事务

一、OceanBase 事务

1. 事务的定义：

在数据库的具体应用中，事务发挥着非常重要的作用。事务在应用过程中作为一个逻辑单元，它包含了数据库上的一系列操作。这些操作要么全部成功执行并永久保存，要么全部失败并回滚到事务开始前的状态。这使得数据库从一个一致的状态转化到另一个一致的状态，保证了数据库的一致性和完整性。

2. 事务的特性：

从定义可以看出，数据库事务具有 4 个特性：原子性、一致性、隔离性、持久性，也就是 ACID 特性。

（1）原子性：OceanBase 是一个分布式数据库系统，分布式事务操作的对象可能分布在不同机器上，为了保证事务的原子性，OceanBase 数据库采用两阶段提交协议，确保多台机器上的事务要么都提交成功要么都回滚。

（2）一致性：事务必须是使数据库从一个一致性状态变到另一个一致性状态，与原子性是密切相关的。

（3）隔离性：OceanBase 数据库兼容 Oracle 和 MySQL 模式。在 Oracle 模式下，它支持 Read Committed 隔离级别和 Serializable 隔离级别。在 MySQL 模式下，它支持 Read Committed 隔离级别、Repeatable Read 隔离级别和 Serializabl 隔离级别。这些隔离级别保证了事务的隔离性。

（4）持久化：对于单个机器来说，OceanBase 数据库通过 redo log 记录数据的修改，通过 WAL 机制保证在宕机重启之后能够恢复出来。保证事务一旦提交成功，事务数据一定不会丢失。对于整个集群来说，OceanBase 数据库通过 paxos 协议将数据同步到多个副本，只要多数派副本存活事务数据一定不会丢失。

3. 事务的重要性

OceanBase 的事务有下面几个重要的作用：

（1）数据一致性：事务确保数据库在执行期间保持一致的状态。通过将操作作为原子单元执行，保证了数据的完整性和一致性。如果有操作失败，所有操作都会被撤销，数据库回滚到开始前的状态，从而保持数据的一致性。

（2）并发控制：在多用户环境下，可能会有多个事务同时访问和修改数据库。通过使用并发控制技术，如锁机制和事务隔离级别，事务可以确保并发执行的事务不会相互干扰，防止数据不一致。

（3）故障恢复：事务提供了故障恢复的机制。在数据库系统中遇到故障，通过使用事务日志和检查点机制，可以将事务的操作记录下来，并在故障后进行恢复，以确保数据的持久性和可靠性。

（4）数据完整性：事务可以维护数据库的完整性。通过一些完整性规则和约束，事务在执行期间会确保这些完整性规则得到满足，防止数据的不一致和损坏。

二、OceanBase 事务操作

1.事务的具体结构

在事务的整个生命周期中，通常包括开启事务、执行查询、DML 语句和结束事务等多个过程。事务中会包含一条或者多条 DML 语句，有着明确的起始点及结束点。

在 OceanBase 数据库 V2.x 版本中，单个事务大小有限制，通常是 100M。事务的大小与两个配置项有关，分别是租户级配置项_tenant_max_trx_size 和集群级配置项_max_trx_size，_tenant_max_trx_size 优先生效。

在 OceanBase 数据库 V3.x 版本中因支持了大事务，不再受此限制。

2.OceanBase 中的事务控制过程

（1）开启事务：

在数据库中，开启事务可以通过 BEGIN、START TRANSACTION 等语句显式开启，也可以通过 DML 语句隐式开启。

以 MySQL 模式为例，数据库在执行以下语句时会开启一个事务。

```
begin
start transaction
insert ...
update ...
delete ...
select ... for update...
```

OceanBase 数据库的事务控制语句与 MySQL 数据库兼容，开启事务的方式与上面相同，还包括了执行 SET autocommit = 0 之后再执行的第一条语句这种方式。

