Challenges of Concurrent DDL

Why is this such a hard problem, and is there anything we can do about it?

- Robert Haas | PGCon 2019

Overview

- Problem Statement
- Acceptance Criteria
- Well-Defined Semantics
- The Relation Cache + Invalidation Messages
- Plan Invalidation
- Multi-Step Changes
- Lock Upgrade Hazards

Problem Statement

Allow users to

change the definition of an object **(DDL)**

while

the object is being used. **(Concurrent)**

Acceptance Criteria

- *Comprehensible*. We must be able to explain the behavior to users. This implies that it must be predictable and not too strange.
- *Reliable*. The system must not crash, hang, spit out scary internal error messages, corrupt data, etc.

Well-Defined Semantics

- What happens if these two things are happening at the same time?
  - ALTER TABLE foo DETACH PARTITION foo1;
  - COPY foo FROM ...

- If any rows are routed to foo1 after the DETACH operation, we could:
  - Store them into foo1 anyway.
  - Throw them away.
  - Emit an error.
  - Something else?

Well-Defined Semantics (2)

- What happens if these two things are happening at the same time?
  - ALTER TABLE foo ADD CONSTRAINT ...
  - COPY foo FROM ...

- If COPY inserts any rows that violate the constraint, we could:
  - ~~Constraint ends up violated~~.
  - Discard the rows.
  - Emit an error.
  - Something else?

Well-Defined Semantics (3)

- What happens if these two things are happening at the same time?
  - ALTER TABLE ... SET (fillfactor = 90);
  - COPY foo FROM …

- The new value will take effect “eventually,” no later than the start of the next transaction, and maybe earlier.

The Relation Cache vs. DDL

- Each backend stores metadata about each table it has accessed in the **relcache**. Might be out-of-date if other sessions have performed DDL.
- When a backend performs DDL on an object, it sends invalidation messages to a shared queue. Sometimes we use the abbreviation **sinval** ("shared invalidation").
- Other backends later read these messages and invalidate their local caches.
- For the system to function as intended, it must be guaranteed that each backend which might have cached data notices the invalidation messages "soon enough."

Locking Provides Sequencing

- When a transaction commits, invalidation messages are added to the shared queue *before* releasing locks.
- When a transaction acquires a lock on a relation, it checks for new invalidation messages *after* acquiring the lock.
- The cache will never contain stale information provided that the relation lock held by the transaction performing DDL conflicts with the relation lock the other transaction is attempting to acquire.
- The data used to build the cache entry will never change while the entry is being read provided that AccessExclusiveLock is used for all DDL.

ENTERPRISEDB

Some Invalidation Gotchas

- Invalidation messages are processed at the beginning of each transaction, and whenever we take a new heavyweight lock, and at some other times.
- Typically, this means that we process invalidations at the beginning of each statement and not afterwards.
- However, we might not process invalidations until as late as the start of the next transaction.
- And on the other hand, we might process them in the middle of running the current statement.
- Whenever we process invalidations, we process *all* pending invalidation messages, not just those pertaining to the relation we locked.

Reducing Lock Levels Breaks Everything

- The relcache contents might be stale.
  - DDL could have committed after we acquired all of our locks.
- The relcache contents might change between one access and the next.
  - Even though we hold a lock, concurrent DDL could still commit meanwhile.
- The underlying data could even change while we are in the process of rebuilding the relcache entry.
  - All data is now read from the catalogs using MVCC snapshots, but different bits of data might be read using different snapshots.
- A relcache data structure to which we hold a pointer might get freed at a surprising time.
  - At any point where we might process invalidation messages, a relcache rebuild could occur and the underlying data might have changed.

Stale Pointer Example

```
TriggerDesc *tg = rel->trigdesc;
HeapTuple tup = SearchSysCache1(…);
int i;

for (i = 0; i < tg->numtriggers; ++i)
{
    /* do something with tg->triggers[i] */
}

ReleaseSysCache(tup);
```

Relation Cache Rebuild: Example Hazard

```
inhoids = find_inheritance_children(rel);

foreach (lc, inhoids)
{
    tuple = SearchSysCache1(RELOID, inhrelid);
    /* … */
}
```

- find_inheritance_children() uses a current snapshot and direct catalog access.
- SearchSysCache1 uses cached information that might be *older or newer*.

