MEP 11 -- Support String DataType

Current state: "Under Discussion"

ISSUE: #1924 #3199 #4201 #4430 #4810 #5603

PRs:

Keywords: string tries hybrid search

Released:

Summary


```
This article introduces the importance of supporting the String data type. Review the data flow and data model of
the Milvus system. Design different StringField implementations for Segments according to different data sources
(Streaming or Historical).

For Streaming Segment, introduce the definition of StreamStringField.

For Histrical Segment, two implementations of HistoricalStringField are introduced.
    HistoricalStringField1  -- base on std::vectors and unorder_map
    HistoricalStringField2 -- base on marisa trie and std::vector

we also Introduce the processing logic of StringField on different working nodes.
```


Motivation

The data types currently supported by Milvus do not include the String type. According to the feedback in the previous issue list, the support of the String data type is expected by many users. One of the most urgent requirements of the String type is to support the primary key of the custom String type. Take the image search application as an example. In actual needs, users need to uniquely identify a picture with a string type value. The string type value can be the name of the picture, the md5 value of the picture, and so on. Since Milvus does not support the string data type, users need to make an additional int64 to string mapping externally, which reduces the efficiency and increases the maintenance cost of the entire application.

In addition to vectors, Milvus2.0 supports data types such as Boolean, integers, floating-point numbers, and more. A collection in Milvus can hold multiple fields for accommodating different data features or properties. Milvus pairs scalar filtering with powerful vector similarity search to offer a modern, flexible platform for analyzing unstructured data. Obviously, scalar filtering should support attributes of type String.

Public Interfaces

When users create a Collection, they can specify a String type Field in the Schema. The Field of the String type can of course be designated as the primary field at the same time.

In the system design, the type of string field is a variable-length character string, but a fixed size limit is set for the character string, such as 64KB, 256KB, etc. If the storage size of the string exceeds the limit value, the insertion fails.

Users can retrieve the previous field of string type according to the search/query interface. Users can add scalar filtering operations for string type Fields in search/query. The filtering operations include: "==", "!=" "<" ,"<=" ,">",">="

A piece of sample code is as follows

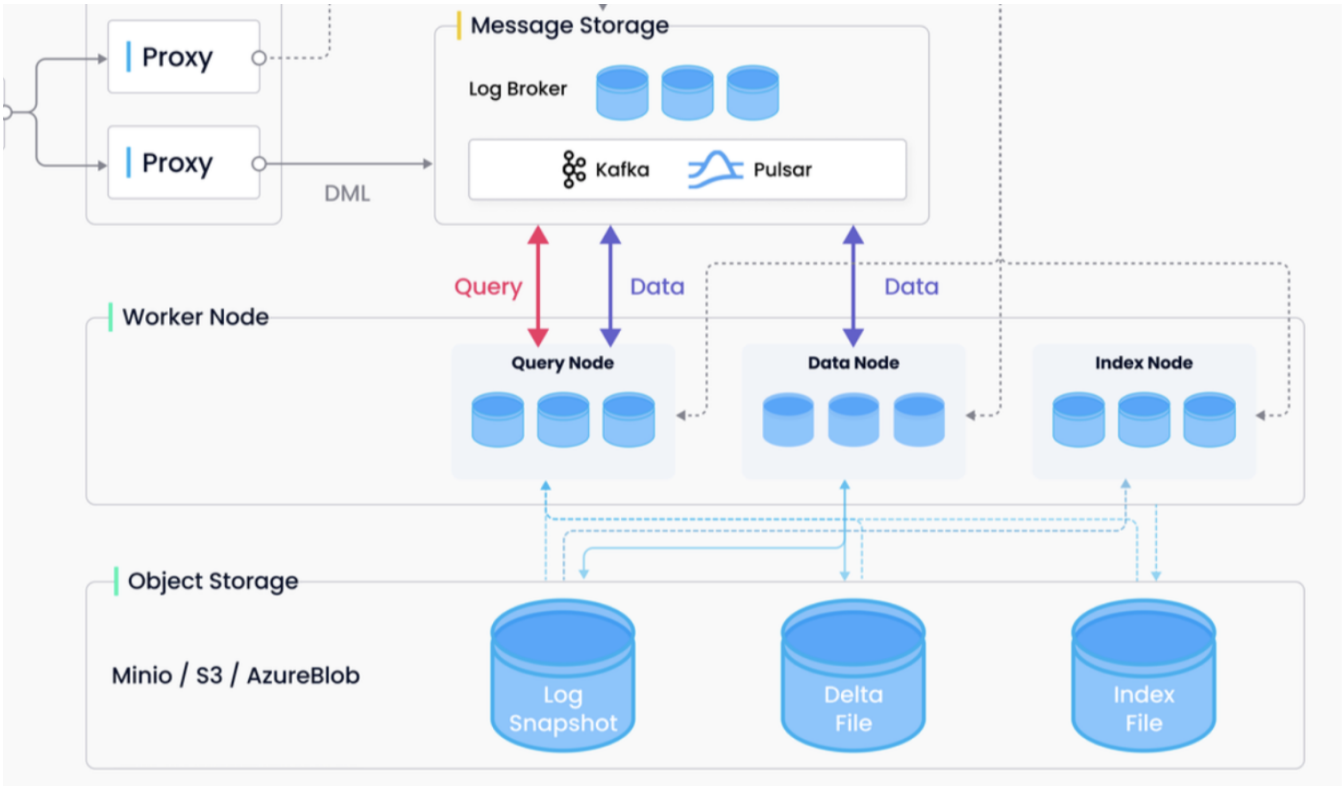
```
from pymilvus_orm import connections, Collection, FieldSchema, CollectionSchema, DataType
>>> import random
>>> schema = CollectionSchema([
... FieldSchema("film_name", DataType.String, is_primary=True),
... FieldSchema("films", dtype=DataType.FLOAT_VECTOR, dim=2)
... ])
>>> collection = Collection("film_collection", schema)
>>> # insert
>>> data = [
... ["film_%d"+str(i) for i in range(10)],
... [[random.random() for _ in range(2)] for _ in range(10)],
... ]
>>> collection.insert(data)
>>> # search
>>> res = collection.search(data=[1.0,1.0],
anns_field="films",
param = {"metric_type":"L2"},
limit=2,
expr = "film_name != 'film_1'")
```