当事务开启时，OceanBase 数据库会为事务分配一个事务 ID，用于唯一的标识一个事务。我们通过 oceanbase.__all_virtual_trans_stat 可以查询事务的具体状态。我们以下面例子来举例说明：

```
session 1:
obclient> SET autocommit=0;
Query OK, 0 rows affected (0.00 sec)

obclient> UPDATE t SET c="b" WHERE i=1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

session 2:
obclient> SELECT trans_id FROM __all_virtual_trans_stat;
+-------------------------------------------------------------------------------------+
| trans_id                                                                            |
+-------------------------------------------------------------------------------------+
| {hash:17242042390259891950, inc:98713, addr:"xx.xx.xx.xx:24974", t:1632636623536459} |
+-------------------------------------------------------------------------------------+
```

（2）语句执行：

在语句执行过程中，当语句需要访问数据库中的数据时，OceanBase 数据库会在语句访问到的每个分区上创建一个事务上下文。

这个事务上下文中包含了该语句执行时的数据快照版本号，其中版本号是用来标识数据在时间轴上的位置，每个数据项都有一个对应的版本号，事务在读取数据时只能读

取早于自己开始的事务提交的版本。

事务上下文中还包含了语句对该分区所做的修改。当语句执行过程中对数据进行修改时，这些修改会被记录在事务上下文中。这样，在语句执行完成之前，对数据的修改只会在事务上下文中暂时生效，并不会立即影响到其他事务。

事务上下文的创建和管理保证了语句的隔离性和原子性。通过记录语句执行过程中的数据快照版本号和修改，OceanBase 能够确保多个事务之间的数据访问和修改不会相互干扰，从而保证了数据库的一致性和可靠性。

（3）活跃事务

活跃事务是指那些已经开启的，但还没有提交或者回滚的事务，相当于一个中间的过渡状态。活跃事务所做的修改在提交前都是临时的，别的事务是无法看到的。虚拟表__all_virtual_trans_stat 里的 state 字段标识了所有事务所处的状态。

（4）结束事务：

在 Mysql 中，结束事务通常包含下面的方式，一种方式是提交事务，还有一种方式是回滚事务。除此之外，在一个活跃事务中执行 DDL 语句也会导致隐式地提交事务。

事务结束时会收集事务执行过程中修改过的所有分区，会根据不同的场景对这些分区发起提交事务或者回滚事务。以下场景会触发事务的提交或者回滚。

——用户显式的发起 Commit 来提交事务：

提交事务通过 COMMIT 命令来完成，具体语法如下所示：

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

此外，如果 autocommit = 0，那么执行一条开启事务的语句也会隐式地提交当前进行中的事务。

——用户显式的 Rollback 来回滚事务:

回滚事务通过 ROLLBACK 命令来完成,回滚事务的 SQL 语法如下：

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

——自动提交：

自动提交是指当 autocommit 这个 Session 变量的值为 1 时，此时每条语句执行结束后，OceanBase 数据库将会自动的把这条所在的事务提交。这样一来，一条语句就是一个事务。

——隐式提交：

隐式提交是指用户还未发出 COMMIT 或者 ROLLBACK 等结束事务的语句给 OceanBase 数据库，而 OceanBase 数据库自动将当前活跃的事务执行 COMMIT 提交的过程。

隐式提交会发生在后面两个情形，一是执行 DDL，这包括 CREATE、DROP、RENAME 或者 ALTER 等 DDL 语句；二是执行一个开启事务的语句。

——自动回滚：

自动回滚是用户未发出 ROLLBACK 指令，而是 OceanBase 数据库内部发起的回滚，通常发生在以下情形：一是客户端断开连接，即 session 断开；二是事

务执行超时，即 ob_trx_timeout；三是活跃事务的 session 超过一定时长没有语句执行，即 ob_trx_idle_timeout。

在上面的这些情形下,事务会自动被 OceanBase 数据库回滚,若用户再次在当前 session（未断开）上执行 SQL 则会提示事务已经中断（无法继续）需要回滚，此时用户需要执行 ROLLBACK 来结束当前事务。

