MEP 9 -- Support delete entities by primary keys

Current state: "Under Discussion"

ISSUE: #6383 #7130

PRs:

Keywords: delete

Released:

Summary

This document describes how to support delete in Milvus. Milvus provides a new delete API to delete entities from a collection.

Motivation

In some scenarios, users want to delete some entities from a collection that will no longer be searched out. Currently, users can only manually filter out unwanted entities from search results. We hope to implement a new function that allows users to delete entities from a collection.

Public Interfaces & Behavior

Delete API can be used to delete entities in the collection, and the deleted entities will no longer appear in the results of the Query and Search request.

`collection_name` is the name of the collection to delete entities from.

`expr` is an expression indicated whether an entity should be deleted in the collection. Only the `in` operator is supported in the Delete API. Document of expression: https://github.com/milvus-io/milvus/blob/master/docs/design_docs/query_boolean_expr.md

`partition_name` is the name of the partition to delete entities from, `None` means all partition.

`Delete` returns after being written into the insert channel, which means the delete request has been reliably saved and will be applied in search/query requests. The type of return value is MutationResult, which contains several properties, and only `_primary_keys` will be filled.

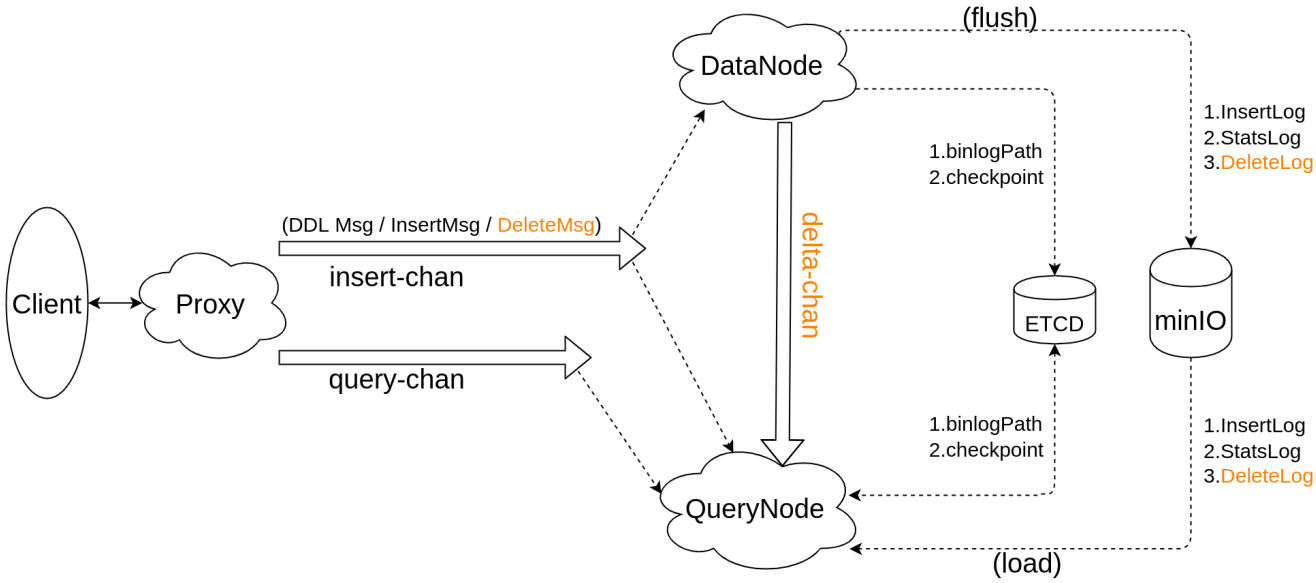
Same as Insert API, Milvus only guarantee the visibility of operations with one client. This means that, within the sequence of the operations "delete(), search()", the result of the search will not contains the entities deleted. Since different clients connect to different Proxy, the time between different Proxy is not exactly the same. So even if you manually call the delete method and the search method on two clients sequentially, it is uncertain whether the search request returns the deleted entities.

Currently, Milvus does not support dedup in inserting, so the delete operation will delete all satisfied entities.