Design Details

Before introducing the design scheme, let's briefly review the data flow of the milvus2.0 system.

DataFlow And DataModel

In milvus2.0, MessageStroge is the backbone of the entire system. Milvus 2.0 implements the unified Lambda architecture, which integrates the processing of the incremental and historical data. Milvus 2.0 introduces log backfill, which stores log snapshots and indexes in the object storage to improve failure recovery efficiency and query performance.



Incremental data flows into MessageStorage through the AccessLayer(Proxy), and nodes such as QueryNode and DataNode consume data from MessageStorage. For incremental data, DataNode persists the data to ObjectStorage in units of Segments to form Historical data. The ObjectStorage layer mainly stores historical data, including Log Snapshot, DeltaFile, and IndexFile. QueryNode can also load historical data from ObjectStorage. Index Node reads historical data from Object Storage and builds an index, and writes the index file back to Object Storage.

The data model of milvus2.0 mainly includes Collection, Partition, and Segment. Collection can be logically divided into multiple Partitions, for example, we can divide different Partitions according to date. The collection is physically composed of multiple Segments. A Segment contains multiple Entities, and an Entity is equivalent to row data in a traditional database. An Entity contains multiple Field data. A Field is equivalent to a Column in a traditional database. These Fields include those specified by the Schema when the user creates the Collection, and some hidden Fields added by the system, such as timestamps.

For OLAP purposes, it is best to store information in a columnar format. Columnar storage lets you ignore all the data that doesn't apply to a particular query because you can retrieve the information from just the columns you want. An added advantage is that, since each Field holds the same type of data, Field data can use a compression scheme selected specifically for the field data type.

In Milvus, the basic unit of reading and writing is the Field of the Segment. The basic module Storage encapsulates the reading and writing, encoding, and decoding of Field in the Object Storage. Therefore, the Storage module needs to support the String type Field.

DataNode's processing of String Field

DataNode's processing of String Field is the same as that of other Fields.

As mentioned earlier, segment data is stored by Fields. a Field is stored in multiple batches of small files. These small files and the data in it are arranged by the insertion order.

For example, Collection C1 has a field named field1. And a Segment of c1 is stored as m files. And the order of file names matters.

```
collection_id/partition_id/segment_id/field1/log_1
collection_id/partition_id/segment_id/field1/log_2
...
collection_id/partition_id/segment_id/field1/log_m
```

The Storage module of the system needs to provide support for String type data. In Milvus, we choose the Parquet format for String data storage.