3.OceanBase 事务其他知识：

（1）Savepoint 标记

Savepoint 是 OceanBase 数据库中提供的可以由用户定义的一个事务内的执行标记。用户可以通过在事务内定义若干标记并在需要时将事务恢复到指定标记时的状态。

如果用户在执行过程中，在定义了某个 Savepoint 之后执行了一些错误的操作，用户不需要回滚整个事务再重新执行，而是可以通过执行 ROLLBACK TO 某个标记的命令来将 Savepoint 之后的修改回滚。

在 OceanBase 数据库的实现中，事务执行过程中的修改都有一个对应的"sql sequence",该值在事务执行过程中是递增的(不考虑并行执行的场景),创建 Savepoint 的操作实际上是将用户创建的 Savepoint 名字对应到事务执行的当前"sql sequence"上，当执行 ROLLBACK TO 命令时，OceanBase 数据库内部会执行以下操作：

一是将事务内的所有大于该 Savepoint 对应"sql sequence"的修改全部回滚，并释放对应的行锁。

二是删除该 Savepoint 之后创建的所有 Savepoint。

（2）语句超时与事务超时

系统变量 ob_query_timeout 控制着语句执行时间的上限，语句执行时间超过此值会给应用返回语句超时的错误，错误码为-6212，并回滚语句，通常该值默认为 10s。

系统变量 ob_trx_timeout 控制着事务超时时间，事务执行时间超过此值会给应用返回事务超时的错误，错误码为-6210，此时需要应用发起 ROLLBACK 语句回滚该事务。

系统变量 ob_trx_idle_timeout 表示 Session 上一个事务处于的 IDLE 状态的最长时间，即长时间没有 DML 语句或结束该事务，则超过该时间值后，事务会自动回滚，再执行 DML 语句会给应用返回错误码-6224,应用需要发起 ROLLBACK 语句清理 Session 状态。

（3）全局时间戳

全局时间戳服务，简称 GTS，是 OceanBase 数据库内部为每个租户启动一个全局时

间戳服务，事务提交时通过本租户的时间戳服务获取事务版本号，保证全局的事务顺序。

GTS 是集群的核心，保证服务高可用。对于用户租户而言，OceanBase 数据库使用租户级别内部表__all_dummy 的 leader 作为 GTS 服务提供者，时间来源于该 leader 的本地时钟。GTS 默认是三副本的，其高可用能力跟普通表的能力一样。对于系统租户，使用__all_core_table 的 leader 作为 GTS 服务的提供者，高可用能力与普通表一样。

GTS 维护了全局递增的时间戳服务，在有主改选和无主选举场景下依然能够保证正确性。有主改选是指原 Leader 主动发起改选的场景，我们称为有主改选。无主选举是指原 leader 与多数派成员发生网络隔离，等 lease 过期之后，原 follower 会重新选主，这一个过程，我们称为无主选举。

GTS 可以通过语句快照获取优化，事务提交的时候都会更新其所在机器的 Global Committed Version，当一条语句可以明确其查询所在机器时，如果是一台机器，则直接使用该机器的 Global Committed Version 作为 Read Version，降低对于全局时间戳的请求压力。GTS 也可以通过事务提交版本号获取优化，多个事务可以合并获取全局时间戳，并且获取时间戳的请求可以提早发送，缩短事务提交时间。

三、OceanBase 事务类型

OceanBase 数据库的事务类型由事务 session 位置和事务涉及的分区 leader 数量两个维度来决定，主要分为单分区事务、单机多分区事务和分布式事务。

1.单分区事务

本地单分区事务需要满足以下两个条件：一是事务涉及的操作总共涉及一个分区。二是分区的 Leader 与 Session 创建的 Server 相同。本地单分区事务是最简单的模型，事务的提交采用了极高的优化。

