ECS-CockroachDB

Final Presentation

CockroachDB

CockroachDB is an open-source distributed SQL database developed by Cockroach Labs. The source code can be found here. At a high level, the CockroachDB setup consists of a cluster of nodes each running its own local instance of a key-value store called RocksDB. RocksDB was developed by Facebook as a successor to Google's LevelDB. CockroachDB uses the Raft consensus algorithm to manage replicas of Ranges, which are discrete chunks of the sorted key space. The CockroachDB client communicates with the cluster via a built-in SQL shell. Note that, since the storage layer consists of local instances of RocksDB, the Raft protocol and other distributed concepts operate at a higher level – i.e., at the level of the Cockroach logic. A detailed design doc can be found in the Github repo linked above.

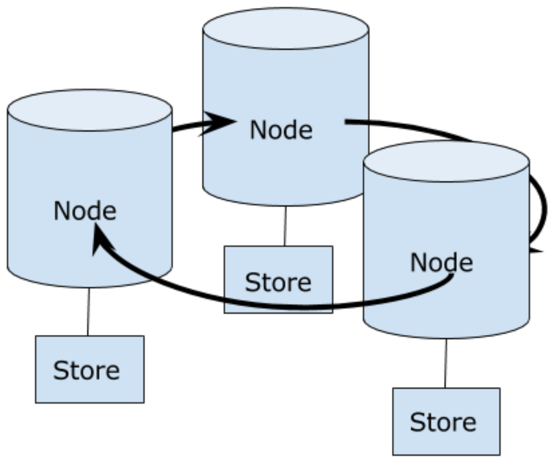
High-Level Goal

ECS-CockroachDB is an implementation of CockroachDB that uses ECS as a backing store for CockroachDB. The purpose of doing this is to show that ECS can support a variety of data services on top of its storage engine – in this case, a distributed relational database like CockroachDB. At a high level, implementing CockroachDB on top of ECS entails the following:

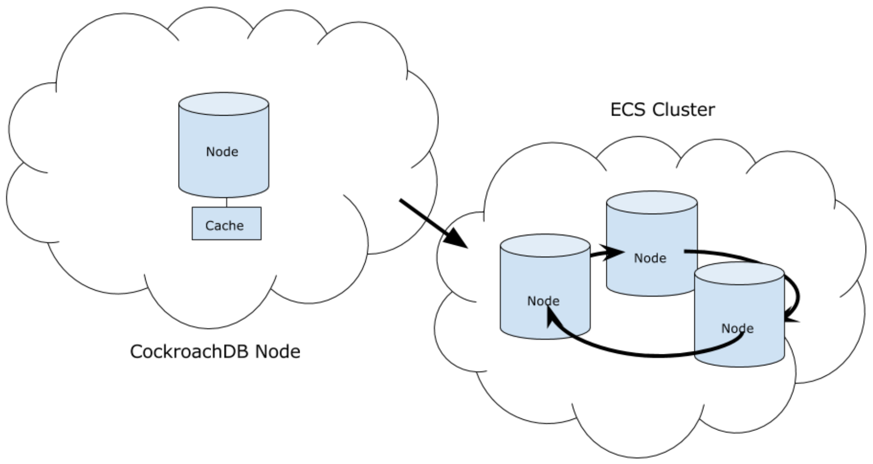
1. Replacing the underlying storage layer of CockroachDB (i.e., RocksDB) with an ECS cluster.
2. Removing CockroachDB's Raft-based replication model so that all replication takes place at the storage layer.
3. Optimizing communication with ECS in order to use ECS-CockroachDB in a practical context.

Work Done

This project involves going from the original CockroachDB setup:



To this:



The primary operations we use to communicate with the ECS cluster are PUT and GET, which are implemented in ecs.go.

In the CockroachDB source code, the file that is responsible for communication with RocksDB is rocksdb.go. This file interfaces with the RocksDB instance through the C++ API imported from the RocksDB library. We have rewritten the methods and functions in rocksdb.go to communicate with ECS directly.

Source Code

Source code can be found at our forked git repository https://github.com/arjun4084346/cockroach/

Current Status

Currently we use the Amazon S3 API to communicate with the ECS cluster via HTTP requests. We follow the original control flow of CockroachDB with RocksDB, and communicate with the ECS cluster instead of communicating with the local RocksDB instance for storage. We cache data on the CockroachDB node in order to improve performance. The ECS-CockroachDB client works the same way as the original CockroachDB client, with a command line SQL interface for database operations.