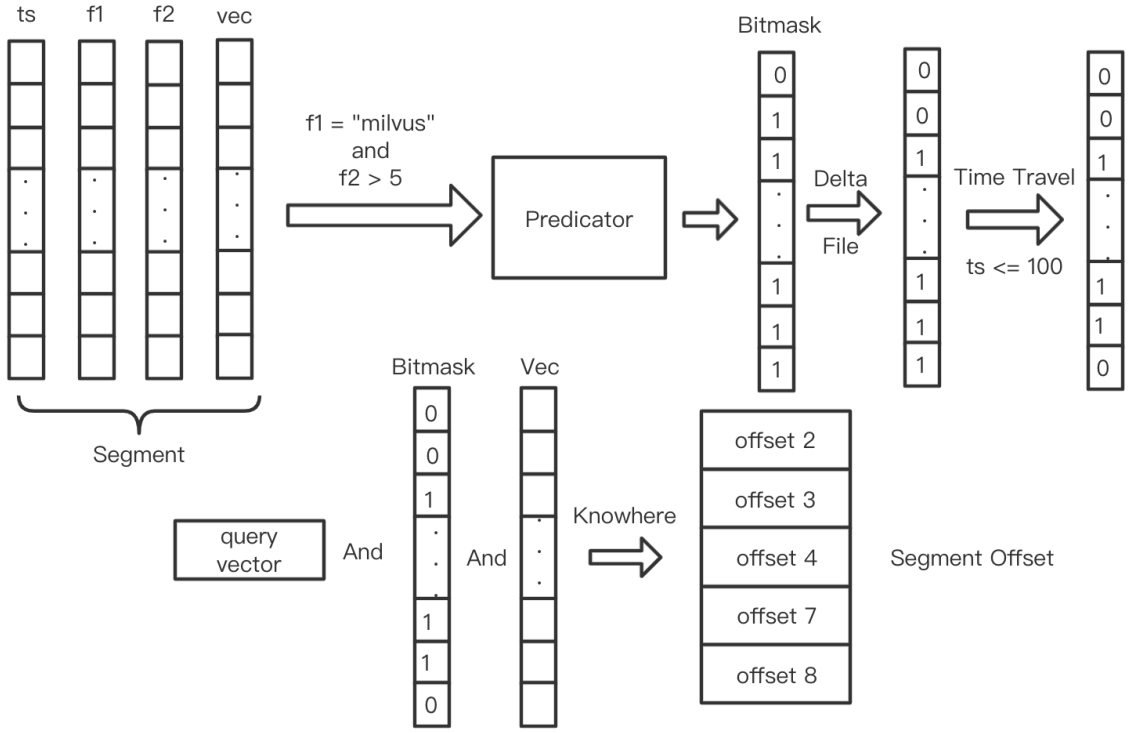
QueryNode's processing of String Field

Query and Search are two different operations, although their parameters are very similar. For the convenience of description, we use Search to represent these two operations in the following text, unless otherwise specified.

For QueryNode, the Query requires Historical and Streaming data. Historical data can be considered Immutable, and Streaming data, as it is continuously consumed from MessageStorage, is constantly added. Therefore, in order to optimize the Search, different designs need to be adopted for the two data sources.

First, let's introduce the general processing flow of QueryNode to the Search requests.

QueryNode first determines the set of segments involved with the Search request for a Collection. The search operation is performed on each segment to obtain the sub-results, and then all the sub-results are merged into the final result. Therefore, Segment is the basic processing unit of Search operations. We need to focus on the search operation in the Segment.



For segment hybrid search, a bitmask needs to be generated in advance. The size of the bitmask is the same as the number of Entities of the segment, and it corresponds to each Entity in the segment by position. For example, bistmask[i] corresponds to the i-th Entity in Segment. Bitmast[i] marks whether the corresponding i-th Entity should be filtered out. When bitmask[i] is 0, it means that the i-th Entity in the segment should be filtered out.

The bitmask roughly needs to go through the following 3 processing steps. First, a bitmask is generated through the query expression, then the bitmask is modified through the delete log in the DeltaFile, and finally, the bitmask is modified again according to the time-travel parameter and the timestamp field.

The final bitmask, together with the vector field and the target vector, participate in the approximate query and returns an array containing the positions of the Entities that meet the search conditions. For convenience, we named this array SegmentOffsets.

When the limit parameter of the Search is K, it means that only TopK results are needed. At this time, the length of SegmentOffsets is K. When doing a hybrid search, you can also specify output_field to retrieve the data of the corresponding Entity's Field.

It can be seen that segment offsets play a key role in segment processing. We need to calculate the segment offsets based on the approximate search on the vector and the filtering of the expression; we also need to extract the data of each Field in the corresponding entity of the Segment based on the offset.

We abstract the Stringfield Interface.

```
type StringField inteface {
extract(segmentOffsets []int32) []string

serialize() []bytes

deserialize([]bytes)

}
func Filter(expression string, field StringField) sgementOffsets []int32
```

We can limit the number of entities in Segment within a certain range, so the type of Segment offset is an int32.