2.单机多分区事务

单机多分区事务也需要满足两个条件：一是事务涉及的表操作所涉及的多个分区。二是分区 Leader 与 Session 创建的 Server 相同。

由于 OceanBase 数据库分区级日志流的设计，单机多分区事务本质上也是分布式事务。为了提高单机的性能，OceanBase 数据库对事务内参与者副本分布相同的事务做了比较多的优化，相对于传统两阶段提交，大大提高了单机事务提交的性能。

3.分布式事务：

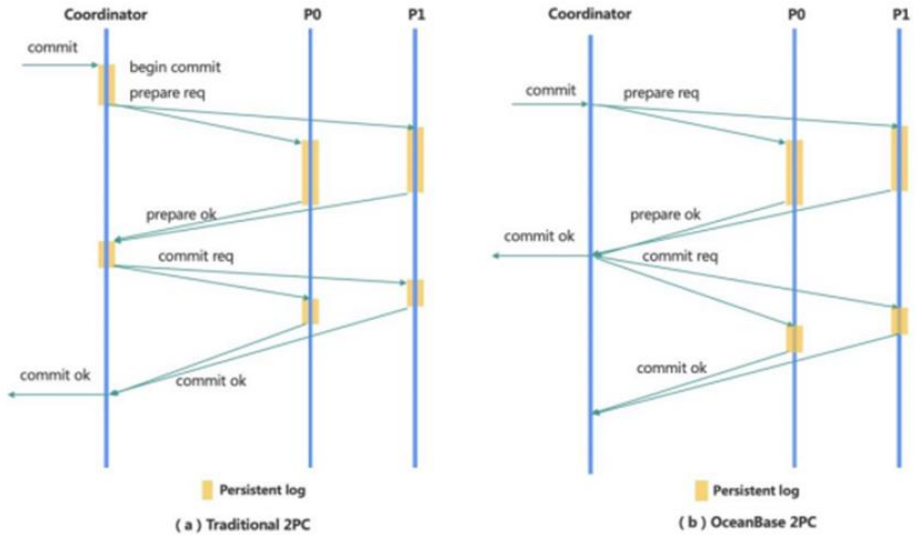
有两种场景的事务被称为分布式事务：一是事务涉及的分区数量大于一个，二是事务涉及的分区数量只有一个，且分区 leader 和事务 session 位置不在同一个 server。

分布式事务满足事务的所有属性，同样需要满足 ACID 的特性。在多机数据修改，且要保证原子性的场景下，分布式事务能够发挥重要作用。为了保证上述特性，通常采用两阶段提交协议。

4. 两阶段提交协议

OceanBase 数据库实现了原生的两阶段提交协议，保证了分布式事务的原子性。两阶段提交协议中包含两种角色，分别是协调者(Coordinator)和参与者(Participant)。协调者负责整个协议的推进，使得多个参与者最终达到一致的决议；参与者响应协调者的请求，根据协调者请求完成 prepare 操作及 commit 或者 abort 操作。

OceanBase 数据库两阶段提交的流程如下图所示：



在 PREPARE 阶段中，协调者会向所有的参与者发起 prepare request，参与者在收到 prepare request 之后，决定是否可以提交，如果可以则持久化 prepare log 并且向协调者返回 prepare 成功，否则返回 prepare 失败。

在 COMMIT 阶段中，协调者会在收集齐所有参与者的 prepare ack 之后，进入 COMMIT 状态，向用户返回事务 commit 成功，然后向所有参与者发送事务 commit request。参与者在收到 commit request 之后释放资源解行锁，然后提交 commit log，日志持久化完成之后给协调者回复 commit ok 消息，最后释放事务上下文并退出。、

并发控制

一、并发控制模型

每个事务包含多个读写操作，操作对象为数据库内部的不同数据。最简单的并发控制就是串行（serial）执行，指一个进程在另一个进程执行完一个操作前不会触发下一个操作。但这明显不符合高并发的需求。因此学者们提出了一种可串行化（serializable），即可以通过并行（非串行）地执行事务内的多个操作，但是可以达到和串行执行相同的结果。

