Leases on Memcached

Introduction

The goal of this project is to design and implement a programming model that employs Shared(S), Inhibit(I) and eXclusive(X) leases to prevent race conditions that may insert stale data in the Key Value Server. This model provides strong consistency in Cache Augmented Data Stores (RDBMS). We have implemented leases for the latest versions of Memcached Server(v1.5.12) and Whalin client (v3.0.2). The implementation is focused on Write Through policy. We have extended the IQ-Framework [1] to enable the concept of sessions consisting of multiple key-value pairs. The report is organized as follows. The first section explains the key terms used throughout this document. In the second section, we define the architecture followed by details of its implementation in the third section. We present the results in section 4 followed by discussion and future work in the subsequent sections.

Contributions

I did this project with Shashwat and following is the list of my contributions.

1. Planned the project along with Shashwat.
2. Wrote the project proposal.
3. Together with Shashwat, worked on designing the server architecture and writing pseudo codes for new server commands.
4. Implemented and tested new methods in the latest version of Whalin Client.
5. Implemented Session coordinator.
6. Worked on building a new YCSB JDBC client to execute benchmarking.
7. Set up client, Memcached server, coordinator and MySQL database on different Azure instances to run benchmarks.
8. Analyzed the results.
9. Lines of code written: Whalin client ~1500, Coordinator ~250, YCSB client ~500

Terms and definition

Session

A session is an operation consisting of one database transaction and multiple read or/and write operations on multiple key-value pairs atomically. One application thread can be imagined as a session. A session may be a read or a write. A read session does not update any of its key-value pairs. If it observes a cache miss, it computes a key-value pair and sets it in the cache. A write session may perform updates on one or more cache entries. A session may acquire leases on one or more keys. A session may hold one or multiple leases on one or multiple keys during its lifetime. Once it commits or aborts, all leases are deleted. Each session has a unique session id.

