MEP 32 -- Atomic switching for collection alias

Current state: Disccusion completed

ISSUE: https://github.com/milvus-io/milvus/issues/17131

PRs: TODO

Keywords: Collection Alias, Atomic Switching

Released: TODO

Summary

1. As the name indicates, CollectionAlias is an alias to an existing collection.
2. The collection alias can be updated to a new collection.
3. Within `RootCoordinator`, `Proxy`, and all the key components, `CollectionName` and `CollectionAlias` are equal.
   e.g. `MetaTable.GetCollectionByName(collectionName string, ts typeutil.Timestamp)` can receive `CollectionAlias` and return corresponding `CollectionInfo.`
4. `CollectionAlias CollectionName = .` `CollectionAlias` cannot collide with existing `CollectionNames`.
5. When alias are added, droped or altered, all proxy should receive the change in the same timestamp.
6. When alias are added, droped or altered, the operations online should not be affected, i.e. they still use the alias information when the tasks begin to excute.

Motivation

1. As `CollectionAlias` works as an extra pointer to the existing collection in the `RootCoordinator`, we can implement collection hot reloading at a low cost.
2. When `AddAlias`, `DropAlias` and `AlterAlias`, the change of relationship between alias and collection should not affect the operations online, i.e. these operations shoud use the relationship when the task begin to excute.
3. When `AddAlias`, `DropAlias` and `AlterAlias`, all proxy should receive the change in the same timestamp to ensure the atomicity between the proxy cache and the rootcoord data.
4. Users can't drop the `collection` if the `collection` is referenced by an `alias`, which can simplify the operation and make it easier to ensure the atomicity property about alias changes.

Design Details

Changes to the MetaTable

- `Metatable` now add `ts2alias2name` and `newestAliasTs` member variables

```
type immutablemap_string2string struct{
    storemap    map[string]string
}

func (imstr2str * immutablemap_string2string) get(key string) (string, error){
    rstr,ok := imstr2str.storemap[key]
    if(!ok){
        return "", fmt.Errorf("key not exist")
    }
    return rstr, nil
}

func (imstr2str * immutablemap_string2string) put(key string, val string) (string, error){
    return "", fmt.Errorf("not allowed put in immutablemap")
}

type metatable struct{
    ...
    ts2alias2name map[Timestamp]immutablemap_string2string
    newestAliasTs Timestamp
    ...
}
```

- Change to alias will add a new `alias2name` immutable map in `ts2alias2name`.

```
func build(im typeutil.ImmutablemapString2string) *Builder {
        builder := &Builder{
```

```
                reference: im,
                mutable:   nil,
        }
        return builder
}

type Builder struct {
        reference typeutil.ImmutablemapString2string
        mutable   map[string]string
}

func (bdr *Builder) maybeClone() {
        if !bdr.reference.IsEmpty() {
                bdr.mutable = make(map[string]string)
                for k, v := range bdr.reference.GetCopy() {
                        bdr.mutable[k] = v
                }

                bdr.reference = typeutil.ImmutablemapString2string{}
        } else {
                bdr.mutable = make(map[string]string)
        }
}

func (bdr *Builder) getnamefromalias(key string) (string, bool) {
        bdr.maybeClone()
        rstr, rbool := bdr.mutable[key]
        return rstr, rbool
}

func (bdr *Builder) putalias2name(key string, val string) (string, bool) {
        bdr.maybeClone()
        var pre string
        v, ok := bdr.mutable[key]
        if ok {
                pre = v
        } else {
                pre = ""
        }
        bdr.mutable[key] = val
        return pre, ok
}

func (bdr *Builder) removealias2name(key string) (string, bool) {
        bdr.maybeClone()
        var pre string
        v, ok := bdr.mutable[key]
        if ok {
                pre = v
                delete(bdr.mutable, key)
        } else {
                pre = ""
        }
        return pre, ok
}

func (bdr *Builder) Build() typeutil.ImmutablemapString2string {
        if !bdr.reference.IsEmpty() {
                reference := bdr.reference
                bdr.reference = typeutil.ImmutablemapString2string{}
                return reference
        }

        mutable := bdr.mutable
        bdr.mutable = nil
        res := typeutil.NewImmutablemapString2string(mutable)
        return res

}
```

```
func (mt * metatable) addAlias(collectionAlias string, collectionName string) error{
    mt.ddLock.Lock()
    defer mt.ddLock.Unlock()
    ...
    ts = getTimestamp()
    tspre = mt.newestAliasTs
    Bdr = build(mt.ts2alias2name[tspre])
    _,ok := Bdr.putalias2name(collectionAlias, collectionName)
    if ok{
        return fmt.Errorf("alias already exist when add alias")
    }
    mt.ts2alias2name[ts] = Bdr.build()
    mt.newestAliasTs = ts
    ...
    return nil
}

func (mt * metatable) dropAlias(collectionAlias string) error{
    mt.ddLock.Lock()
    defer mt.ddLock.Unlock()
    ...
    ts = getTimestamp()
    tspre = mt.newestAliasTs
    Bdr = build(mt.ts2alias2name[tspre])
    _, ok := Bdr.removealias2id(collectionAlias)
    if !ok{
        return fmt.Errorf("alias not exist when drop alias")
    }
    mt.ts2alias2name[ts] = Bdr.build()
    mt.newestAliasTs = ts
    ...
    return nil
}

func (mt * metatable) alterAlias(collectionAlias string, collectionName string) error{
    mt.ddLock.Lock()
    defer mt.ddLock.Unlock()
    ...
    ts = getTimestamp()
    tspre = mt.newestAliasTs
    Bdr = build(mt.ts2alias2name[tspre])
    _,ok := Bdr.putalias2name(collectionAlias, collectionName)
    if !ok{
        return fmt.Errorf("alias not exist when alter alias")
    }
    mt.ts2alias2name[ts] = Bdr.build()
    mt.newestAliasTs = ts
    ...
    return nil
}
```

- `GetCollectionByName(), IsAlias(), ListAlias() ...` which is related to alias will add alias timestamp as parameter to select which `alias2name` map will be used in `ts2alias2name`.
- Users can't drop the `collection` if the `collection` is referenced by an `alias`

Change to globalMetaCache

- Add a function `fetchAliasCache()` to fetch the `newestAliasTs` and the corresponding `alias2name` from rootcoord into proxy if `newestAliasTs` is different with the stored timestamp in `globalMetaCache.`
- When each proxy task begin to excute, running `fetchAliasCache()` to get the `newestAliasTs` and the corresponding `alias2name` from globalMetaCache to task environment.
- When proxy task need a collecition name in task request, first check if the name is alias. If yes, replace it with the corresponding collecition name use gotten `alias2name.`
- If we do this, the cache in `globalMetaCache` will be the map with key as collection real name and value as collection information.
- When proxy task refers to the rootcoord task, it's necessary to pass the cached `newestAliasTs` as the rootcoord task request in order to make the apis which is related with alias to select which `alias2name` they need to use.

Compatibility, Deprecation, and Migration Plan

Above changing plan will add a item in rootcoord task request, which may need some attention.

Test Plan

- Unit tests

References

- ElasticSearch also use immutablemap https://github.com/elastic/elasticsearch/blob/2e992fd4eb1e262a63a57b780400f3de4811d1e5/server/src/main/java/org/elasticsearch/common/collect/ImmutableOpenMap.java