我们可以利用事务中的读写操作来为事务来建立依赖关系（依赖关系代表事务串行化成串行执行序后的事务定序，若事务 B 依赖事务 A，事务 A 应该排在事务 B 前面）：

- 写写冲突（Write Dependency）：当事务 A 修改数据 X 后，事务 B 再修改同一数据 X，事务 B 依赖事务 A。
- 写读冲突（Read Dependency）：当事务 A 读取数据 X 后，若数据 X 对应是由事务 B 修改的，事务 A 依赖 事务 B。
- 读写冲突（Anti Dependency）：当事务 A 读取数据 X 后，事务 B 再修改了同一数据 X，事务 B 依赖事务 A。

通过冲突来定义的可串行化，一般称为冲突可串行化（conflict serializable），可以轻易地通过以上的冲突机制来分析。当事务间的冲突关系没有成环的话，就可以保证冲突可串行化。有两种常见的实现机制，即**两阶段锁**和**乐观锁**机制。前者是通过排它地通过加锁限制其他的事务的冲突修改，并通过死锁检测机制回滚产生循环的事务，保证无环；后者通过在提交时的检测阶段，回滚所有可能会导致成环的事务保证不会产生环。

锁的类型

数据库的锁包括锁(Lock)和闩锁(Latch)两种类别，闩锁是一种轻量化的锁，它通过算法控制应用程序持不同锁（互斥锁、共享锁）的顺序保证无死锁的发生。

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥锁
死锁	通过waits-for graph、time out等机制进行死锁检测与处理	无死锁检测机制。仅通过应用程序枷锁的顺序（lock leveling）保证无死锁发生
存在于	Lock Manager的哈希表中	每个数据结构的对象中

两阶段锁(Two-Phase Locking, 2PL)是一种悲观锁机制，该机制认为在事务执行的过程中一定会遭到并发访问，产生数据竞争，它会使用上面提到的锁(Lock)，在事务执行的过程中锁定数据库内容。这种机制的优点在于安全，缺点在于限制了数据库的并行性，增加死锁的机会，为了处理锁的问题会产生额外的开销。2PL 对锁的管理策略是 Growing-Shrinking，在 Growing 阶段，数据库只申请锁，或者拒绝锁的申请，在 Shrinking 阶段，数据库只进行锁的释放，第一个锁释放后就不再进行任何锁的申请。2PL 足以保证冲突可串行化，但是它容易受级联中止的影响，也有死锁的风险。

强严格两阶段锁（Strong Strict Two-Phase locking）是在 2PL 的基础上，要求一个事务写入的值在提交前不会被其他事务访问，也就是事务只在提交的时候才释放锁，这可以有效防止级联中止的影响，但这是一种更悲观的调度，进一步降低了数据库的并发性。

