CS223 Report: Part 2

1 Introduction

Two-phase commit Protocol (2PC) is a type of atomic commitment protocol which coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort (roll back) the transaction. In this project I implemented distributed transactions using 2PC protocol based on PostgreSQL. SQL statements on IOT data are divided into transactions, *e*.g., SQLs within 10 minutes are equated as a transaction, and the SQL statements in a transaction are partitioned to n groups based on a hash of ¡sensor-id, timestamp¿ in the statement, where n is the number of the distributed agents. Then the partitioned transaction is forwarded to the n agents. At each of the agents, a PostgreSQL server is running to mimic a distributed system. And a coordinator will interact with the agents, routing the SQL statements to the corresponding agents to process distributed transactions. The protocol is summarized in Fig. 1

2 Entities

This project includes three entities, a job reader, a coordinator, and several agents to enable distributed transactions on multiple PostgreSQL servers.
Job Reader. The job reader reads SQL statements from files and transfers the statements to transactions. The transactions are stored in a queue, and once a coordinator is to process a transaction, he pops a transaction from the queue.
Coordinator. The coordinator first gets a transaction from the job reader, and then allocate the transaction to agents [1]. Then it broadcasts a message of PREPARE to agents, and waits those agents to return votes after processing SQL statements. After it receives all the votes, it generates a vote result. If all the agents return COMMIT votes, then the vote result is COMMIT; otherwise, the result is ABORT. The vote result is broadcasted to agents, and the coordinator then waits the agents to finish the transaction and return a ACKNOWLEDGE. After it receiving all the ACKNOWLEDGE messages, the coordinator set his status to ACKNOWLEDGED, which means an end of the current transaction.
Agent. After an agent gets SQL statements (the partitioned transaction) from the coordinator, the agent prepares the transaction and return a vote. After it gets a vote result from the coordinator, the agent then commits or aborts the transaction, and finally return an ACKNOWLEDGE message to the coordinator.

3 Recovery

In this project, machines, *i.e*, a coordinator and several agents, maintain a log to record states of transactions in case of failures. When failures happen on a machine, such as, the coordinator or the agents crashes down, the uncompleted transaction can be processed

[1] The partition is based on a hash of sensor-id and timestamp.

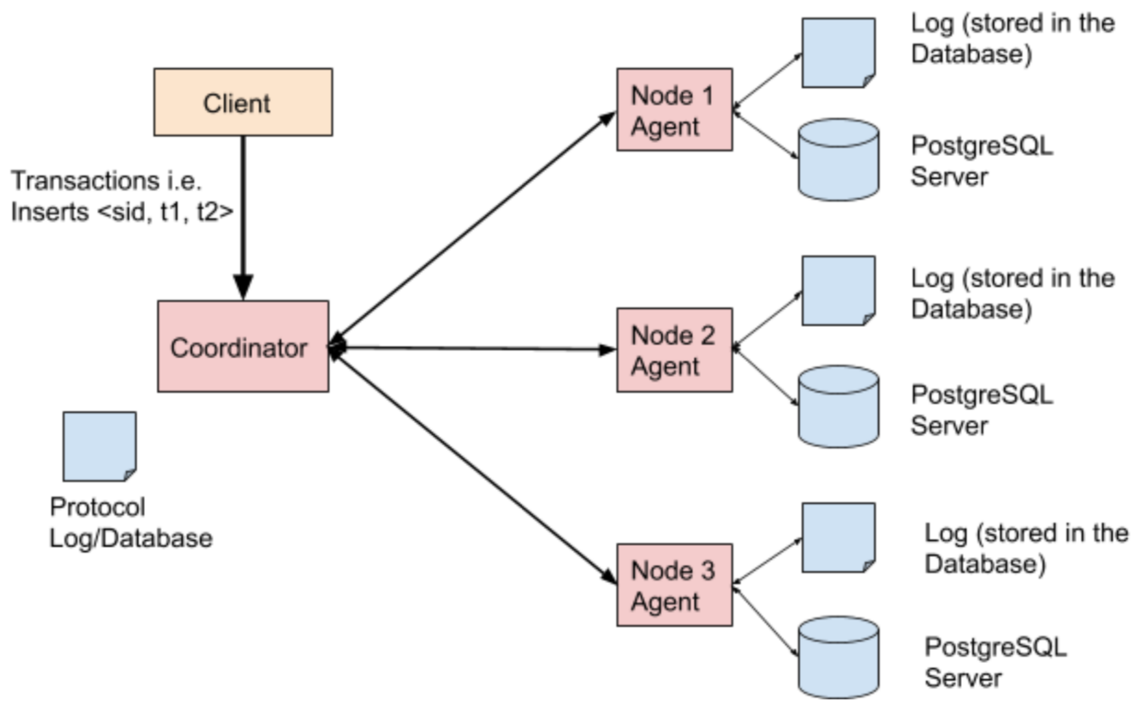

Figure 1: 2PC Protocol

based on this log after that machine comes back. For example, an agent node goes down after it finishes preparing a transaction and sending a vote message of ABORT to the coordinator. After sometime, the agent comes back, and it will load logs to continue processing the unfinished transaction.