Get Started

Download the code

```
go get -d github.com/arjun4084346/cockroach
```

replace $GOPATH/src/github.com/cockroachdb/* with cd $GOPATH/src/github.com/arjun4084346/*

set upstream with command

```
git remote add upstream https://github.com/cockroachdb/cockroach
```

One can see the diff between original cockroachdb and forked cockroachdb with command

```
git diff HEAD..origin/master
```

Download Amazon S3 library for Go

```
go get -u github.com/aws/aws-sdk-go
```

Update ecs.go with your ECS data node details

Compile the code with

```
1. cd $GOPATH/src/github.com/cockroachdb/cockroach
2. make build
```

Follow https://github.com/cockroachdb/cockroach/blob/master/CONTRIBUTING.md to understand the process in detail.
If you get “too many open files” error, you can increase open files limit with "ulimit -n 10480".

Please note that network at EMC can interfering in downloading during the build process.

Any online go tutorial is good to learn Go. https://play.golang.org/ can be used for quick practice.

To learn using amazon s3 go library, one can refer http://docs.aws.amazon.com/sdk-for-go/latest/v1/developerguide/sdkforgo-dg.pdf

To learn more about CockroachDB, please watch tech talks available on youtube, read blogs available on official website or ask for support at https://gitter.im/cockroachdb/cockroach

Command "cockroach debug keys" is particularly helpful in debugging.

Code changes

I have followed the convention of writing related functions in the same file. Following files/functions have been added/modified

1. storage/engine/iter.go
2. storage/engine/ecs.go
3. storage/engine/rocksdb.go
   a) modified rocksDBIterator struct

   b) modified functions init, Seek, SeekReverse, Next, Prev, NextKey, PrevKey, Key, Value, unsafeKey, unsafeValue, dbClear, setState

   c) added functions getECSKey, getECSValue, setECSState, goToECSKey, ecsToGoKey, putTimestamp, getTimestamp
4. storage/engine/mvcc.go
5. storage/engine/batch.go
   a) modified functions encodeKeyValue, Clear
6. util/hlc/timestamp.go

   a) added functions EffectiveLess, Minus
7. myMake.sh

Limitation

RocksDB->ECS change could not be tested on all the KV data. The reason being there is so much data and continuous queries to the data that it is increasing latency so much that SQL queries are getting timed out. KV_MAP is used to solve this problem, but it is not a stable caching mechanism, and still unable to handle all the data. Thus, we are not handling some system tables. Function qualifiedKey() controls which key to be considered and which to be ignored. Tables we are skipping are : lease, UI, eventlog, rangelog, system, meta and local. All user data and namespace, descriptor tables are being stored in ECS.

Technical Details

Replacing storage:

Functions getObject() and createObject() in ecs.go perform the read/write from/to ECS.

Replacing RocksDB Iterator:

As we aim to replace data being stored in RocksDB with data being stored in ECS, we also needed to change rocksDBIterator to search for data in ECS instead of searching it in RocksDB.
Any newly instantiated iterator has to first 'Seek' to some KV pair, and then it can call various other functions like, Next, Prev, Key, Value. Because of limitation 1, we restrict our code to execute only on a subset of keys. qualifiedKey() does that, and restrict code in Next(), NextKey(), Prev(), PrevKey(), Seek(), SeekReverse() for such keys. If a key qualifies, their ECS equivalent functions ECSIterNext(), ECSIterPrev(), ECSIterSeek() and ECSIterSeekReverse() are called. Difference in Next(), NextKey() and Prev(), PrevNext() are handled with a bool "skip_current_key_versions". <u>Read storage/engine/engine.go Iterator interface for more detail.</u>
Once the iterator has Seek'ed to some KV pair, we can call it's Key(), Value() functions to fetch Key and Value. Again, due to limitation 1, not all the iterators are correctly Seek'ed and cannot be used further; function qualifiedIter()is used to restrict further ECS operations of fetching Key, Value using functions Key(), Value() respectively on such iterators. If an iterator qualifies, their ECS equivalent functions getECSKey() and getECSValue() are called.

