Lab3: Transactional Key-Value Database

518021910184 DengShiyi

Prepreparation

What I have

Before this lab, I have already implemented class `inode_manager` and class `lock_server`. Class `inode_manager` is used to manage inodes, including allocation free operation, reading or writing contents of files, etc. Class `lock_server` is used to process requests of lock acquiring and releasing.

In this lab, I choose class `lock_server` instead of `lock_client_cache` as the lock server.

Hash function

To implement a key-value database, we need to map keys of type string to integer. In this lab, I implement a function used to do the mapping. Here's the code.

```
inline static unsigned int BKDHash(const std::string &str){
    const char *cstr = str.c_str();
    unsigned int hash = 0;
    while(unsigned int ch = (unsigned int)*(cstr++)){
        hash = hash * 131 + ch;
    }
    return hash % 1024;
}
```

The input is a string and output is a integer between 0 and 1023. The function traverses the string and give each character the coefficient powers according to its index in the string. The coefficient here is a constant which is decided before program execution. Then add up the product of all the characters and the power of the coefficient. Finally take the remainder with 1024 (`INODE_NUM`).

Part 1: Simple Database without Transaction

The implementation of this part is in the file `ydb_server.cc`. There are three function to implement. The idea of three functions is similar which is hashing the key and put/get the value to/from extent server. Delete operation is the same as set operation except that former put an empty string while latter put a real value. Here is the code of `ydb_server::get()` as an example.

```
ydb_protocol::status ydb_server::get(ydb_protocol::transaction_id id, const
std::string key, std::string &out_value) {
    unsigned int eid = BKDHash(key);    // hash the key
    ec->get(eid, out_value);        // get value according to hash
    return ydb_protocol::OK;
}
```

Part 2: Two Phase Locking

Design

The idea of two phase lock is add lock before accessing a sharing variable and release lock at commit stage. To optimize performance, I add cache to each transaction so that it can read/write data quickly. Besides, cache logs what key a transaction has accessed. So when aborting or committing, the transaction can release correspond lock.

The method of detecting deadlock is to build a graph and check whether there is a cycle in the graph. There are two kinds of nodes in the graph which are lock nodes and transaction nodes. Both kinds of nodes contain a pointer pointing to next node. A next pointer in lock node can only point to a transaction node and that in a transaction node can only point to a lock node. When a transaction wants to acquire a lock of a specified variable, it will add an edge from its transaction node to the lock node corresponding to the variable. Then the transaction will check whether there is a cycle in the graph. If so, the transaction will abort and delete all the related data structure. Else, the transaction will acquire the lock.

Based on the above method, when a transaction begin, program will create a new transaction node and push it to transaction node set. When a transaction get a value with a key, it will find in the cache first. If cache misses, the transaction will try to edit the graph and check the cycle. If no deadlock, it will get the value through extent server. Get operation is similar. The commit operation needs to flush all the written cache entry of the transaction into extent server and then release the corresponding lock. Abort operation simply deletes the transaction and release corresponding lock.

Especially, all the shared variables including graph should be protected by `LOCK_LOCK`.

Code Explanation

All the explanation is in the code comment.

- **Data structure**
    - **Cache entry**

```
typedef struct cache_entry {
    extent_protocol::extentid_t eid;    // hash result of key
    std::string value;                  // value
    bool write_flag;                    // whether value is changed

    cache_entry(extent_protocol::extentid_t eid, std::string value, bool
write_flag=false){
        this->eid = eid;
        this->value = value;
        this->write_flag = write_flag;
    }
} cache_entry_t;
```

    - **Graph node**

```
typedef struct gnode {
    gntype type;                // transation node or lock node
    ydb_protocol::transaction_id trans_id;  // no this entry in lock
node
    std::vector<cache_entry_t> cache;       // cache set
    gnode *next;                            // next pointer

    gnode(){
        type = LOCKNODE;
        trans_id = -1;
        next = NULL;
    }
} gnode_t;
```

  - **Graph**

```
gnode_t ydb_server_2pl::lock_nodes[1024];
gnode_t ydb_server_2pl::trans_nodes[MAX_TRANS];
```

