generation	GC time	CPU usage
Querying the TEXT column	4400	300MB, 300KB	19MB	53 collections, 166ms	8%
Querying the STRING column	3725	60MB, 64KB	19MB	588 collections, 585ms	8%

Comments:

- First thing to note is that Lucene does not load the entire index in memory. It mmaps the index directory.
- A lot of short lived objects are created on the query path inside Lucene. This is true for Pinot as well which is why all the utilization is coming from Young generation and barely anything from Old.
    - However, Lucene creates additional objects to process the query, return the search result etc.
        - Note that lucene query result caching.
        - Also, search query was wrapped around ConstantScore search query to not score the results -- all matched docs have a constant score of 1.0
    - After digging into the code, one good optimization here would be to implement the collector interface which does not use a PriorityQueue since we aren't really interested in top docs/scoring. In other words, instead of using the in-built TopScoreDocCollector and TopDocsCollector, we can override the latter and get rid of priority queue. Something there documentation also suggests.
- The fact that in both the cases we are having short lived objects but still we are seeing relatively low number of collections for query on TEXT column probably points towards a memory leak either in the POC code that does the Lucene query processing or inside Lucene.

- Another very important observation is for the number of results requested. Usually text search queries will not be interested in seeing over 100k matched documents. The selectivity of text search queries should be relatively low (overall text data could be huge though).
    - Lower the limit, the lesser the heap overhead inside Lucene of these short lived objects required to store in-progress or final search result. In any case, providing our own Collector (as noted above) will help a lot.

Impact on segment generation time

An experiment was done with 10 segments, 5million documents per segment and 3 text columns per segment.

When building Lucene index, it took 21mins to create and load 10 segments. On the other hand, if we disable the text index and just have the regular dictionary and Pinot's inverted index, then it took about 7-8mins to create and load the segments.

This comparison is obviously not fair but just to give an idea that segment creation time will go up if the segment has text columns.