Seata AT mode

The basic idea

Prerequisite

- Relational databases that support local ACID transaction.
- Java applications that access database via JDBC.

Overall mechanism

Evolution from the two phases commit protocol:

- Phase 1：commit business data and rollback log in the same local transaction, then release local lock and connection resources.
- Phase 2：
  - for commit case, do the work asynchronously and quickly.
  - for rollback case, do compensation, base on the rollback log created in the phase 1.

Write isolation

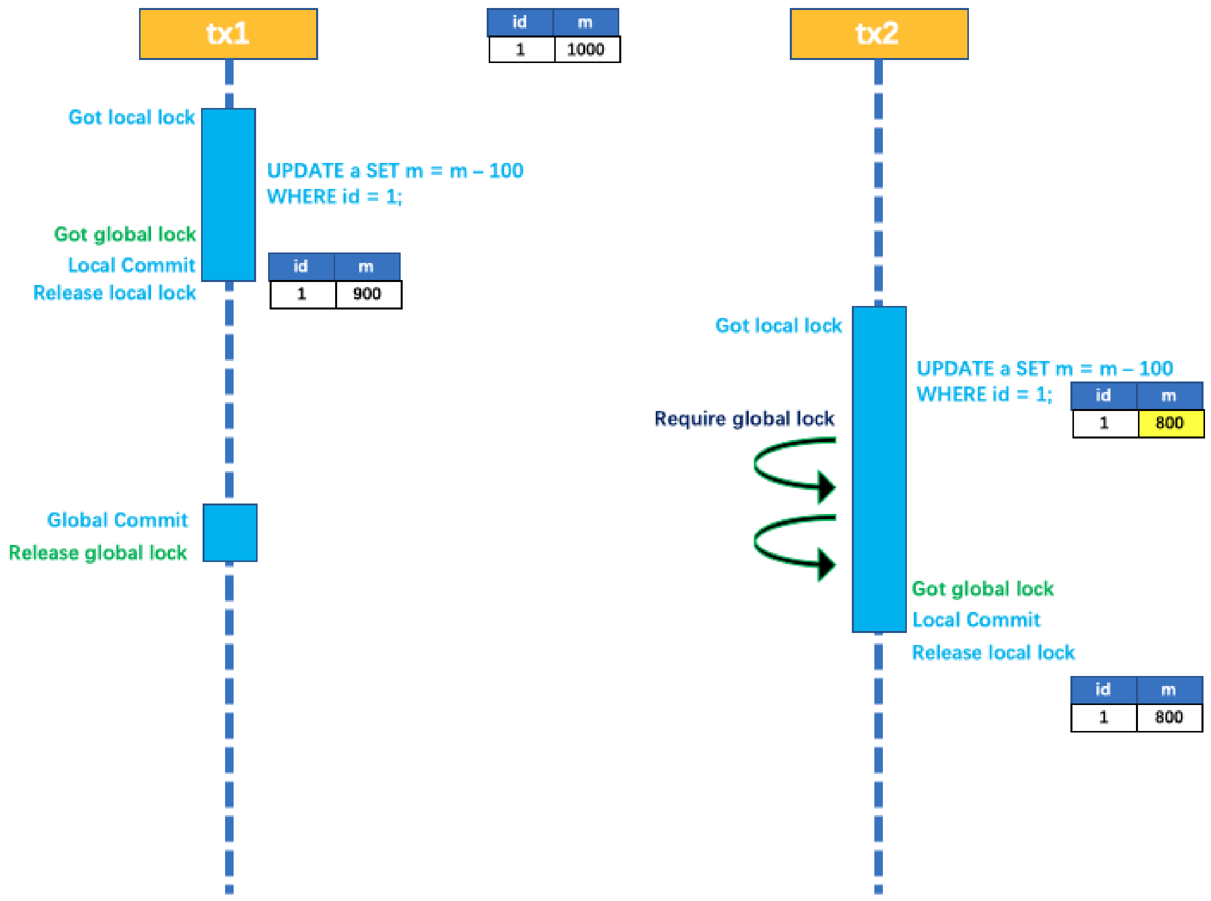
- The global lock must be acquired before committing the local transaction of phase 1.
- If the global lock is not acquired, the local transaction should not be committed.
- One transaction will try to acquire the global lock many times if it fails to, but there is a timeout, if it's timeout, rollback local transaction and release local lock as well.

