MEP 22 -- Support make storage engine configurable

Summary

Support making storage engine configurable.

Motivation

We found that the standalone version needs to rely on minio as a third-party component. In fact, the standalone version does not need to rely on minio, it can use local storage as a storage engine. If you use local storage, you can also get the following benefits:

1. More efficient reading and writing of persistent data for it don't need network communication.
2. Reduce the impact of its dependency component. For example, when minio crashes, the system is unavailable.
3. Dependency injection can be done with local storage, removing the dependency of minio in unittest.

At the same time, in order to preserve the selectivity of users, minio can also be used as a storage engine for the standalone version. We are going to make the storage engine configurable.

Problems

If we need to achieve the above goals, we will encounter the following problems:

1. It is best not to change the existing startup logic and related interfaces too much.
2. The storage engine should be configurable through milvus.yaml, it means that users can also choose to use minio or others as storage when using standalone.
3. Minio is now used as kv storage interface in milvus, but minio is an object storage engine. This will cause problems. For example, for the current kv interface Load(key string) (string, error), the implementation of minio will convert the obtained []byte to string. For go, converting []byte to string will make a copy of the data in memory. This is a disaster for performance and memory.

Configurable storage engine design

1. For solving Problem 3, We redefine the interface implemented by minio. All file storage needs to implement this interface.

   ChunkManager interface

   ```
   type ChunkManager interface {
           GetPath(filePath string) (string, error)
           GetSize(filePath string) (int64, error)
           Write(filePath string, content []byte) error
           MultiWrite(contents map[string][]byte) error
           Exist(filePath string) bool
           Read(filePath string) ([]byte, error)
           MultiRead(filePaths []string) ([][]byte, error)
           ReadWithPrefix(prefix string) ([]string, [][]byte, error)
           ReadAt(filePath string, off int64, length int64) (p []byte, err error)
           Remove(filePath string) error
           MultiRemove(filePaths []string) error
           RemoveWithPrefix(prefix string) error
   }
   ```

   For this interface we will have three implementations, LocalChunkManager, MinioChunkmanager and VectorChunkManager. VectorChunkManager is an optimized management class for vector reading under distributed milvus. It will use minio as a storagechunkManager and local file system as a vectorChunkStorage. When reading a file, it will be downloaded from the minio to the local, and then the relevant data will be read from the local. In the standalone version, we will replace minioChunkManager with localChunkManager as the implementation of storageChunkManager.
2. ChunkManagerFactory
   For Problem 1, a chunkManagerFactory similar to msgStream. Factory is added to generate chunkManagers with different configurations.

ChunkManager Factory

```
type ChunkManagerFactory struct {
        ChunkStorage string
        VectorCacheStorage string
}
func NewChunkManagerFactory(ChunkStorage,VectorCacheStorage string) *ChunkManagerFactory{}
func (cmf *ChunkManagerFactory) NewChunkStorage(opts ...storage.Options){
        switch (cmf.ChunkStorage){
        case "s3":
        case "minio":
        ...
        }
}
func (cmf *ChunkManagerFactory) NewVectorCacheManager(opts ...storage.Options){}
```

Options is needed when generating a new chunkManager. The Options maybe like this.

ChunkManager Config

```
type config struct {
        address string
        bucketName string
        accessKeyID string
        secretAccessKeyID string
        useSSL bool
        createBucket bool
        rootPath string
}

type Option func(*config)

func Address(addr string) Option {
        return func(c *config) {
        c.address = addr
        }
}
```

This structure will have some redundancy. For example, local storage will not require parameters such as address and bucketname. but will be easier to reuse.

3. Use a more generic factory instead of the existing msgFactory to build nodes.

Interface extened

```
factory := newMsgFactory(localMsg)
rc, err := components.NewRootCoord(ctx, factory)

factory := newFactory(localMsg)
rc, err := components.NewRootCoord(ctx, factory)
```

And the factory will be like this

Factory Struct

```
type Factory struct {
        msgF msgstream.Factory
        storageF storage.ChunkManagerFactory
}

func newDefaultFactory(opts ...Option) *Factory {
        c := newDefaultConfig()
        for opt := range opts {
                opt(c)
        }
        return &Factory{
                MsgFactory: msgstream.NewFactory(c.msgstream),
                storageF: storage.NewChunkManagerFactory(c.vectorCacheStorage, c.chunkStorage),
        }
}

type config struct {
        msgStream string
        vectorCacheStorage string
        chunkStorage string
}
func newDefaultConfig() *config{}

type Option func(*config)

func vectorCacheStorage(vectorCacheStorage string) Option {
        return func(c *config) {
                c.vectorCacheStorage = vectorCachestorage
        }
}
```

4. deploy in milvus.yaml will set the default storage engine for different deploy mode. The vectorCacheStorage and chunkStorage will be used to set what to use as the storage engine.

milvus.yaml

```
deploy:
        standalone:
                chunkStorage: "local"
                vectorCacheStorage: "local"
        distributed:
                chunkStorage: "minio"
                vectorCacheStorage: "local"
```

In standalone mode, chunkStorage using local is a more efficient choice. Of course you can also choose minio.
However, if the distributed version chooses local storage, because the nodes are located on different machines, the data stored locally is inconsistent. Therefore, careful consideration should be given when choosing a storage engine.