表锁功能开发文档

修订历史

版本	修订日期	修订描述	作者	备注
Cedar 0.2	2016-07-01	表锁功能开发文档	王嘉豪	无

1. 总体设计

1.1 综述

Cedar是由华东师范大学数据工程与科学研究院基于OceanBase 0.4.2 研发的可扩展的关系数据库。在Cedar 0.1中，仅有行级别的锁，没有大粒度的锁（如页锁、范围锁、表锁等）的实现。

为了满足某些应用的需求，需要在Cedar中实现表级锁，实现之后，主动在事务执行过程中主动调用锁表机制有如下作用：

- 在读写时避免幻读：某个事务对表上锁之后，仅允许该事务对表进行修改，其他事务只能读取该表
- 避免长写事务的饥饿：长写事务需要对许多数据加锁，在冲突较高的负载中，加锁成功率很低，导致事务回滚率高

1.2 名词解释

主控服务器（RootServer, RS）：Cedar集群的主控节点，负责管理集群中的所有服务器，以及维护tablet信息。

更新服务器（UpdateServer, UPS）：负责存储Cedar系统的增量数据，并提供事务支持。

基准数据服务器（ChunkServer, CS）：负责存储Cedar系统的基线数据。

合并服务器（MergeServer, MS）：负责接收并解析客户端的SQL请求，经过词法分析、语法分析、查询优化等一系列操作后发送到CS和UPS，合并基线数据和增量数据。

1.3 功能设计

在事务执行时，支持以下的SQL语句用于给t1表添加排它锁：

- `LOCK TABLE t1`

事务需要 调用`LOCK TABLE`语句，否则不会自动添加表锁

1.4 性能指标

不使用表锁时，对原有的性能几乎没有影响。
使用表锁时，事务在该表上的事务执行等价于单线程执行，会严重降低事务执行的并发度，但事务之间的执行是串行化的。

2. 模块设计

2.1 表锁状态设计

一张表的表锁状态有如下几种：

- 无锁（NO_LOCK）
- 共享锁（SHARE_LOCK）
- 意向排它锁（INTENTION_EXCLUSIVE_LOCK）
- 排它锁（EXCLUSIVE_LOCK）

他们的冲突关系为：

	N_LOCK	**S_LOCK**	**IX_LOCK**	**X_LOCK**
N_LOCK	no	no	no	no
S_LOCK	no	no	yes	yes
IX_LOCK	no	yes	no	yes
X_LOCK	no	yes	yes	yes