- `ydb_server_2pl::transaction_begin()`

```
ydb_protocol::status ydb_server_2pl::transaction_begin(int,
ydb_protocol::transaction_id &out_id) {
    lc->acquire(TRANS_COUNT_LOCK);  // lock local var
    out_id = (trans_count++) % MAX_TRANS;   // trans_id increase
progressively
    lc->release(TRANS_COUNT_LOCK);
    put_trans(out_id);          // put transaction node into graph
    printf("transaction begin %d\n", out_id);   // debug info
    return ydb_protocol::OK;
}
```

- `ydb_server_2pl::transaction_commit()`

```
ydb_protocol::status ydb_server_2pl::transaction_commit(
                                        ydb_protocol::transaction_id id, int
&) {
    printf("transaction commit %d\n", id);
    gnode_t *trans = find_trans(id);        // find transaction node
    if(!trans){                             // check invalid transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    lc->acquire(LOCAL_LOCK);                // protect local shared
variables
    int size = trans->cache.size();
    for (int i = 0; i < size; ++i){         // traverse the cache
        extent_protocol::extentid_t eid = trans->cache[i].eid;
        if(trans->cache[i].write_flag){     // write to extent server if
written
            ec->put(eid, trans->cache[i].value);
        }
        lc->release(eid);                   // release lock of key
        del_edge(&lock_nodes[eid]);         // edit the graph
    }
    del_trans(id);                          // delete transaction node from graph
    lc->release(LOCAL_LOCK);
    return ydb_protocol::OK;
}
```

- `ydb_server_2pl::transaction_abort()`

```
ydb_protocol::status ydb_server_2pl::transaction_abort(
                                    ydb_protocol::transaction_id id, int &)
{
    printf("transaction abort %d\n", id);
    gnode_t *trans = find_trans(id);        // find transaction node
    if(!trans){                             // check invalid transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    lc->acquire(LOCAL_LOCK);                // protect local variables
    int size = trans->cache.size();
    for (int i = 0; i < size; ++i){         // traverse cache
        extent_protocol::extentid_t eid = trans->cache[i].eid;
        lc->release(eid);                   // release all lock of key
        del_edge(&lock_nodes[eid]);         // edit the graph
    }
    del_trans(id);                          // delete transaction node from graph
    lc->release(LOCAL_LOCK);
    return ydb_protocol::OK;
}
```

- `ydb_server_2pl::get()`

```
ydb_protocol::status ydb_server_2pl::get(ydb_protocol::transaction_id id,
                               const std::string key, std::string &out_value) {
    printf("T%d: get(%s) begin\n", id, key.c_str());
    extent_protocol::extentid_t eid = BKDHash(key);     // hash the key
    gnode_t *trans = find_trans(id);            // find transaction node
    if(!trans){                                 // check invalid transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    int size = trans->cache.size();
    for (int i = 0; i < size; ++i){             // find eid in cache of trans
        if (trans->cache[i].eid == eid) {       // found
            out_value = trans->cache[i].value;
            printf("T%d: get(%s) = %s\n", id, key.c_str(), out_value.c_str());
            return ydb_protocol::OK;
        }
    }
    lc->acquire(LOCAL_LOCK);                    // first time to read
    add_edge(&lock_nodes[eid], trans);          // edit graph
    bool dead_lock = find_circle(&lock_nodes[eid]);     // check cycle in
graph
    if (dead_lock){                             // deadlock
        for (int i = 0; i < size; ++i) {        // same logic as abort
            extent_protocol::extentid_t eid = trans->cache[i].eid;
            lc->release(eid);
            del_edge(&lock_nodes[eid]);
        }
        del_trans(id);
        lc->release(LOCAL_LOCK);
        return ydb_protocol::ABORT;
    }
    lc->release(LOCAL_LOCK);                    // no deadlock

    lc->acquire(eid);                           // acquire lock of key

    lc->acquire(LOCAL_LOCK);                    // protect local variables
    del_edge(trans);                            // edit graph
    add_edge(trans, &lock_nodes[eid]);
    ec->get(eid, out_value);                    // get value from extent
server
    lc->release(LOCAL_LOCK);
    cache_entry_t c(eid, out_value);            // alloc new cache entry
    trans->cache.push_back(c);
    printf("T%d: get(%s) = %s\n", id, key.c_str(), out_value.c_str());
    return ydb_protocol::OK;
}
```