For example:

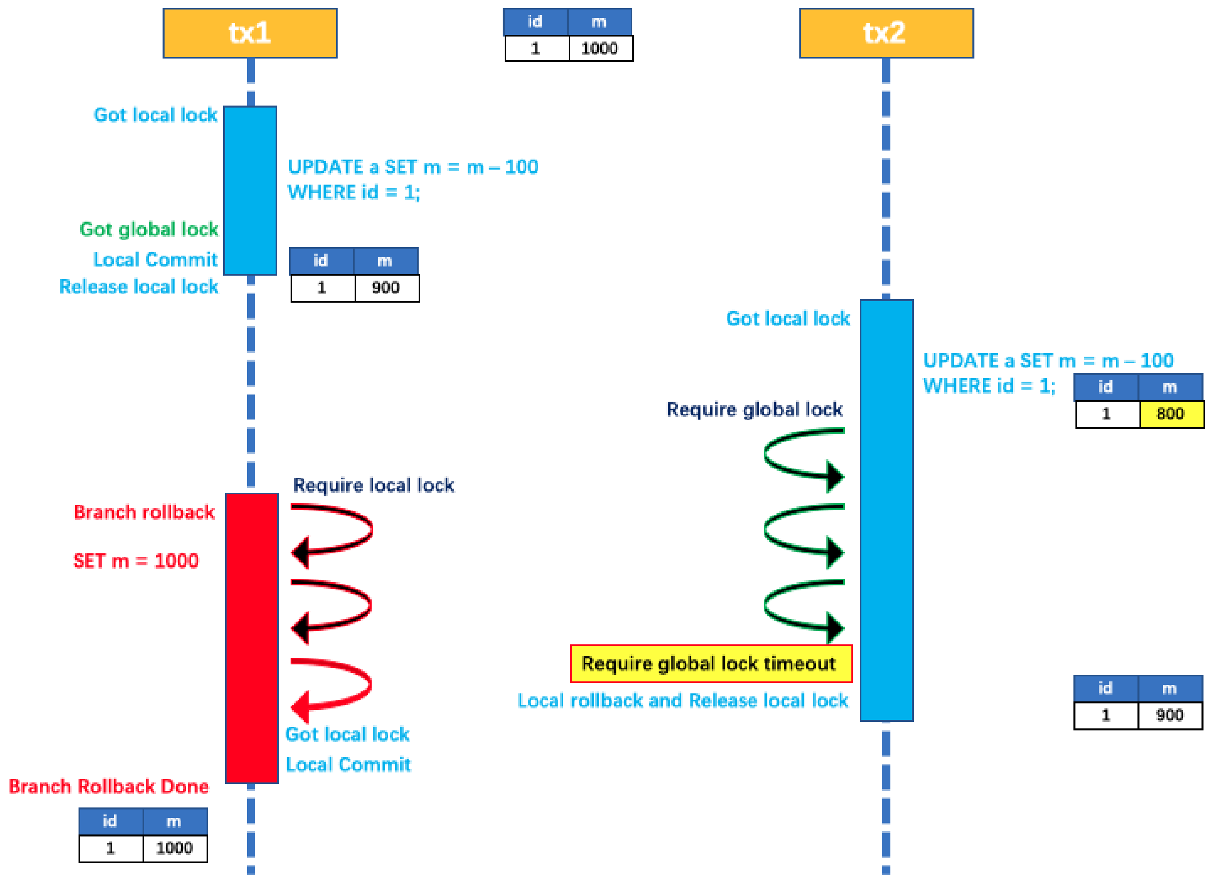
Two transactions tx1 and tx2 are trying to update field m of table a. The original value of m is 1000.

tx1 starts first, begins a local transaction, acquires the local lock, do the update operation: m = 1000 - 100 = 900. tx1 must acquire the global lock before committing the local transaction, after that, commit local transaction and release local lock.

next, tx2 begins local transaction, acquires local lock, do the update operation: m = 900 - 100 = 800. Before tx2 can commit local transaction, it must acquire the global lock, but the global lock may be hold by tx1, so tx2 will do retry. After tx1 does the global commit and releases the global lock, tx2 can acquire the global lock, then it can commit local transaction and release local lock.



See the figure above, tx1 does the global commit in phase 2 and release the global lock, tx2 acquires the global lock and commits local transaction.