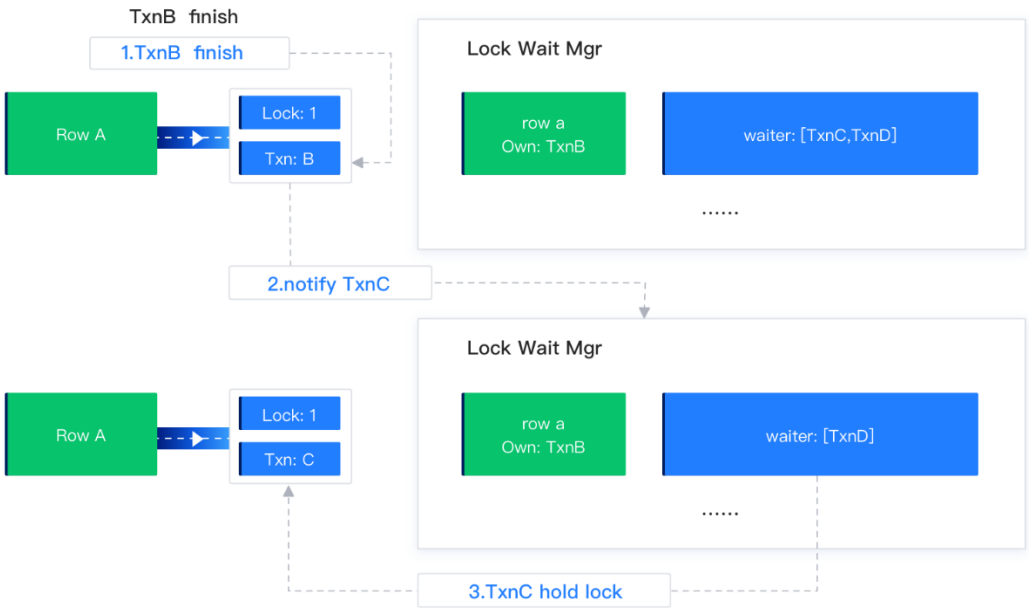
关于死锁问题，存在死锁检测和死锁预防的处理策略。死锁检测方面，DBMS 创建一个事务-资源等待图（有向图），定期检查图中是否产生循环，如果产生循环则选择受害者事务将其回滚，选择的依据常常是事务的时间戳、执行进度、持锁数、回滚的次数（防饿死）等，选中后可以局部回滚也可以全部回滚。死锁预防方面，是在一个事务试图获取另一事务已持有的锁时发挥作用，存在 wait-die 和 wound-wait 两种方案。wait-die 是一种非剥夺策略，当老事务试图申请新事务的锁时会进行等待，而如果新事务试图获取老事务的锁则会被回滚。wound-wait 是一种剥夺策略，当老事务试图申请新事务的锁时会杀死新事务以抢占自身需要的资源，而如果新事务试图获取老事务的锁则会进行等待。

OceanBase 锁机制

OceanBase 数据库使用了多版本两阶段锁，事务的修改每次并不是原地修改，而是产生新的版本。因此读取可以通过一致性快照获取旧版本的数据，因而不需要行锁依旧可以维护对应的并发控制能力，因此能做到执行中的读写不互斥，这极大提升了 OceanBase 数据库的并发能力。比较特殊的是 SELECT ... FOR UPDATE，此类执行依旧会加上行锁，并与修改或 SELECT ... FOR UPDATE 产生互斥与等待。而修改操作则会与所有需要获取行锁的操作产生互斥。

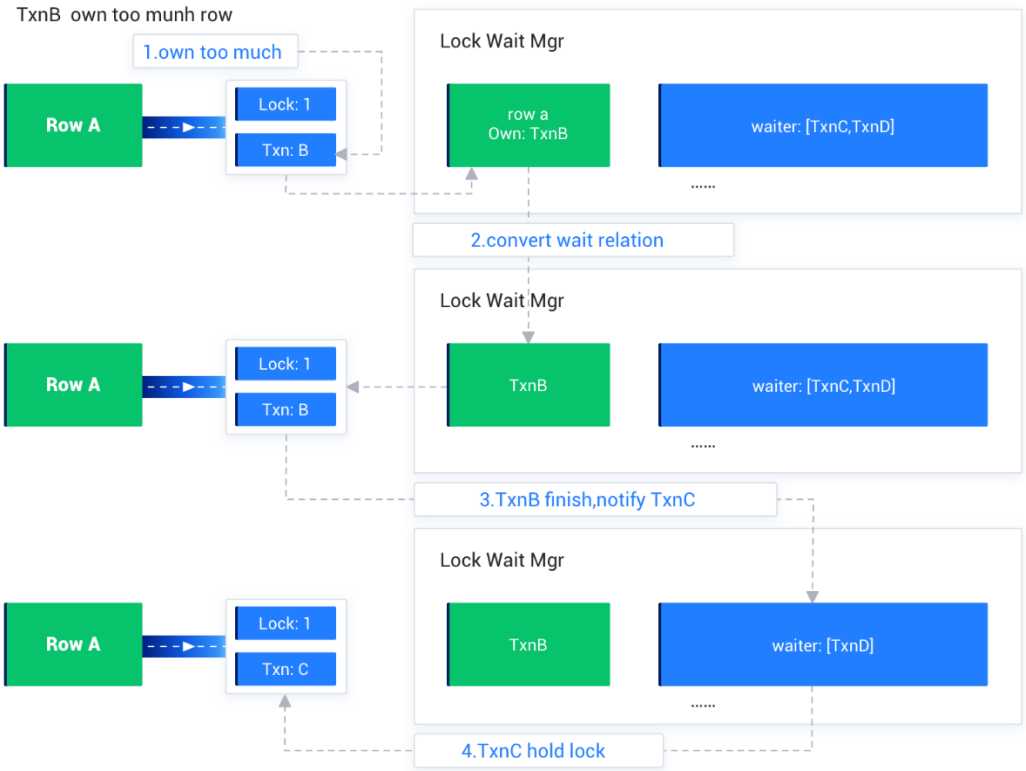
类似于大部分的两阶段锁实现，OceanBase 数据库的锁在事务结束（提交或回滚）的时候释放的，从而避免数据不一致性的影响。OceanBase 数据库还存在其余的释放时机，即 SAVEPOINT，当用户选择回滚至 SAVEPOINT 后，事务内部会将 SAVEPOINT 及之后所有涉及数据的行锁，全部根据**上文的互质机制**中介绍的机制进行释放。