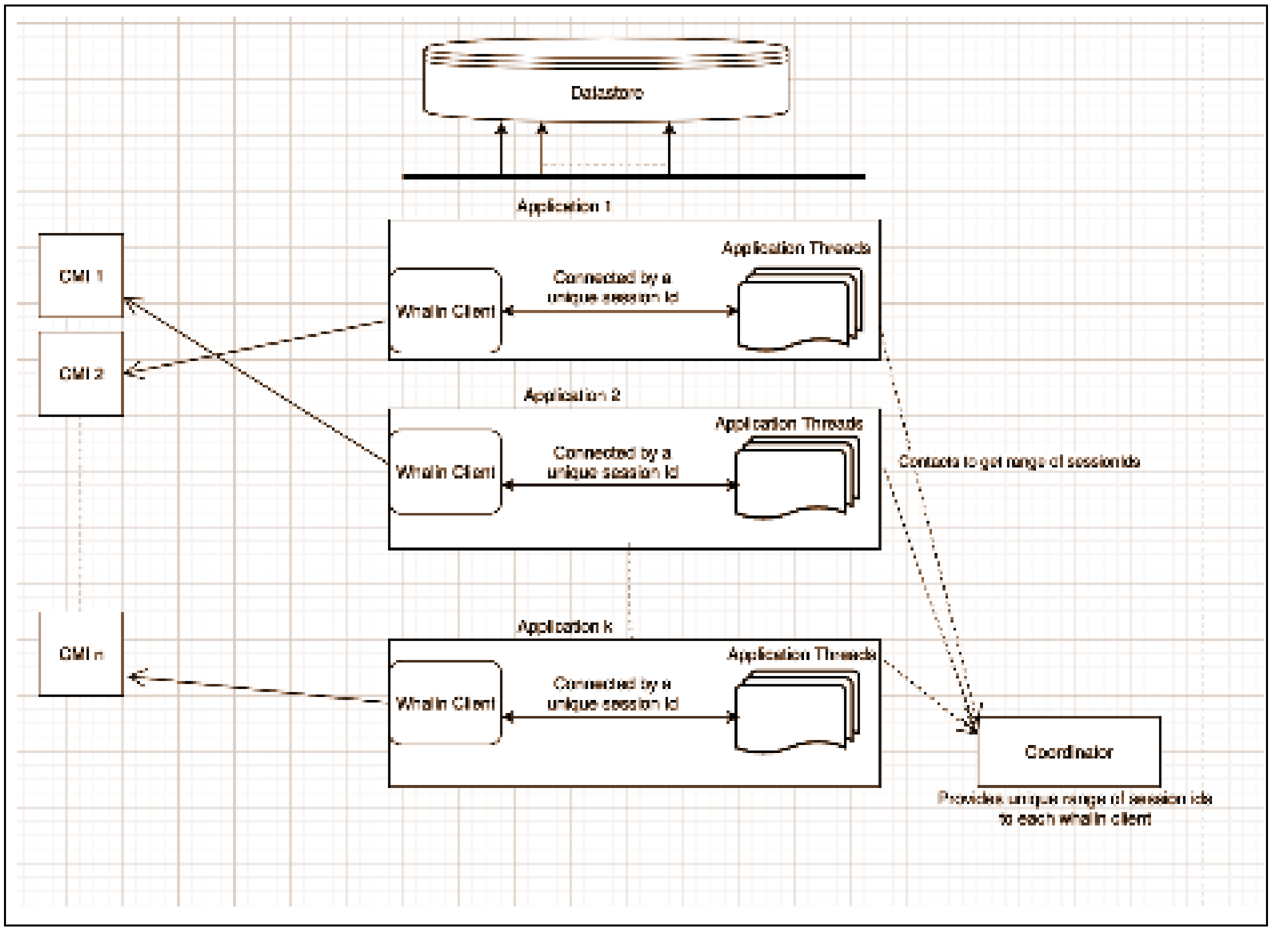
Life Cycle of a Session

1. Active: An application asks for a new session id from the Whalin client and uses the same id until the session ends. A session actively communicating with the cache server until it is either committed or aborted.
2. Aborted: The session’s state changes to aborted when either the client explicitly calls *labort* on a session or when an operation can not be performed by the server.
3. Validated: The session state changes to validated when a client validates the session successfully. Once a session is validated, it becomes golden and no other session can abort it.
4. Committed: The session is committed once all relevant transactions are completed and the session id is released and made available for other application threads.

Lease

A lease is similar to locks in database systems with a difference that a lease may expire. Leases serialize concurrent sessions, preventing undesirable race conditions to put stale data in the cache. A lease may be a Shared, Inhibit or eXclusive and is granted at the granularity of a key.

Architecture


Datastore
Application 1
Application Threads
Connected by a unique session id
Whalin Client
CMI 1
CMI 2
Application 2
Application Threads
Connected by a unique session id
Whalin Client
Contacts to get range of session ids
Application k
Application Threads
Connected by a unique session id
Whalin Client
CMI n
Coordinator
Provides unique range of session ids to each whalin client

In our design, there are 4 major components: Session Coordinator, Cache Server, Client, and the data store.

Application

The application is the component which communicates to the cache server using the client. It is the responsibility of the application to handle the exceptions or errors correctly.

Session Coordinator

A Session coordinator is responsible to assign ranges of session ids to clients. When the client is initialized, it gets a range of session ids from the coordinator. When a session starts, an available session id is assigned to an application thread. Once the range is exhausted, the client sends a request to the coordinator for a new range. The coordinator maintains the mappings of clients and their assigned session ids.

Client

The client acts as an mediator between an application and cache server. Also, it owns the responsibility of contacting the session coordinator to get a range of sessions and assign them to application threads as and when requested. It is mandatory for each of the client requests to pass the session id to the server for each lease operation it performs.

Cache Server

The cache server maintains session information for each session during its lifetime. This information is stored as key-value pairs pinned in memory. The information includes the session id, the session's status, the leases acquired by the session and a mapping of keys involved in the session and their pending versions (if exist). A session status may be one of these values: active, validated or aborted. When a client initializes a session with the cache server, the cache server creates an item to store the session, setting its state to active. The state changes to validated when a client validates the session successfully. Once validated, the server no longer may abort the session. The session commits when the client sends lcommit command and the server acknowledges. All session related information is deleted at that time.

Following is the list of key-value pairs pinned in memory for a session.