Delete a non-existent entity is not an error, so delete() will not raise an Error.

```
def delete(self, collection_name, expr, partition_name=None, timeout=None, **kwargs)->MutationResult:
    """
    Delete entities with an expression condition.
    And return results to show which primary key is deleted successfully

    :param collection_name: Name of the collection to delete entities from
    :type  collection_name: str

    :param expr: The query expression
    :type  expr: str

    :param partition_name: Name of partitions that contain entities
    :type  partition_name: str

    :param timeout: An optional duration of time in seconds to allow for the RPC. When timeout
                    is set to None, client waits until server response or error occur
    :type  timeout: float

    :return: delete request executed results.
    :rtype: MutationResult

    :raises:
        RpcError: If gRPC encounter an error
        ParamError: If parameters are invalid
        BaseException: If the return result from server is not ok
    """
```

Design Details



In Milvus, Proxy maintains 2 Pulsar channels for each collection:

1. Insert channel, handle following msg types:
    a. DDL msg (CreateCollection/DropCollection/CreatePartition/DropPartition)
    b. InsertMsg
    c. DeleteMsg
2. Query channel, handle following msg types:
    a. SearchMsg
    b. RetrieveMsg

DataNode consumes messages from Insert channel only.

QueryNode consumes messages from both Insert channel and Query channel.

To support delete, we will send DeleteMsg into Insert channel also.

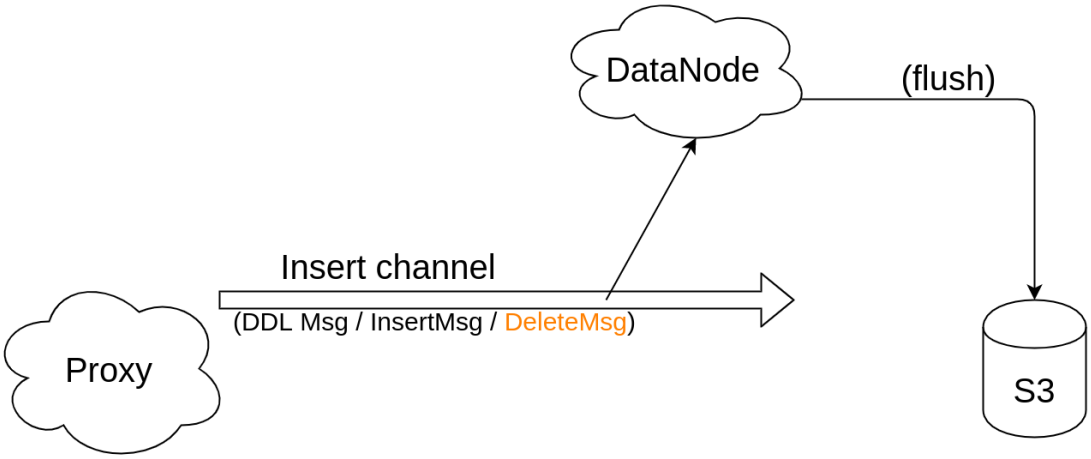
Since Milvus's storage is append-only, `delete` function is implemented using soft delete, setting a flag on entity to indicate this entity has been deleted.

This solution needs:

- record deletion offset in Milvus
- let algorithm library support to search with a bitset

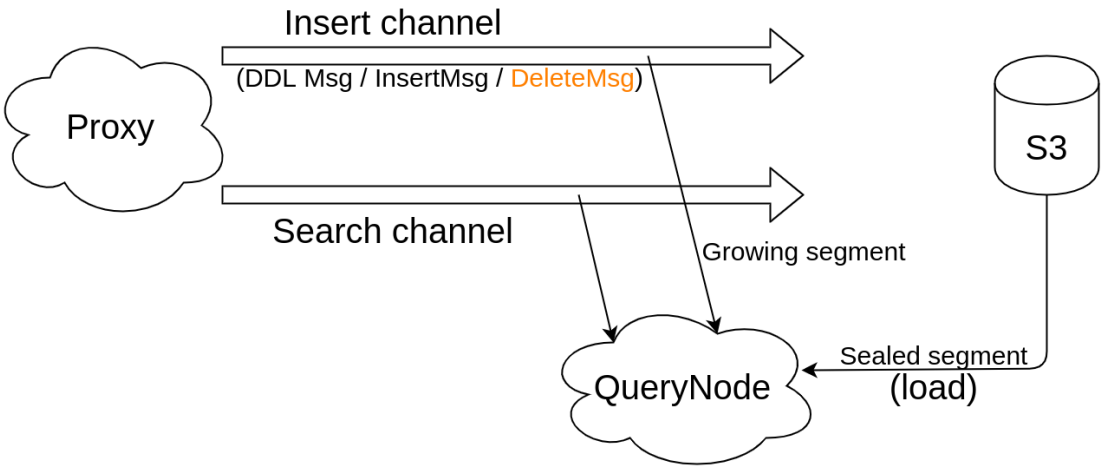
Now the algorithm library `Knowhere` has already supported to search with a bitset which indicates whether an entity is deleted. So we discuss how to store the deleted primary keys here.