The extract interface on Stringfield can retrieve the corresponding String according to the provided segment offsets.

The function Filter calculates the segment offsets on the Stringfield based on the expression string.

The serialize method serializes itself into a slice of bytes, which is convenient to store in ObjectStroage as an index file.

The deserialize method deserializes the index file into a Stringfield object.

An implementation of Stringfield of Historical Segment

The following gives a C++ definition of HistoricalStringField.

```
class HistoricalStringField1 {
std::vector<string> strs;

std::unordered_map<int32, std::vector<int32>> strOffsetToSegOffsets;

std::vector<int32> segOffsetToStrOffset;

std::vector<string>
extract(const std::vector<int32>& segmentOffsets);

std::vector<Blob>
serialize();

void
deserialize(const std::vector<Blob>&)

}

class Blob {
std::vector<char> blob_;
}
```

The strs member contains all the strings after deduplication and is sorted in ascending order.

The strOffsetToSegOffsets represents the mapping from String offset to segment offset. A string can appear in multiple entities, so the value type here is a vector.

SegOffsetToStrOffset represents the mapping from segment offset to String offset. Using string offset, we can retrieve the original string from strs.

Thus, opeations ("==", "!=", "<", "<=", ">", ">=") are transformed into binary search on strs to get the corresponding string offset, and then converted to segment offsets according to strOffsetToSegOffsets.

For the extract interface, you only need to retrieve the corresponding String according to segment offsets and segOffsetToStrOffset.

When there is no index file in the object store, QueryNode loads the original data from the object store and creates a Stringfield object based on the original data.

When the index file exists, QueryNode loads the index file from the object store, calls the deserialize method of Stringfield, and generates a Stringfield object.

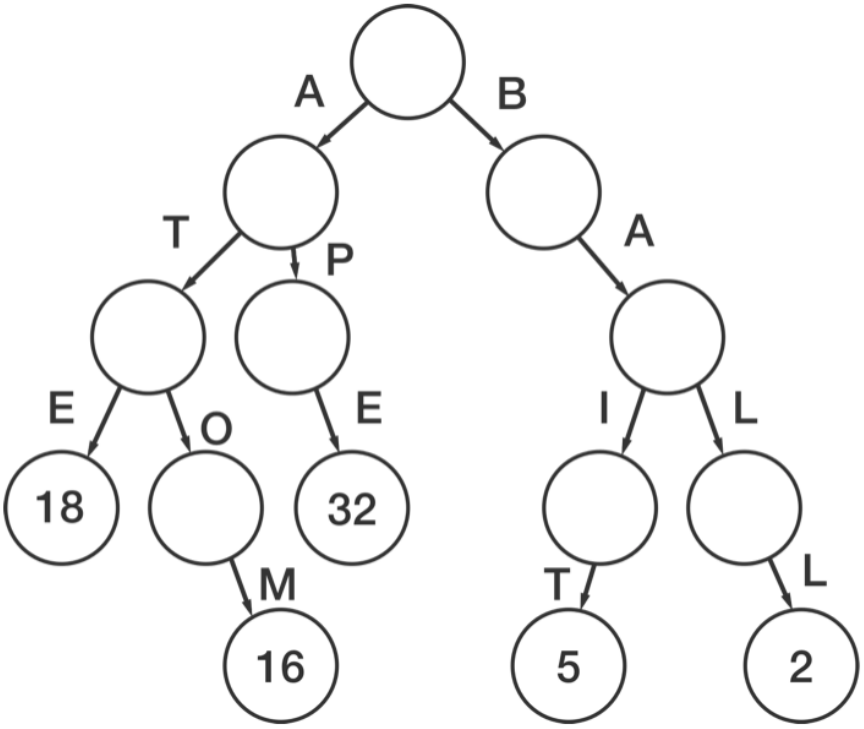
Another implementation of StringField of Historical Segment

Tries (also known as radix trees or prefix trees) are tree-based data structures that are typically used to store associative arrays where the keys are usually strings.

For one, nodes in the tree do not store keys. Instead, they each store parts of keys. Traversing down from the root node to a leaf allows you to build the key as you progress. Also, there doesn't need to be a value at every node. In fact, values are typically only associated with leaf nodes.

For example, below is a graphic that is a representation of a trie that contains the following associative array.

map = { 'ape': 32, 'ball': 2, 'atom': 16, 'ate': 18, 'bait': 5 }



A trie

The word "ape" is built by traversing the "a" edge, the "p" edge, and the "e" edge. When we reach a leaf, we can extract the value, which is 32.

Tries have many advantages over their counterpart, the hash table. They are used in many string search applications such as auto-complete, text search, and prefix matching. Radix trees, a kind of trie, are often used in IP routing.