To store all leases for a key
lease_{lease_type}_{key}: {session_{sess_id}, session_{sess_id}, session_{sess_id}…...}
eg :- 1. [lease_X_K1: session_S1]
2. [lease_S_K2: session_S2 session_S3 session_S4]
3. [lease_I_K3: session_S2]

To store pending versions for a key and session
These versions are moved to the original hashtable once, the session is committed. If a session is aborted, all these versions are deleted.
{session_id}_{key} : Pending versions
eg:- s1_K1: v1, s1_K2:v2

To maintain session's status and leases acquired

session_{session_id} : { status, lease_{Lease_Type}_{Key}....} // First 8 bits represent lease
eg:- session_s1: {V lease_I_K1 lease_S_K2 lease_X_K3}

Datastore

Datastore can be any standard database which stores the data and which application can use to get data and populate in cache server if required.

Implementation

Memcached Whalin Client

We have extended the latest version of WhalinClient to implement methods that support new methods for leases. Each application thread maintains an instance of the client to manipulate sessions generated by the thread. Many of the introduced methods might throw *IncompatibleLeaseTypeException* which is a custom exception created to notify application to handle the error when the cache server is unable to provide a specific lease on a key. Any application thread which gets this exception should abort itself by rolling back the session executed so far.

Following are a set of important new methods with their details.

Method Signature	**Description**
beginSession()	Assigns a unique id to a session. If no session id is available, this method internally requests the session coordinator for another range of unique session ids.
endSession(sessionId)	Releases a session id and makes it available for other sessions.
disableBackoff()	Disables the back-off which lets client abort itself when an Inhibit lease is not available for a key.
enableBackOff()	Enables the back-off which lets client retry and waits for the lease to be available.
lget(key,hashCode, asString,sess_id, exclusive)	This method is used to retrieve the value for a key. The exclusive flag is set to false or true if the context of the call is read-only or read-modify-write, respectively. If the session is aborted then, the server returns ABORT. If the key exists and has no existing leases or does not have an exclusive lease assigned to other sessions, the server

	returns the value of the key. If the key does not exist and no other session is granted an Inhibit lease on it, the server assigns Inhibit lease to this session and returns LEASE as a response.
Iset(key,value, hashCode,sess_id)	Stores data on the server. If the session object exists and its state is aborted, the server deletes it and returns ABORT. Otherwise: 1. If the key has X lease but it does not belong to the session, the server returns ABORT. 2. If the session has a lease belonging to the session. The server puts (key, val) in the cache, upgrade lease to X(if required) and returns SUCCESS. 3. If the key has an X or no lease, the server puts (key, val) in the cache,and returns SUCCESS.
Ivalidate(sid)	This method validates all the leases acquired by a specific session prior to committing the data store transaction by calling server's Ivalidate method. A session can be invalid if it does not hold a needed lease. A lease can be invalid if it is either acquired by a different session or it expired. Once a session is validated, it becomes Golden and no other session can abort it. The server may return SUCCESS when all leases are still valid, or ABORT if the server aborted the session or there is a lease expire. A session must be validated before committing it to the database.
Icommit(sid)	This method commits a session which is already validated by Ivalidate If the sessions state is aborted, the server deletes it and returns ABORT. This may happen when the client issues Icommit without a Ivalidate method. Otherwise, the server releases all leases hold by this session. With each key, if its pending version exists, it is swapped with the original version. If its pending version is empty, the server deletes the original version.If its pending version does not exist, the server keeps the original version. Finally, the server returns SUCCESS.