- `ydb_server_2pl::set()`

```
ydb_protocol::status ydb_server_2pl::set(ydb_protocol::transaction_id id,
                       const std::string key, const std::string value, int
&) {
    printf("T%d: set(%s)=%s begin\n", id, key.c_str(), value.c_str());
    extent_protocol::extentid_t eid = BKDHash(key);     // hash the key
    gnode_t *trans = find_trans(id);            // find reansaction node
    if(!trans){                                 // check invalid transaction
id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    int size = trans->cache.size();
    for (int i = 0; i < size; ++i){             // find eid in cache of
transaction
        if (trans->cache[i].eid == eid) {       // cache hit
            trans->cache[i].value = value;
            trans->cache[i].write_flag = true;  // set the write flag
            printf("T%d: set(%s)=%s end\n", id, key.c_str(), value.c_str());
            return ydb_protocol::OK;
        }
    }
    lc->acquire(LOCAL_LOCK);                    // first time to write
    add_edge(&lock_nodes[eid], trans);          // edit graph
    bool dead_lock = find_circle(&lock_nodes[eid]); // check cycle in graph
    if (dead_lock){                             // deadlock
        for (int i = 0; i < size; ++i) {        // abort
            extent_protocol::extentid_t eid = trans->cache[i].eid;
            lc->release(eid);
            del_edge(&lock_nodes[eid]);
        }
        del_trans(id);
        lc->release(LOCAL_LOCK);
        return ydb_protocol::ABORT;
    }
    lc->release(LOCAL_LOCK);                        // no deadlock

    lc->acquire(eid);                               // acpuire lock of key

    lc->acquire(LOCAL_LOCK);
    del_edge(trans);                                // edit graph
    add_edge(trans, &lock_nodes[eid]);
    lc->release(LOCAL_LOCK);
    cache_entry_t c(eid, value, true);              // alloc new cache entry
    trans->cache.push_back(c);
    printf("T%d: set(%s)=%s end\n", id, key.c_str(), value.c_str());
    return ydb_protocol::OK;
}
```

- `ydb_server_2pl::del()`

  Delete operation is similar to set operation.

Part3: Optimistic Concurrency Control

Design

The main idea of OCC is read/write freely and check consistency when commit. If data in read set is the same as data in extent server, write data in write set to extent server. Else, abort.

Each transaction contains a read set and a write set. When a transaction read a value, it will find in the write set first, then in the read set and then request the value from extent server. Because a transaction should read the data it writes in the same transaction, the priority of read set should be higher of that of write set in get operation. Set operation is simple. We first find the value of the same key in write set and cover the value. If the value is not found in write set, simply alloc a new write set entry and push it to the write set. Commit operation first checks whether value in the read set is the same as the extent server. If so, it will write the value in write set to the extent server. Else, the transaction will abort. Abort operation simply deletes all the data structure related to the transaction.

Code Explanation

- **Data structure**
  - **Cache entry**

```
typedef struct cache_entry_occ {
    extent_protocol::extentid_t eid;    // hash value of key
    std::string value;                  // value

    cache_entry_occ(extent_protocol::extentid_t eid, std::string value){
        this->eid = eid;
        this->value = value;
    }
} cache_entry_occ_t;
```

    - **Transaction entry**

      ```
      typedef struct trans_entry{
          ydb_protocol::transaction_id trans_id;          // tramsaction id
          std::vector<cache_entry_occ_t> read_set;        // read set
          std::vector<cache_entry_occ_t> write_set;       // write set

          trans_entry(){
              trans_id = -1;      // initiate transaction id as an invalid id
          }
      } trans_entry_t;
      ```

    - **Transaction list**

      ```
      trans_entry_t trans_nodes[MAX_TRANS_COUNT];
      ```

- `ydb_server_occ::transaction_begin()`

  ```
  ydb_protocol::status ydb_server_occ::transaction_begin(int,
  ydb_protocol::transaction_id &out_id) {
      lc->acquire(TRANS_COUNT_LOCK);          // protect local variables
      out_id = (trans_count++) % MAX_TRANS_COUNT; // trans_id increase
  progressively
      lc->release(TRANS_COUNT_LOCK);
      put_trans(out_id);                      // push new transaction entry to
  list
      printf("transaction begin %d\n", out_id);
      return ydb_protocol::OK;
  }
  ```