There are many variant implementations of Trie. The main consideration when we choose an implementation is whether it can support reverse lookup and save memory. The reverse loop means that we can get the original string from Tries through a certain ID.

In the following tests, the memory usage was measured for 3 million unique Unicode Russian words; "lookup time " was measured on a lookup for one word.

	Description	memory usage (MegaBytes)	Unicode	Lookup time (nanosecond)	Reverse-lookup
PAT-Trie	a pointer-based implementation of PAT-trie (aka Patricia-Trie and Radix-Trie) and may use a lot of memory because of that.	242	No	333	No
HAT-Trie	Trie-HashMap hybrid. It is claimed to be the state-of-art Trie-like structure with the fastest lookups.	125	Yes	195	No
DA-Trie	Double-Array Trie C implementation	101	Yes	281	No
Marisa-Trie	memory-efficient recursive LOUDS-trie-based data structure implemented as C++ library	11	Yes	2010	Yes
DAWG	Directed Acyclic Word Graphs	2.8	Yes	249	No
Python-Dict	/	600			
Python-list	/	300			

MARISA-trie is a very memory-efficient recursive trie-based data structure implemented as a C++ library (repo). String data in a MARISA-trie may take up to 50x-100x less memory than in a standard Python Dict; the raw lookup speed is comparable; trie also provides fast advanced methods like prefix search. MARISA-trie also supports memory-mapped IO so it is possible to have an on-disk trie and reduce the memory usage even further.

The following gives another C++ definition of HistoricalStringField.

```
class HistoricalStringField2 {
MarisaTrie trie_;
std::vector<StrID> segOffsetToStrID;
std::vector<string>
extract(const std::vector<int32>& segmentOffsets);
std::vector<Blob>
serialize();
void
deserialize(const std::vector<Blob>&)
}
class Blob {
std::vector<char> blob_;
}
using StrID = int32;
```

Marisa Trie will return a unique StrID for each inserted string. We can get the original string through the StrID. Marisa trie can specify the value when inserting the key. Here we set the value to be an array of the segment offset. In addition, we need to maintain an additional mapping relationship from segment offset to StrIDs.

Thus, opeations ("==", "!=",) are transformed into lookup on the trie to get the corresponding segment offset .

Operations ("<=", "<", ">", ">=") needs to modify the underlying implementation of marisa to support range queries.

For the extract interface, you only need to retrieve the corresponding String according to segment offsets and segOffsetToStrID. When the StrID is obtained, the original string is retrieved through the reverse lookup of the trie.

An Implementation of StringField of Streaming Segment

The Streaming Segment mentioned here refers to the segment whose data keeps growing, also called the Growing Segment.

As the data keeps increasing, it is neither convenient to sort the string array nor to use trie. One possible implementation StreamingStringField is defined as follows.

```
class StreamingStringField {

std::vector<string> strs;

std::vector<string>
extract(const std::vector<int32>& segmentOffsets);

std::vector<Blob>
serialize();

void
deserialize(const std::vector<Blob>&)

}
class Blob {
std::vector<char> blob_;
}
```

The strs member contains the original string without deduplication and sorting, and the corresponding string can be retrieved directly by using segment offset as a subscript. string offsets are equivalent to segment offsets.

Operations ("==", "!=", "<", "<=", ">", ">=") are transformed into brute force search on strs to get the corresponding string offset, which is also the segment offset .

IndexNode's processing of String Field

The basic unit of IndexNode read and write is a Field of Segment. It is used to build an index file for a Field to speed up the search. You can build an index on either a vector field or a scalar field. When IndexNode receives an index request for a certain field of a Segment, it reads the original historical data from ObjectStorage, builds an index file for the Field, and writes the index file back to ObjectStorage.

For the String type Field, IndexNode needs to construct a HistoricalStringField object from the original data, and persist it in the object storage.

If the implementation of HistoricalStringField1 is adopted, what we need to persist are the strs member and the SegmentOffsetToStrOffset member.It is easy to recover strOffsetToSegOffsets based on segOffsetToStrOffset.

If the implementation of HistoricalStringField2 is adopted, what we need to persist are the trie member and the SegmentOffsetToStrID member. Marisa has implemented serialization and deserialization methods.

Compatibility, Deprecation, and Migration Plan

This is a new feature. There is no need to consider compatibility issues, no need to deprecate old interfaces, and no need to provide data migration tools.

Test Plan

- Unit tests
- CI tests

References

https://en.wikipedia.org/wiki/Trie

https://brilliant.org/wiki/tries/

https://github.com/s-yata/marisa-trie