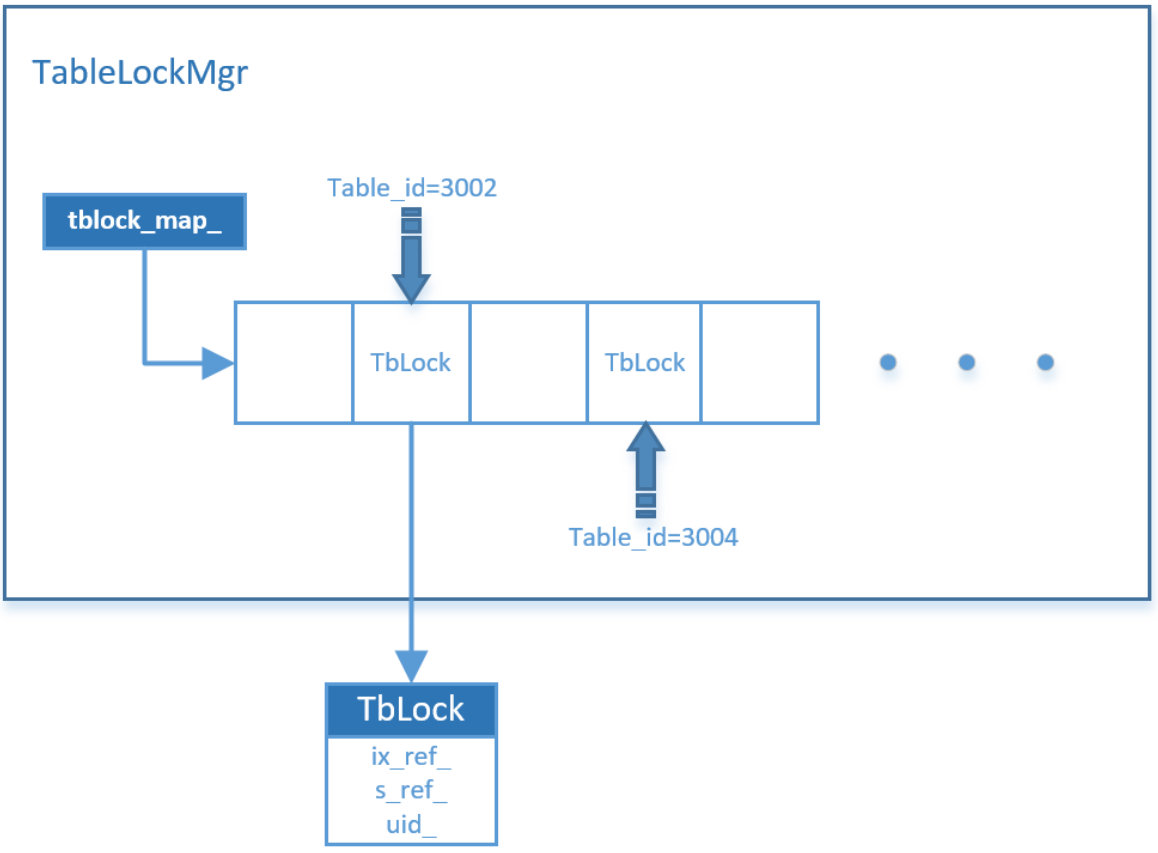
注：no表示不冲突
　　yes表示冲突

行锁与表锁的关系:

- 行锁：仅对某一条记录进行加锁
- 表锁：对整张表加锁
- 行锁与表锁：在事务执行过程中，使用表锁中的“意向排它锁”来兼容这两个锁

2.2 全局表锁管理器模块

表锁的全局管理器（**TableLockMgr**）使用了一个数组（**tblock_map_**）记录每张表对应的表锁（**TbLock**）。每个表锁是一个**TbLock**结构体，记录了加锁信息。如下图所示：