关于锁机制的唤醒为了唤醒事务，当产生互斥后，会在内存中维护行与事务的等待关系，如图所示，行 A 被事务 B 持有，被事务 C 与事务 D 等待。此等待关系的维护，是为了行锁释放的时候可以唤醒对应的事务 C 与 D。当事务 B 释放行 A 后，会根据顺序唤醒事务 C，并依赖事务 C 唤醒事务 D。



除了行与事务的等待关系，OceanBase 数据库可能会维护事务与事务的等待关系。为了减小对于内存的占用，OceanBase 数据库内部可能会将行与事务的等待关系转换为事务与事务的等待关系。如图所示，行 A 被事务 B 持有，被事务 C 与事务 D 等待，

并被转换为事务 B 被事务 C 与事务 D 等待。当事务 B 结束后，由于不明确知道行之间的锁等待关系，会同时唤醒事务 C 与事务 D。



在锁机制的粒度方面，OceanBase 数据库现在不支持表锁，只支持行锁，且只存在互斥行锁。传统数据库中的表锁主要是用来实现一些较为复杂的 DDL 操作，在 OceanBase 数据库中，还未支持一些极度依赖表锁的复杂 DDL，而其余 DDL 通过在线 DDL 变更来实现。

多版本并发控制(MVCC)

多版本并发控制指的是 DBMS 在数据库中维护单个逻辑对象的多个物理版本，事务进行写入时，DBMS 创建该对象的新版本，事务读取时，会读取事务启动时存在的最新版本。

这样做的好处是，作者和读者不会互相屏蔽，一个事务在修改对象的同时，其他事务可以读取旧版本，只读事务可以读取数据库的快照，而无需使用任何类型的锁。此外，MVCC 支持 time-travel queries，即可以像数个小时之前一样对过去的数据库进行查询。

多版本数据&事务表

为了支持读写不互斥，OceanBase 数据库从设计开始就选择了多版本作为存储，并让事务对于全局来说，维护两个版本号，读版本号和提交版本号，如图所示的本地最大读时间戳和最大提交时间戳。其次，在内存中会为每一次更新记录一个新的版本（可以做到读写不互斥）。

如图所示，在内存中有三行数据 A、B 和 C；其中每次通过版本（ts）、值（val）

和事务 id（txn）来维护更新，并将多次更新同时维护来保持多版本；其次，内存中存在一个事务表，事务表中记录了每个事务的 id、状态以及版本。事务开始和提交时会通过全局时间戳缓存服务（Global Timestamp Cache）获取时间戳作为读时间戳和作为提交时间戳参考的一部分。