Memcached Server

We introduce new methods at the cache server in the context of sessions. Most methods can be mapped one to one to the new client methods. With a session, the client communicates with the server only by using the session id. We have implemented the server based on write-through policy. Below, we present pseudocodes for a few important ones.

```
Lget(key, sessionID){
   if(checkIfAborted(sessionID)){
      LAbort(sessionID, true)
```

```
        return ABORT
    }
    value = get(sessionID + "_" + key)
    if(value!=null){
        //Already modified this key, return modified value
        return value
    }

    //check X/I/S lease
    retreivedSessionID = get("lease_X_" + key)
    if(retreivedSessionID != null && retreivedSessionID != sessionID){
        LAbort(sessionID, true);
        return ABORT
    }
    retreivedSessionID = get("lease_I_" + key)
    if(retreivedSessionID != null && retreivedSessionID != sessionID){
        return BACK_OFF
    }
    //no one has either X or I lease

    val = get(key)
    if(val == null){ //Have to grant S and I lease
        set("lease_I_" + key, sessionID)
        //Granted I lease, now will have to write this to control
        upsertControl("session_"+sessionID, ACTIVE, INHIBIT|key)

        //Granted S lease as well, now will have to write this to control
        upsert("lease_S_" + key, sessionID)
        upsertControl("session_"+sessionID, ACTIVE, SHARED|key)
        return SUCCESS_LEASE

    }else{ //Have to grant S Lease
        upsert("lease_S_" + key, sessionID)
        //Granted S lease, now will have to write this to control
        upsertControl("session_"+sessionID, ACTIVE, SHARED|key)
        // Return success with value
        return SUCCESS + val
    }
}
```

```
Lset(key, value, sessionID){
  if(checkIfAborted(sessionID)){
      Labort(retreivedSessionID,true)
      return ABORT
  }
    //should we check if we are already validated or not?
    //check X/I/S lease
  retreivedSessionID = get("lease_X_" + key)
  if(retreivedSessionID != null && retreivedSessionID != sessionID){
      LAbort(sessionID, true)
      return ABORT
  }
  else if(retreivedSessionID == sessionID) //already X lease available for the
 session
  {
      set(sessionID+"_"+key, value)
      return SUCCESS
   }

  retreivedSessionID = get("lease_I_" + key)

  if(retreivedSessionID != null && retreivedSessionID != sessionID){
      if(!LIsValidated(retreivedSessionID)){
          Labort(retreivedSessionID, false)
      }else{
          Labort(sessionID, true)
          return ABORT
      }
  }else if(retreivedSessionID == sessionID){
      //set value in cache
      set(key, value)
      //release I lease
      delete("lease_I_" + key)
      return SUCCESS
  }
  //WE did not have the I lease on this key
  //Now we have to grant X lease to this session
  retreivedSessionIDList = get("lease_S_" + key)

 //First check if any session is validated and in that case, do not abort any
 of the shared
```

```
   if(LIsAnySessionsValidated(retreivedSessionIDList) {
        Labort(sessionID, true)
        return ABORT
          }
    else{
        for(sess in retreivedSessionIDList){
            Labort(sess, false)
        }
   }

   delete("lease_S_" + key)
   set("lease_X_" + key, sessionID)
   upsertControl("session_"+sessionID, ACTIVE, EXCLUSIVE|key)
   set(sessionID+"_"+key, value)
   return SUCCESS
}
```

```
Lvalidate(sessionID){
       if(checkIfAborted(sessionID)){
           LAbort(sessionID, true)
           return ABORT
       }
       control = get("session_" + sessionID)
       for(lease|key in control[8:]){
           if(not have lease on key){
               Labort(sessionID, true)
               return ABORT
           }
       }
       set("session_" + sessionID, VALIDATED + control[8:])
       return VALIDATED
   }
```

```
Lcommit(sessionID){
      if(checkIfAborted(sessionID)){
          LAbort(sessionID, true)
           return ABORT
```

```
    }
    control = get("session_"+sessionID)
    //check first byte of control here
    for(lease|key in control[8:]){
        if(lease == EXCLUSIVE){
            //get modified value
            modifiedValue = get(sessionID+"_"+key)
            if(modifiedValue == null){
                delete(key)
            }else{
                set(key, modifiedValue)
            }
            //delete modified value
            delete(sessionID+"_"+key)
            delete("lease_X_"+key)
        }else if(lease == SHARED){
            LDeleteFromList(key, get("lease_S_"+key), sessionID)
        }
    }
    delete("session_"+sessionID)
    return SUCCESS

}
```

```
LAbort(sessionID, cleanupSession){
    control = get("session_"+sessionID)
    if((control[:8]==ABORTED) && cleanupSession)
    {
        delete("session_" + sessionID)
        return SUCCESS
    }
    for(lease|key in control[8:]){
        if(lease == EXCLUSIVE){
            delete(sessionID+"_"+key)
            retreivedSessionID = get("lease_X_"+key)
            if(retreivedSessionID == sessionID){
                delete("lease_X_"+key)
            }
        }else if(lease == SHARED){
            LDeleteFromList(get("lease_S_"+key), sessionID)
        }else{
            retreivedSessionID = get("lease_l_"+key)
            if(retreivedSessionID == sessionID){
                delete("lease_l_"+key)
            }
        }
    }
    if(!cleanupSession) {
        set("session_" + sessionID, ABORTED)
    }
    else{
        delete("session_" + sessionID)
    }
    return SUCCESS
}
```