See the figure above, if tx1 wants to do the global rollback, it must acquire local lock to revert the update operation of phase 1.

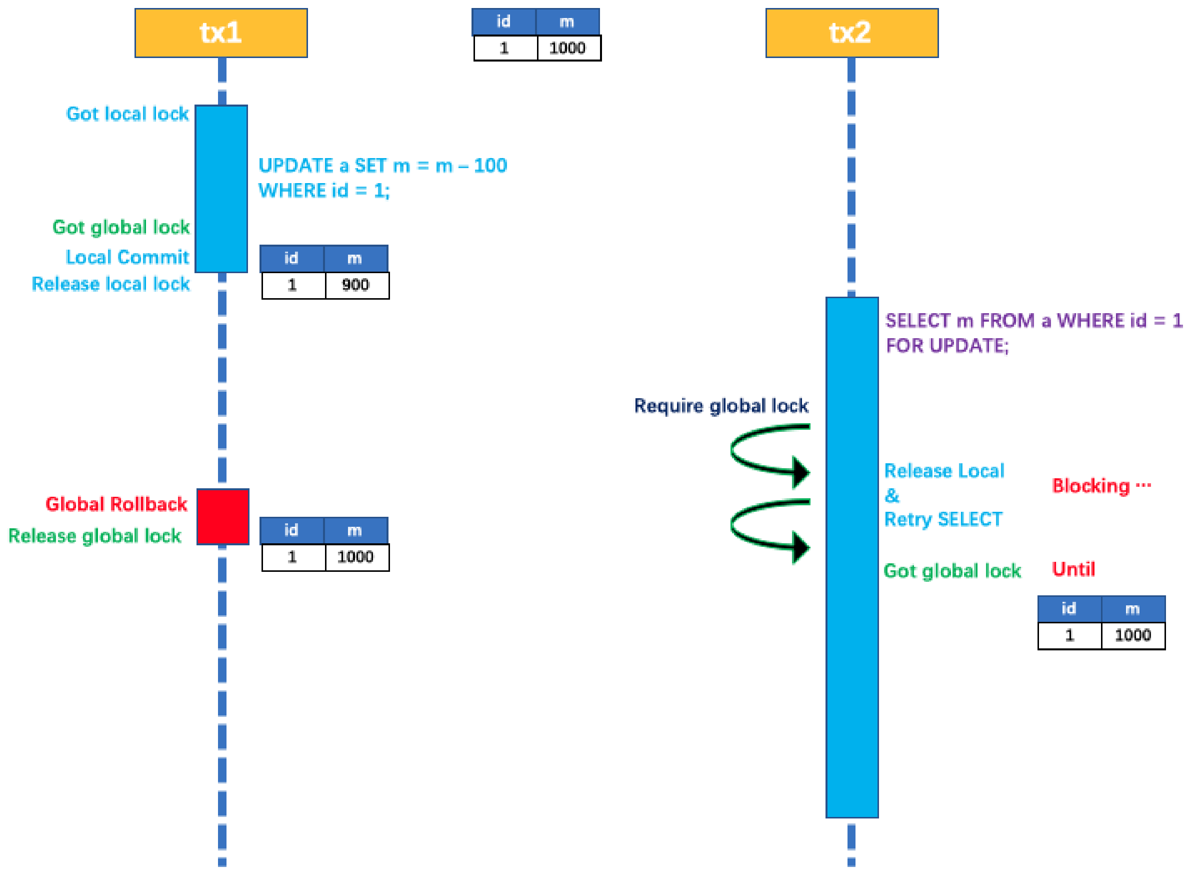
However, now the local lock is held by tx2 which hopes to acquire the global lock, so tx1 fails to rollback, but it would try it many times until it's timeout for tx2 to acquire the global lock, then tx2 rollbacks local transaction and releases local lock, after that, tx1 can acquire the local lock, and do the branch rollback successfully.

Because the global lock is held by tx1 during the whole process, there isn't no problem of dirty write.

Read isolation

The isolation level of local database is read committed or above, so the default isolation level of the global transaction is read uncommitted.

If it needs the isolation level of the global transaction is read committed, currently, Fescar implements it via SELECT FOR UPDATE statement.