恢复

恢复的定义：

数据库中的恢复是指在数据库系统发生故障或异常时，通过使用备份数据和日志文件等手段，将数据库从错误状态恢复到某一已知的正确状态，也称为一致性状态或完整状态。恢复的目的是保证数据库的安全性和完整性，以及事务的原子性和持久性。

恢复的特性：

（1）冗余：通过建立数据的副本来防止数据的丢失或损坏。

（2)数据转储和登记日志文件:恢复的实现技术主要有数据转储和登记日志文件两种。数据转储是指定期将数据库中的数据复制到另一个存储介质上,以便在发生故障时从备份数据中恢复；登记日志文件是指记录数据库中所有修改操作的日志，以便在发生故障时根据日志中的信息进行恢复。

（3）恢复策略：根据不同的故障类型，可以分为事务撤消、重做、系统故障恢复、介质故障恢复等。事务撤消是指在事务执行过程中发生故障或异常时，将事务所做的修改回滚到事务开始前的状态，以保证事务的原子性；重做是指在事务提交后发生故障或异常时，将事务所做的修改重新应用到数据库中，以保证事务的持久性；系统故障恢复是指在数据库系统崩溃或停机时，通过使用日志文件和检查点机制，将数据库恢复到最近的一致状态；介质故障恢复是指在数据库存储介质损坏或丢失时，通过使用数据转储和日志文件，将数据库恢复到最近的一致状态。

（4）恢复过程：通常包括分析、重做和撤消三个阶段。分析阶段是指系统分析日志文件，以确定发生故障时处于活动状态的事务和需要恢复的脏页；重做阶段是指系统通过重新执行日志文件中记录的操作，将数据库中的数据前滚到故障发生时的状态；撤消阶段是指系统通过执行日志文件中记录的撤消操作，将未提交的事务所做的修改回滚，以保证数据库的一致性。

恢复的重要性：

（1）保证数据库的安全性和完整性：恢复可以从各种故障中恢复数据库，如硬件故障、软件错误、操作员失误或恶意破坏等。这些故障可能导致数据库中的数据丢失、损坏或不一致，从而影响数据库的正常运行和业务的正常进行。恢复的目的是通过使用备份数据和日志文件等手段，将数据库从错误状态恢复到某一已知的正确状态，也称为一致性状态或完整状态。恢复的过程可以分为事务撤消、重做、系统故障恢复、介质故障恢复等，根据不同的故障类型和恢复需求采用不同的策略和技术。

（2）保证事务的原子性和持久性：恢复可以确保事务的修改是持久的，并且可以在发生故障时进行恢复。事务是数据库中的一个逻辑工作单元，它包含了一系列对数据库的操作，要么全部成功执行，要么全部失败回滚，不允许部分执行。事务的原子性是指事务的不可分割性，事务的持久性是指事务的修改在提交后不会丢失。恢复的原理是利用预写式日志，即在修改数据库之前先将修改记录在日志中，然后根据日志中的信息进行恢复。如果事务在执行过程中发生故障或异常，恢复可以通过撤消操作将事务所做的修改回滚到事务开始前的状态，以保证事务的原子性。如果事务在提交后发生故障或异常，恢复可以通过重做操作将事务所做的修改重新应用到数据库中，以保证事务的持久性。

（3）提高数据库的可用性和性能：恢复还可以通过使用检查点和数据库镜像等技术提高数据库的可用性和性能。数据库的可用性是指数据库能够在任何时候为用户提供服务的能力，数据库的性能是指数据库能够快速响应用户请求的能力。恢复的过程通常需要一定的时间，期间数据库可能无法为用户提供服务，或者服务质量下降，因此需要尽量缩短恢复的时间，减少恢复的开销。检查点是指定期将缓冲区中的脏页刷新到磁盘，以减少恢复时需要重做的日志量。数据库镜像是指将数据库的数据实时同步到另一个备用数据库，以便在主数据库发生故障时切换到备用数据库，实现数据库的高可用性。