CockroachDB Key to ECS Object encoding:

<u>Understanding cockroach/rocks key, value encoding thoroughly is one of the most important prerequisite before changing the source code of cockroach.</u>
<u>One should first study https://www.cockroachlabs.com/blog/sql-in-cockroachdb-mapping-table-data-to-key-value-storage/ to understand how SQL data is converted to a cockroach key-value pair.</u>

If you print a key, it will be printed in a "pretty format", like "/Table/3/1/50/2/1". This is printed using functions MVCCKey.String() and PrettyPrint() written in storage/engine/mvcc.go and keys/printer.go respectively.

However, be informed that a key does not actually look like this. One can get the actual key bytes from its pretty format using functions UglyPrint, but it does not always work.

Key, value pair is encoded with util/encoding package in such a way that keys (without timestamp) are sorted in the same lexicographic order like their pretty format. Functions goToECSKey and ecsToGoKey have been written to change a key (with timestamp) to ECS object name. A point to note here is that though the encoded key part of an MVCCKey follow the lexicographic order, timestamps follow just the reverse of this order; and this need to be taken care of when iterating through the keys. This is implemented in function ECSIterSeek() in iter.go. Following table explain this. First row is without timestamp, which means it is an Intent. Read more about intents at https://www.cockroachlabs.com/blog/how-cockroachdb-distributes-atomic-transactions/

Order on disk	Order Seek should follow
/Table/51/1/1/0/	/Table/51/1/1/0/
/Table/51/1/1/0/2470089384.722797093,0	/Table/51/1/1/0/4470089384.722797093,0
/Table/51/1/1/0/3470089384.722797093,0	/Table/51/1/1/0/3470089384.722797093,0
/Table/51/1/1/0/4470089384.722797093,0	/Table/51/1/1/0/2470089384.722797093,0

Known Issues

1) CockroachDB is still in Beta state; and there are few resources available to understand its and RocksDB code. How RocksDB Iterator actually works, is still not completely understood and thus in a few cases, result of our ECSIterSeek() does not match with rocksDBIterator.Seek(). The mismatch rate has been brought down to around 5%; and now only these two cases are giving different results

1. Seeked Key - /Table/2/1/0/"bank"/3/1
   Rock returns - /Min
   ECS returns - /Table/2/1/0/"system"/3/1/1470464995.388815723,0

   This debugging information is in pretty format (obviously, otherwise we cannot read it 🙂), "/Min" means no key matched, i.e. there is no key after /Table/2/1/0/"bank"/3/1. This happens only when you try to create a database "bank". **It works like it is prefix matching even when "prefix" bool is false**. Read "Reader" interface in engine.go for more details on "prefix".

2. ~~Seeked Key - /Table/2/1/0/"bank"/3/1~~
   ~~Rock returns - /Table/2/1/0/"bank"/3/1~~
   ~~ECS returns - /Table/2/1/0/"bank"/3/1/1470465015.492820888,0, 13, 1470465015492820888, 0~~
   ~~Role of intent is not clear during Seek operation; for the same Seek query, completely opposite result has also been observed.~~

   ~~Seeked Key - /Table/3/1/50/2/1~~

   ~~Rock returns - /Table/3/1/50/2/1/1470459450.260533982,0~~

   ~~ECS returns - /Table/3/1/50/2/1~~

   ~~So we need to understand the order of intents in the sorted Key map. "debug keys" does not even show that they exist, but they are returned. Sometime key with timestamp is returned, sometime key without timestamp (intent) is returned.~~

   This is now resolved and code pushed to master. Problem was with intents. Intents are keys without timestamp. Function "mvccResolveWriteIntent" in mvcc.go clears the intent. I have modified function rocksDBBatchBuilder.Clear in batch.go to clear the intent from ECS too. This solved all the problem.

2) During the communication with ECS, if there are more items to return in the response object, we need to check the value of "ListObjectsOutput.IsTruncated" field and correspondingly query again with "ListObjectsInput.Marker" being equals to "ListObjectsOutput.NextMarker" of previous response. It is not implemented yet.

Known Enhancements

1. Remove replication on the Cockroach side and rely on ECS for all replication.
2. Change the way Cockroach communicates with ECS in order to enhance performance:
   a. We want to extend the Amazon S3 API that we currently use with ECS's extended key-value REST API in order to communicate directly with the ECS Object Table.
   b. Another option is to link the ECS blob client directly to CockroachDB in order to bypass the intermediate head server in ECS.