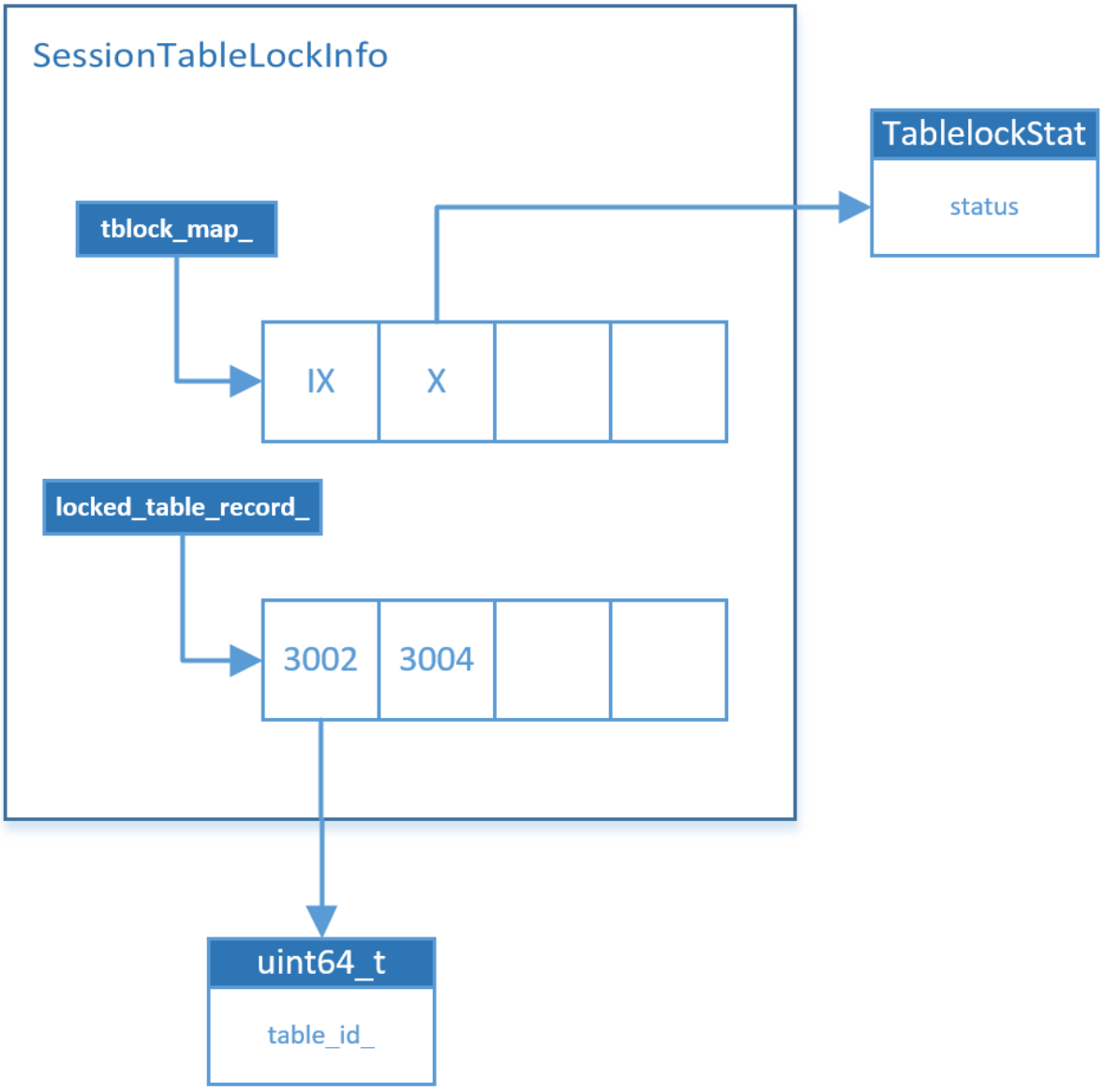
TbLock的属性值变化情况如下：

- 当某张表被某个session使用排他锁锁定时，这三个元素的值为：
  - **ix_ref_** = 0
  - **s_ref_** = 0
  - **uid_** = session_id & EXCLUSIVE_BIT
- 当某张表被3个session使用意向排它锁锁定时，这三个元素的值为：
  - **ix_ref_** = 3
  - **s_ref_** = 0
  - **uid_** = 0
- 当某张表没有被锁定时，这三个元素的值为：
  - **ix_ref_** = 0
  - **s_ref_** = 0
- **uid_** = 0

ix_ref_ 表示添加的意向排他锁的个数
s_ref_表示添加的共享锁的个数
uid_表示给表添加排它锁的session的标识符。

2.3 每个session表锁管理器模块

每个**session**的表锁管理器（**SessionTableLockInfo**）用于维护每个session对哪些表进行了加锁。使用了两个数组进行维护，其中**locked_table_record_**记录了该session上锁的表集合，**tblock_map_**存放每个表的加锁状态。如图所示：



3. 模块接口

3.1 表锁的数据结构

数据结构TbLock中记录着每张表对应的表锁信息， 该结构是上文 2.2 节中描述的每张表的TbLock结构体，具体数据结构如下：

```
struct TbLock  //结构体大小12B
{
    enum Status {
        EXCLUSIVE_BIT = 1UL<<31,
        UID_MASK = ~EXCLUSIVE_BIT
    };
    volatile uint32_t ix_ref_;
    volatile uint32_t s_ref_;
    volatile uint32_t uid_;
}
```

其中：

- **uid_**存放session_id。当某session对表添加排它锁时，由于排它锁只能由惟一的session锁定，因而用session_id来唯一标识。
- **ix_ref**存放意向排他锁的计数，由于意向排他锁之间不冲突，*ix_ref*用于统计有多少个session对该表添加了意向排它锁。
- **s_ref_**是读锁在表锁中没有使用。