In this project, logs are recorded in a table with attributes of log id *lid*, machine id *machine_id*, transaction id *tid*, and status *status*, which records identifiers for log, machines, transactions, and current status. The table is created as follows. Note that the log id is a self-increment attribute, where later inserted logs have larger log id than the earlier inserted ones.

```
CREATE TABLE IF NOT EXISTS LOG_TABLE (
  lid SERIAL PRIMARY KEY ,
  machine_id varchar (100),
  tid varchar (100),
  status varchar (100)
);
```

There are three types of logs, *i*) coordinator logs, *ii*) agent logs, and *iii*) job reader logs. The first two logs record activities of a coordinator or agents, and the last logs mark a point where the job reader reads a new transaction. for example, when processing transaction t_1, a coordinator sets its status to PREPARE, and a record of ('COORDINATOR', 't_1', 'PREPARE') is inserted. After an agent, say, a_1 receives the prepare message, he will process his allocated statements and vote. Suppose he votes COMMIT, then a record of ('a_1', 't_1', 'COMMIT') will be inserted... and so on. After the current transaction t_1 is completed, the job reader then reads a new job and inserts a log to mark the time point of a new transaction.

Totally, there are 5 possible status for a coordinator, including INITIATE, PREPARE, COMMIT, ABORT, and ACKNOWLEDGED. For agents, the status can be INITIATE, COMMIT, ABORT, ACKNOWLEDGE, COMMIT_A_TRANSACTION, and ABORT_A_TRANSACTION. Note that the job log only acts as a sign of a timestamp where the last transaction has already been finished, so here we don't need to record the transaction id. Actually we can get the max log id of the job reader's log. And if the log id is larger than all the ids of the last transaction, we can say that after the last round of processing, the coordinator has received a new job. Thus we can simply set the job reader's log to ('JOB_READER', 'JOB', 'JOB'). Based on these idea, we design recovery algorithms for the coordinator and the agents.

3.1 Coordinator Recovery

When a crashed coordinator wants to recover his unfinished transaction, he needs to first get his crash point, *i.e.*, the status when he crashed down. Considering that the status of the coordinator can be INITIATE, PREPARE, COMMIT, ABORT, and ACKNOWLEDGED, the coordinator queries his log, and then processes the unfinished functions to complete the status for a whole transaction. For example, if the coordinator fails with a latest log of PREPARE, he needs to first waits agents to process SQL statements and then collects their votes, and sends a message of COMMIT or ABORT, then waits for agents to finish processing, and finally collects ACKNOWLEDGE and sets status to ACKNOWLEDGED.

3.2 Agent Recovery

When an agent wants to recover his unfinished transaction, he first queries the log table for his latest log as well as the coordinator's latest log. Given a transaction, a status of a specific machine means all the status before that one have already been successfully finished. So, to determine the crash point, the agent only needs to know the latest log. Based on different cases of the status of the machines, *i*.e., the coordinator $\mathcal{C}$'s status, and the crashed agent $\mathcal{A}$'s status, there exist the following cases.
$\mathcal{C}$'s status = INITIATE. At the crash point the coordinator just initiate a new transaction. Nothing is uncompleted, and the agent don't need to do anything.
$\mathcal{C}$'s status = ACKNOWLEDGED. The transaction at the crash point has already been completed, and the agent don't need to do anything.
$\mathcal{C}$'s status = PREPARE. If the agent log is INITIATE, the agent need to process the uncompleted transaction, execute SQLs, prepares the transaction, and then votes. If the agent log already shows COMMIT or ABORT, the agent has already finished preparing transactions, so it doesn't need to do anything.
$\mathcal{C}$'s status = COMMIT or $\mathcal{C}$'s status = ABORT. If the agent log shows ACKNOWLEDGE, he has already completed the transaction when he crashed down, so he doesn't need to do anything. If the agent log shows he has already committed or aborted the transaction, *i.e.*, COMMIT_A_TRANSACTION or ABORT_A_TRANSACTION, the agent has already finished operations but he did not send acknowledge to the coordinator. So here the agent needs to send to set the status to ACKNOWLEDGE. If the agent log shows COMMIT or ABORT, it indicates the agent failed after he voted. So he needs to process committing (or aborting) the transaction and then sends an acknowledge to the coordinator.