Session Coordinator

The coordinator is implemented as a lightweight Spring boot application. It uses an integer sequence to create a new session id. We have made use of concurrentHashMap to make it thread-safe.

Following is the sample code to get a new range of sessions for a specific client from the coordinator.

```
public static Map<String, Boolean> getSessions(String clientId)
  {
      Map<String, Boolean> map = new ConcurrentHashMap<String, Boolean>();
      try {
          URL url = new URL("http://localhost:8080/getRange/"+clientId);
          HttpURLConnection conn = (HttpURLConnection) url.openConnection();
          conn.setRequestMethod("GET");
          conn.setRequestProperty("Accept", "application/json");
          if (conn.getResponseCode() != 200) {
              throw new RuntimeException("Failed : HTTP error code : "
                      + conn.getResponseCode());
          }
          BufferedReader br = new BufferedReader(new InputStreamReader(
                  (conn.getInputStream())));
          String output= br.readLine();
          String sessionIdList=
           output.substring(output.indexOf("sessionIds")+"sessionIds".length()+3,output.length()-2
          String[] array=sessionIdList.split(",");
          for(String str: array)
          {
              map.put(str,true);
          }
          conn.disconnect();
       } catch (MalformedURLException e) {
          e.printStackTrace();
       } catch (IOException e) {
          e.printStackTrace();
       }
     return map;    }
```

YCSB Benchmarking

For benchmarking, we have forked from CADS-JDBC repository which has ycsb client support for benchmarking IQ Framework. To make it work for our project, we made a few changes which include changes in the ycsb script, adding the changed Whalin client jar, and write required methods to use the new methods introduced in the client.

Results and Discussion

Configuration

We tried to set up each of the database(MySql in our case), Client & Coordinator, and Memcached server in different Azure instances. However, due to some issues and time constraints, we could not run our benchmarking in this setup. Therefore, we present the results by running benchmarks on our local machine(16 GB dram, 256 SSD, 2.8 GHz Intel Core i7) with 20,000 rows stored in datastore and 10,000 operations performed.

Benchmarking Results

The results did not meet our expectations. As we see in the following table, latency for cached augmented datastore with leases is much higher than the one without any cache. The reason might be a combination of the following.

1. All of the applications were competing for limited resources.
2. MySql maintains its own buffer pool to cache the query results leading to the low latency in pure JDBC case.
3. CADS with leases perform operations which acquire and release locks, and this can add to the latency.

However, we can see that the performance for JDBC and JDBC-CADS improves as we increase the number of threads from 1 to 10 and then deteriorates as we increase it further to 100. The reason is the CPU becomes the bottleneck with 100 threads.
Furthermore, for write-heavy workload, the performance of JDBC-CADS becomes worse. The reason could probably be the waiting time for acquiring the exclusive lease and aborting the session and restarting again.

Workload A(Latency in ms)

#threads	**1**	**10**	**100**
JDBC	6135	1994	2187
JDBC-CADS	13849	12877	15219

Workload A(Throughput)

#threads	**1**	**10**	**100**
JDBC	1629.99185	5015.045135	4481.024234
JDBC-CADS	722.0737959	698.920556	657.0733951

Workload C(#threads=1)

	Latency	**Throughput**
JDBC	3179	3145.643284
JDBC-CADS	42916	233.0133284

Related and Future work

This project is an extension to IQ framework and supports a session with multi-key updates. Another difference is that in this project, the leases information is stored in the cache server instead of in the client. As future work, we want to compare our project results with IQ framework's results. Moreover, we want to verify the strong consistency aspect of our project using Polygraph module.

References

[1] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. Middleware, December 2014.