表锁数据结构的主要接口如下：

- 其中uid为session_id，end_time为超时时间，wait_flag为是否继续等待的标识符

关于intention lock：

```
int try_intention_lock(uint32_t uid);

int try_intention_unlock(uint32_t uid);

int intention_lock(const uint32_t uid, const int64_t end_time);

int intention_unlock(const uint32_t uid);
```

关于share lock：

```
int try_share_lock(uint32_t uid);

int try_share_unlock(uint32_t uid);

int share_lock(const uint32_t uid, const int64_t end_time);

int share_unlock(const uint32_t uid);
```

关于exclusive lock：

```
int try_exclusive_lock(const uint32_t uid);

int try_exclusive_unlock(const uint32_t uid);

bool is_exclusive_locked_by(const uint32_t uid) const;

int exclusive_lock(const uint32_t uid, const int64_t end_time);

int exclusive_unlock(const uint32_t uid);
```

关于intention lock and exclusive lock：

```
int intention2exclusive_lock(const uint32_t uid,
                             const int64_t end_time);

int exclusive2intention_lock(const uint32_t uid);
```

关于share lock and exclusive lock：

```
int share2exclusive_lock(const uint32_t uid,
                         const int64_t end_time);
```

关于wait lock：

```
int try_wait_other_lock_release(const uint32_t uid);

int wait_other_lock_release(const uint32_t uid,
                            const int64_t end_time);
```

3.2 全局表锁管理器

- 在UPS中设计一个全局的表锁管理器，用于锁定或解锁每张表
- 主要接口：
  - 升级表锁**up_lock**
  - 降低表锁**down_lock**
- 主要数据成员：
  - **tblock_map_**存储了每张表对应的表锁信息
- 代码设计如下：

```
class TableLockMgr
{
  public:
    //升级表锁，包括：
    //NO-LOCK -> IX-LOCK
    //IX-LOCK -> X-LOCK
    //NO-LOCK -> X-LOCK
    int up_lock(uint32_t uid, uint64_t table_id,
                TablelockStat old_stat, TablelockStat new_stat,
                const int64_t end_time...);
    //降级表锁，包括：
    //IX-LOCK -> NO-LOCK
    //X-LOCK -> NO-LOCK
    int down_lock(uint32_t uid, uint64_t table_id,
                  TablelockStat old_stat, TablelockStat new_stat);

  private:
    bool inited_;
    //存储每个table对应的表锁信息
    TbLock* tblock_map_[OB_MAX_TABLE_NUMBER+TABLE_ID_NUMBER_SHIFT];
};
```

3.3 每个session的表锁管理

- 每个session保留自己对那些表进行了加锁
- 代码设计如下：

```
class SessionTableLockInfo : public ISessionCallback
{
  public:
    //对table_id对应的表加锁
    int lock_table(TableLockMgr &globle_tblock, uint32_t uid,
                   uint64_t table_id, TablelockStat new_stat);
    //对table_id对应的表解锁
    int unlock_table(TableLockMgr &globle_tblock, uint32_t uid);
    //session结束时，调用cb_func释放所有锁
    int cb_func(const bool rollback, ...BaseSessionCtx &session);
    int find_table_idx(uint64_t table_id);
```

```
  private:
    RWSessionCtx* session_ctx_;
    //记录被加锁表的状态
    TablelockStat* tblock_map_[OB_MAX_TABLE_NUMBER];
    //记录被加锁表的table_id
    uint64_t locked_table_record_[OB_MAX_TABLE_NUMBER];
    //所有被加锁的表的个数
    uint32_t locked_table_sum_;
};
```

3.4 表锁的初始化

- 在系统启动时，初始化全局的TableLockMgr
- 在每个session创建时，初始化每个session的SessionTableLockInfo

4 使用限制条件和注意事项

- 为表锁添加排它锁需要显式调用 `LOCK TABLE`
- 使用 `LOCK TABLE` 后，将阻塞其他修改该表的事务，TPS可能会严重下降