Participant Overview!

Mark Hannum

Quick Overview

- We are implementing two-phase-commit (2PC)
- This PR is concerned with implementing the low-level participant logic
- Specifically, prepared transactions must learn their outcome from the coordinator
- Crashing is no excuse: new masters must identify prepared transactions and resolve them

OH NO PARTICIPANT CRASH!

- Normally recovery will roll back uncommitted transactions
- Unresolved prepares are still “in-flight”
- One strategy could be to add an additional phase of recovery
- openfiles -> forward-roll -> backward-roll -> resolve-prepares
- Downside #1: no-writes will proceed while this is playing out
- Downside #2: limited ability to interact with the new master

“Online” Approach

- Is similar to ‘online-recovery’ code
- New master keeps a list and acquires locks for unresolved prepares
- In addition to the list of locks, the prepare record has coordinator’s information
- The new master needs this to learn the outcome of the transaction
- A future PR will spawn a thread which uses this information to resolve prepared transactions
- Online approach allows the new master to write transactions, and allows users to interact with the database

Simplified “Normal Operations”

- Today, the block-processor writes a ***COMMIT*** record
- A participant should instead write a ***PREPARE*** record
- Conceptually the next line of code blocks on the ***COORDINATOR***
- Coordinator alone tells us to write ***COMMIT*** or ***ABORT*** record
- Simplest implementation is to wait inline with the block-processor
- To facilitate ***PREPARE*** we need some additional Berkley logic and logging

Berkley TXN changes for MASTER

Introducing txn->dist_prepare .. !

```
static int
__txn_dist_prepare_pp(txnp, dist_txnid,
coordinator_name, coordinator_tier,
coordinator_gen, blkseq_key, lflags)
      DB_TXN *txnp;
      const char *dist_txnid;
      const char *coordinator_name;
      const char *coordinator_tier;
      u_int32_t coordinator_gen;
      DBT *blkseq_key;
      u_int32_t lflags;
```

- Store dist_txnid, coordinator_information & blkseq_key in txnp struct
- Write ***PREPARE*** record / mark txnp as ***PREPARED***
- Retain ***all write-locks***, retain ***read tablelocks,*** discard ***read page locks***
- Berkley will automatically do the right thing if a prepared txnp is committed or aborted

Surgical view of txn.c “Normal Operations”

- ***PREPARES*** call lock_vec with **DB_LOCK_PREPARE** rather than **DB_LOCK_PUT_READ** - Produces same lock-list but maintains table readlocks
- Transactions being prepared then emit a ***DIST_PREPARE*** log-record, but maintain the txn-struct
- Already prepared transactions automatically emit a ***DIST_COMMIT*** log-record & do normal transaction tear-down
- Similarly, a transaction abort will emit a ***DIST_ABORT*** log-record & do normal transaction tear-down
- A prepared-txn which is ***DISCARDED*** (maybe for a master-downgrade) must never reclaim allocated pages
- Checkpoint code considers prepared-but-unresolved transactions for determining it’s ckp-lsn- but that is because of a race between recovery & a new-master upgrading prepares

Prepare log record - the “Heart” of this PR

berkdb/dbinc_auto/txn_auto.h

```
#define DB___txn_dist_prepare  17
typedef struct __txn_dist_prepare_args {
    u_int32_t type;
    DB_TXN *txnid;
    DB_LSN prev_lsn;
    u_int32_t generation;
    DB_LSN begin_lsn;
    DBT dist_txnid;
    u_int64_t genid;
    u_int32_t lflags;
    u_int32_t coordinator_gen;
    DBT coordinator_name;
    DBT coordinator_tier;
    DBT blkseq_key;
    DBT locks;
} __txn_dist_prepare_args;
```

- begin_lsn : checkpoint code should consider recovered-transactions as “active”
- Logging this allows us to set an allocated txn’s begin_lsn without collecting
- A recovered prepare has to set the gblcontext to prevent genid48 overlaps
- Aborted prepares will need to update the blkseq tmp-table
- Prepared txns can only be aborted if coordinator aborts
- Txn-’s locks are stored in prepare, not commit

Unresolved Prepares

- ***Unresolved prepares*** are tracked in ***DB_TXN_PREPARED*** structures on both master and replicants- see berkdb/txn/txn_util.c
- The master updates these structures directly from txn.c, inline with preparing, committing, or aborting transactions
- Replicants keep these updated from rep_record.c, after writing a ***PREPARE***, ***DIST_COMMIT*** or ***DIST_ABORT*** record
- Cold-start or single-node deployment will populate the ***DB_TXN_PREPARED*** structures from recovery

Recovery Gotchas!

- Unresolved ***PREPARED*** transactions are ROLLED BACK
- ***RECOVERY*** code will try to add allocated pages to the B-tree’s freelist for aborted transactions
- ***We cannot allow this for any unresolved prepares***
- To solve this, the code maintains a utxnid-hash of unresolved prepares
- pg_alloc_recover checks this before adding a page to the limbo-list
- As an aside, while limbo-code for txn-aborts is solid… ***limbo-code for RECOVERY will add to the freelist without logging***
- This is obviously suspect, but out of scope here .. (I will study later)

One Final Recovery Gotcha!

- A prepared transaction may be committed by a ***different master***
- Normal-recovery detects mastership changes, and assumes any non-committed transaction at that point should be aborted
- But dist-transactions may actually be ***COMMITTED*** by a different master!
- Obviously this shouldn’t roll back :)
- So prepare is “special cased”: DB_TXN_DIST_ADD_TXNLIST
- If committed by a different master, we update the Berkley txnlist to TXN_COMMIT
- If not committed, then it is a normal, ***unresolved prepare***

Another Interesting Gotcha: *DOWNGRADE*

- Imagine a normal transaction is blocked on a lock held by a prepare
- We obviously ***cannot allow the transaction to acquire this lock*** until the prepared is resolved
- Downgrade has to somehow abort all normal transactions which are blocked on ***PREPARED*** transactions
- We can abort ***RECOVERED PREPARED*** transactions after we have acquired the BDB-lock in ***WRITE*** mode
- ***WEIRD-TRICK*** is to issue a deadlock to every txn blocked on a prepared-txn
- ***NORMAL PREPARE*** case (as opposed to "recovered-prepared") isn't written yet: simplest approach is "do nothing": block downgrade until all prepares are resolved

Replace COMMIT with PREPARE->COMMIT

.. for TESTING .. !

- A new tunable, ‘debug_all_prepare_commit’, does exactly this
- Running roborivers with this flag ***passes almost all tests***
- The failures fall into 2 categories: ***performance*** and ***dangling-prepares***
- **PREPARE**+**COMMIT** will always be slower than simple **COMMIT** because replication forces a __log_flush upon receiving a **PREPARE**
- Tests which do alot of restarts (i.e., sc_downgrade) can leave ***dangling prepares***
- Since there’s no actual coordinator (right now), these can’t be resolved