Why Does Concurrent DDL Break This?

```
build_simple_rel(int relid) /* simplified, from v11 */
{
    rel->part_rels =
        palloc(sizeof(RelOptInfo *) * rel->nparts);
    foreach(l, append_rel_list)
    {
        if (appinfo->parent_relid != relid)
            continue;
        childrel = build_simple_rel(…);
        rel->part_rels[cnt_parts] = childrel;
        cnt_parts++;
    }
    Assert(cnt_parts == nparts);
}
```

Don’t Ask The Same Question Twice!

```
build_simple_rel(int relid) /* simplified, from v11 */
{
    rel->part_rels =
        palloc(sizeof(RelOptInfo *) * rel->nparts);
    foreach(l, append_rel_list)
    {
        if (appinfo->parent_relid != relid)
            continue;
        childrel = build_simple_rel(…);
        rel->part_rels[cnt_parts] = childrel;
        cnt_parts++;
    }
    Assert(cnt_parts == nparts);
}
```

Plan Invalidation

- Shared invalidation messages not only invalidate relcache entries but also cached plans!
- Reducing the lock level below AccessExclusiveLock creates a risk that an “old” plan will be executed.
- If the information is non-critical, e.g. whether newly-inserted values can be TOAST-compressed, a small race of this kind may be acceptable.
- However, it’s clearly unacceptable for critical data such as column types.

Plan Invalidation Examples

- Concurrent ATTACH PARTITION: Just ignore the new partitions.
- Concurrent DETACH PARTITION: What do we do about partitions that are not partitions any more? And that maybe have been dropped or further altered?
- Concurrent ADD COLUMN: Just ignore the new column.
- Concurrent DROP INDEX: What if the plan uses the dropped index?

Multi-Step Changes: Examples

- Existing Cases:
  - CREATE INDEX CONCURRENTLY
  - REINDEX INDEX CONCURRENTLY
  - DROP INDEX CONCURRENTLY
- Wish List:
  - Enable checksums on a running cluster
  - Table-rewriting operations such as CLUSTER

Multi-Step Changes: Strategy

- Change some kind of state to let everyone know that the change is in progress.
- Wait until you're sure that everyone knows about this initial change.
- Then do the next step of the process.

- For instance, for DROP INDEX CONCURRENTLY:
  1. "Please don't read from this index." … wait
  2. "Please don't insert into to this index." … wait
  3. Remove index.

Multi-Step Changes: Inefficient Waiting

- We have no way of knowing which backends have read any shared invalidation messages we've sent.
- And we have no way of getting them to do so quickly.
- Current approach is to collect a list of transactions that have the index locked, and then wait until all of those transactions have ended.
- They might have actually read the invalidation messages much sooner, but we don't know!
- Possible solution: Andres Freund's global barrier stuff.

Multi-Step Changes: Garbage

- If a backend crashes while performing one of these multi-step sequences, there is no mechanism to clean things up automatically.
- The changes made and committed in earlier stages remain in effect, but the work doesn't get completed.
- Typical result: We pay for an index that we don't get to use.
- Could potentially be fixed by some kind of background worker.
- Multi-step changes are a powerful technique, but every new use of this technique adds a new kind of "garbage" risk.

Locking Considerations

- Any DDL statement must acquire the strongest lock it will need at the beginning of the operation, or risk deadlock upon upgrade.
- For example, suppose process A acquires ShareUpdateExclusiveLock and later AccessExclusiveLock.
- Normally, that's fine, but if process B acquires AccessShareLock and then later AccessExclusiveLock, deadlock will occur.
- It's pretty sad if the process that is aborted is one that has done a lot of work.

Thanks

- Any questions?

ENTERPRISEDB