- `ydb_server_occ::transaction_commit()`

  ```
  ydb_protocol::status ydb_server_occ::transaction_commit(
                                  ydb_protocol::transaction_id id, int &) {
      printf("T%d: transaction commit begin\n", id);
      bool abort_flag = false;
      trans_entry_t *trans = find_trans(id);      // find transaction entry in
  list
      if (!trans) {                               // invalid transaction id
          printf("find_trans fault!\n");
          return ydb_protocol::TRANSIDINV;
      }
      lc->acquire(COMMIT_LOCK);                   // protect local variable
      int read_size = trans->read_set.size();
      for (int i = 0; i < read_size; ++i) {       // validation procedure
          cache_entry_occ_t loc_entry = trans->read_set[i];
          std::string remote_val;
          ec->get(loc_entry.eid, remote_val);     // get value from extent
  server
          if(remote_val != loc_entry.value){      // if inconsisitent, abort
              abort_flag = true;
              break;
          }
      }
  ```

```
    if (!abort_flag){                           // validation success
        int write_size = trans->write_set.size();
        for (int i = 0; i < write_size; ++i) {
            cache_entry_occ_t loc_entry = trans->write_set[i];
            ec->put(loc_entry.eid, loc_entry.value);// store value to extent server
        }
        del_trans(id);                  // reset the transaction entry
        lc->release(COMMIT_LOCK);
        printf("T%d: transaction commit\n", id);
        return ydb_protocol::OK;
    }
    else{                               // abort
        del_trans(id);                  // simply reset the transaction
        lc->release(COMMIT_LOCK);
        printf("T%d: transaction ABORT ABNORMALLY\n", id);
        return ydb_protocol::ABORT;
    }
    return ydb_protocol::OK;
}
```

- `ydb_server_occ::transaction_abort()`

```
ydb_protocol::status ydb_server_occ::transaction_abort(
                                    ydb_protocol::transaction_id id, int &)
{
    trans_entry_t *trans = find_trans(id);      // find transaction id in list
    if (!trans) {                               // check invalid transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    del_trans(id);                              // delete the transaction list
    printf("T%d: transaction abort\n", id);
    return ydb_protocol::OK;
}
```

- `ydb_server_occ::get()`

```
ydb_protocol::status ydb_server_occ::get(ydb_protocol::transaction_id id,
                                 const std::string key, std::string &out_value) {
    extent_protocol::extentid_t eid = BKDHash(key); // hash key
    trans_entry_t *trans = find_trans(id);          // find transaction entry
    if (!trans) {                                   // check invalid transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    // find in cache first in write set then read set
    int write_size = trans->write_set.size();
    for (int i = 0; i < write_size; ++i) {
        if (trans->write_set[i].eid == eid) {
            out_value = trans->write_set[i].value;
            printf("T%d: get(%s) = %s\n", id, key.c_str(),
out_value.c_str());
            return ydb_protocol::OK;
        }
    }
    int read_size = trans->read_set.size();
    for (int i = 0; i < read_size; ++i) {
        if (trans->read_set[i].eid == eid) {
            out_value = trans->read_set[i].value;
            printf("T%d: get(%s) = %s\n", id, key.c_str(),
out_value.c_str());
            return ydb_protocol::OK;
        }
    }
    // first time to read
    ec->get(eid, out_value);
    // alloc new cache entry
    cache_entry_occ_t c(eid, out_value);
    trans->read_set.push_back(c);
    printf("T%d: get(%s) = %s\n", id, key.c_str(), out_value.c_str());
    return ydb_protocol::OK;
}
```

- `ydb_server_occ::set()`

```
ydb_protocol::status ydb_server_occ::set(ydb_protocol::transaction_id id,
                          const std::string key, const std::string value,
int &) {
    extent_protocol::extentid_t eid = BKDHash(key); // hash key
    trans_entry_t *trans = find_trans(id);          // find transaction
entry
    if (!trans) {                                   // check invalid
transaction id
        printf("find_trans fault!\n");
        return ydb_protocol::TRANSIDINV;
    }
    int write_size = trans->write_set.size();
    for (int i = 0; i < write_size; ++i) {      // find cache entry in write
set
        if (trans->write_set[i].eid == eid) {   // cache hit
            trans->write_set[i].value = value;
            printf("T%d: set(%s) = %s\n", id, key.c_str(), value.c_str());
            return ydb_protocol::OK;
        }
    }
    // first time to write
    cache_entry_occ_t c(eid, value);
    trans->write_set.push_back(c);
    printf("T%d: set(%s) = %s\n", id, key.c_str(), value.c_str());
    return ydb_protocol::OK;
}
```

- `ydb_server_occ::del()`

  The logic is the same as set operation.