Proposal delete operation persistent



1. Proxy receives a delete request
    a. parse `expr` to get primary keys, then split DeleteMsg into insert channels by primary keys (done)
2. DataNode watches insert channel
    a. receive DeleteMsg and persist delete data
        i. when DataNode start up, load all segments info from Minio into memory (done)
        ii. update datanode flowgraph
            1. when DDNode receive DeleteMsg, save it into FlowGraphMsg structure, and send FlowGraphMsg to next node InsertBufferNode (done)
            2. when InsertBufferNode receive FlowGraphMsg, process InsertMsg in it, then wrapper DeleteMsg into another FlowGraphMsg and sent it to next node DeleteNode (done)
            3. when DeleteNode receive FlowGraphMsg, process DeleteMsg in it, save deleted ids and timestamps into delBuf (done)
        iii. update DeleteNode to handle flushMsg
            1. add another flushChan for DeleteNode (done)
            2. when DeleteNode receives flushMsg, save all data in delBuf into MinIO
                a. deleted ids and timestamps are saved into two separated Minio file with name "/by-dev/deltalog/collectionID /partitionID/segmentID/xxx" (done)
    b. DataNode forwards ttMsg and DeleteMsg to delta channel (doing)

Proposal delete operation serving search(sealed+growing)



1. QueryNode subscribe the insert channel
2. QueryNode load the checkpoint and recovery by the checkpoint
3. Proxy receives a delete request, split into insert channels by primary keys
4. QueryNode retrieves a delete request from the insert channel, judges the segment to which each deletion belongs, and updates the Inverted Delta Logs(IDL)
5. ...
6. QueryNode retrieves a search request, search on each segment
7. finish

Proposal delete operation serving search(sealed only)

1. QueryNode subscribe the delta channel
2. QueryNode load the checkpoint and recovery by the checkpoint
3. Proxy receives a delete request, split into insert channels
4. DataNode filter out all delete requests, and write them into the delta channel
5. QueryNode retrieves delete requests from the delta channel, judges the segment to which each deletion belongs, and update the Inverted Delta Logs(IDL)
6. ...
7. QueryNode retrieve a search request, search on each segment
8. finish

Process delete operation in system recovery

Unaffected

SegmentFilter

SegmentFilter provide a method that can used to get the segment id which a PK possible existed in. Implement by segments statistics and bloomfilter.

DeltaLog

DeltaLog is the persistent file, recording the primary keys deleted and the delete timestamp. Each DeltaLog only belongs to a segment.

InvertedDeltaLog

InvertedDeltaLog provides a method that can be used to get deletion that meets the timestamp condition fastly.

Test Plan

Testcase1

Search a deleted entity, except not in the resultset

```
client.insert()
client.search()
client.delete()
client.search()
```

Rejected Alternatives[WIP]

1. Without any channel changes, the delete requests will be forward into the insert channels by Proxy
    a. Since the balance policy of QueryNode, some QueryNodes will only serve the sealed segments. They should discard all insert data from this channel. It cause a lot of performance waste.
2. Add a delta channel for all shards, and all the delete requests will be forward into this delta channel by Proxy
    a. DataNode should subscribe insert channel and delta channel, each time to consume data need to align timetick between these two channel, complicatedly
3. Add a delta channel for each shards, and the delete requests will be forward into this channel and the origin insert channel concurrently. DataNode subscribe the insert channel only, and the QueryNode(serving sealed segments) subscribe the delta channel
    a. It's difficult to sync the insert channel and delta channel

References