The global lock is be applied during the execution of SELECT FOR UPDATE statement, if the global lock is held by other transactions, the transaction will release local lock retry execute the SELECT FOR UPDATE statement. During the whole process, the query is blocked until the global lock is acquired, if the lock is acquired, it means the other global transaction has committed, so the isolation level of global transaction is read committed.

For the performance consideration, Fescar only does proxy work for SELECT FOR UPDATE. For the general SELECT statement, do nothing.

Work process

Take an example to illustrate it.

A business table: `product`

Field	Type	Key
id	bigint(20)	PRI
name	varchar(100)	
since	varchar(100)	

The sql of branch transaction in AT mode:

```
update product set name = 'GTS' where name = 'TXC';
```

Phase 1

Process:

1. Parse sql: know the sql type is update operation, table name is product, the where condition is name = 'TXC' and so on.
2. Query the data before update(Named before image): In order to locate the data that will be updated, generate a query statement by the where condition above.

```
select id, name, since from product where name = 'TXC';
```

Got the "before image":

id	name	since
1	TXC	2014

3. Execute the update sql: update the record of name equals 'GTS'.
4. Query the data after update(Named after image): locate the record by the primary key of image data before update.

```
select id, name, since from product where id = 1;
```

Got the after image:

id	name	since
1	GTS	2014

5. Insert a rollback log: build the rollback log with image before and after, as well as SQL statement relelated information, then insert into table `UNDO_LOG` .

```
{
	"branchId": 641789253,
	"undoItems": [{
		"afterImage": {
			"rows": [{
				"fields": [{
					"name": "id",
					"type": 4,
					"value": 1
				}, {
					"name": "name",
					"type": 12,
					"value": "GTS"
				}, {
					"name": "since",
					"type": 12,
					"value": "2014"
				}]
			}],
			"tableName": "product"
		},
		"beforeImage": {
			"rows": [{
				"fields": [{
					"name": "id",
					"type": 4,
					"value": 1
				}, {
					"name": "name",
					"type": 12,
					"value": "TXC"
				}, {
					"name": "since",
					"type": 12,
					"value": "2014"
				}]
			}],
			"tableName": "product"
```

```
                },
                "sqlType": "UPDATE"
        }],
        "xid": "xid:xxx"
}
```

6. Before local commit, the transaction submmit an application to TC to acquire a global lock for the record whose primary key equals 1 in the table product.
7. Commit local transaction: commit the update of PRODUCT table and the insert of UNDO_LOG table in the same local transaction.
8. Report the result of step 7 to TC.

Phase 2 - Rollback case

1. After receive the rollback request from TC, begin a local transaction, execute operation as following.
2. Retrieve the UNDO LOG by XID and Branch ID.
3. Validate data: Compare the image data after update in UNDO LOG with current data, if there is difference, it means the data has been changed by operation out of current transaction, it should be handled in different policy, we will describe it detailedly in other document.
4. Generate rollback SQL statement base on before image in UNDO LOG and related information of the business SQL.

```
update product set name = 'TXC' where id = 1;
```

5. Commit local transaction, report the result of execution of local transaction(The rollback result of the Branch transaction) to TC.

Phase 2 - Commit case

1. After receive the commit request from TC, put the request into a work queue, return success to TC immediately.
2. During the phase of doing the asynchronous work in the queue, the UNDO LOGs are deleted in batch way.

Appendix

Undo log table

UNDO_LOG Table：there is a little bit difference on the data type for different databases.

For MySQL example:

Field	Type
branch_id	bigint PK
xid	varchar(100)
rollback_info	longblob
log_status	tinyint
log_created	datetime
log_modified	datetime
ext	varchar(100)

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'increment id',
  `branch_id` bigint(20) NOT NULL COMMENT 'branch transaction id',
  `xid` varchar(100) NOT NULL COMMENT 'global transaction id',
  `context` varchar(128) NOT NULL COMMENT 'undo_log context,such as
serialization',
  `rollback_info` longblob NOT NULL COMMENT 'rollback info',
  `log_status` int(11) NOT NULL COMMENT '0:normal status,1:defense
status',
  `log_created` datetime NOT NULL COMMENT 'create datetime',
  `log_modified` datetime NOT NULL COMMENT 'modify datetime',
  `ext` varchar(100) DEFAULT NULL COMMENT 'reserved field',
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='AT
transaction mode undo table';
```

performance and easy to use distributed transaction services under a microservices architecture.

Report a doc issue

Edit This Page on GitHub