NewSQL研究

论文翻译：https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2777732/no.ntnu%3ainspera%3a57320302%3a31535683.pdf?sequence=1&isAllowed=y

一、摘要

在我们研究期间，我们发现数据库的性能是可比较的，但是它们有不同的权衡，CockroachDB和serializable YugabyteDB比TiDB和snapshot YugabyteDB提供更高层的事务隔离，但是在写操作上，它们有耕地的吞吐和高延迟。不管怎样，CockroachDB在读上超出其他系统，这意味着事务隔离级别权衡只是影响写操作。

二、介绍

在过去的的20年中，随着产生和处理的数据量的大量增长，比传统数据库系统更好的扩展性的新数据库已经产生。许多这些系统牺牲一致性和隔离性保障以达到更好扩展的目的，经常将这类系统归类为NoSQL。尽管这类系统比传统数据库提供更好的扩展性，它们的弱保证意味着，应用需要负责处理不一致。对一些应用，如金融系统，这点需求使这些数据库变得无法使用。

作为这些挑战的回应，需要研究已经开始进入一个新型数据库系统，它能够很好的扩展性同时，又能保障隔离级别和事务一致性。这些数据库通常称作NewSQL，并且提供带有传统ACID保障的SQL接口，又具备容错性和可扩展性。这些系统中许多是受到Google’s Percolator和Spanner架构的启发，同时另一种新的架构事由Calvin引入。

三个流行的开源NewSQL数据库是CockroachDB，TiDB，YugabyteDB。这些是受Google Spanner和Google Percolator的启发，采用不同的方法去实现可扩展性和处理事务。这三个数据库都能够很好的在云上工作，并且高度关注提供横向的扩展性，意味着应用在使用这些数据库时，不再被单机的性能所限制。但，没有这三个系统的深度的比较研究，不清楚每个系统的性能权衡是什么。在这个理论下，我们想要解决这个问题，尽可能通过研究这些数据库去揭开架构和性能的不同。

三、理论

1、CAP理论

CAP理论，由 Eric Brewer创造，声明一个分布式系统不能同时提供一致性，可用性和分区容错性。这个意味着任何分布式系统必须选择这些属性中的两个。由于网络是不可信的，任何分布式系统必须支持分区容错性，因为在一个网络中的分布式系统选择要么是一致性的（CP），要么是可用性的（AP）。一个CP系统保持完全一致性在网络隔断期间，但是不完全可用。通常这意味着更小的隔断或

者有时整个数据库变得不可用在隔断期间。一个AP系统选择一致保持可用性，以一致性为代价。在一个网络隔断期间，不同的隔断区或许不在一个一致的壮腰，在这个案例中，其中一个状态需要被选择，而另一个是被抛弃的。

CAP理论声明在分布式系统遇到隔断期间一个非常简单的权衡事实，但没有描述在一个网络健康时候的权衡。为了处理这个，Daniel J. Abadi创造一个新的理论，叫做PACELC。这个理论声明，在一个隔断期间，系统必须选择可用性或一致性，但与此不同，它必须优先考虑低延迟或一致性。这意味着，为了在网络正常期间实现更低的延迟，一个分布式数据库必须牺牲一些一致性保证。

2、ACID

2.1、Atomicity

2.2、Consistency

2.3、Isolation

2.4、Durability

3、Isolation levels

完美的隔离级别，著名的是strict serializability，对性能有重要的影响，在普通数据库中使用低级别的隔离级别。为了理解不同隔离级别的不同，一组常见的异常已经被定义，并在下面列举。异常是发生在一个事务能够注意到它不是唯一在跑或没串行的运行的事件。strict serializability不允许一下的任何异常发生。

- Lost update：如果两个事务读取同一个key，然后进行写入，则会发生更新丢失。如果两个事务（例如，同时递减一个字段），它们可能会在无意识的情况下相互覆盖，导致两个事务都提交，但值仅递减一次。这是一种隔离异常，因为顺序执行将导致与此并发执行不同的状态。
- Dirty write：当一个事务读取一个另一个事务已经写入但没提交的值时，会发生脏写。如果第二个事务随后基于第一个事务执行一些操作，并且第一个事务在第二个提交时中止，则数据库状态可能会变得不一致。以上面例子为例，如果事务A递减一个值，B随后递减该值，但事务A中止，B提交，则你现在有一个事务提交，但是值被递减了2。在这种情况下，事务的执行顺序也会导致不同的的状态，因为第二个事务永远不会看到第一个事务被中止。
- Dirty read: 两个事务，A和B，正在并行的执行，事务A已经写入key K1和K2，如果事务B现在读到K1，并看到A执行的写入，但随后读取key K2，但没有看到A执行的写入，这种异常被视为脏读。因为B只看到A执行的一部分，因为不能考虑在A之前或之后执行，结果与串行执行不同，因为A或者B将需要在顺序执行中首先执行。
- Non-repeatable read: 如果一个事务读取相同的记录两次，但是第一次和第二次读到的值不同，然后该事务在运行时观察到了另一个事务的结果。这与顺序执行的结果不同，因为除了正在运行的事务之外，不应该有任何其他内容更改数据。

- Phantom read（幻读）：幻读和不可重复读有些类似，读取时候返回不同的值。然而，对于幻读，每个键返回相同的值，但扫面返回不同的结果集。如果另一个事务插入与该事务执行的扫描匹配的行，则可能发生这种情况。
- Write skew:当两个事务读取相同的key，但随后基于此数据写入不同的key，会发生写偏离。例如，如果两个事务都读取key K1=5和K2=4，第一个事务将K1值设置为K2，也就是4，而第二个事务将K2设置为K1的值，也就是5。这两个事务的目的都是使两个key有相同的值，但写入偏离意味着这两个值现在交换，而任何顺序执行将意味着这两个值相等。

当说到隔离级别时，经常讨论的6种异常。隔离级别被标准定义为：READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ 和 SERIALIZABLE。另一种在SQL标准中没有定义的是Snapshot Isolation，提供一个数据库能够读取的快照，并且只允许Write skew异常。

4、Distributed concurrency control

并发控制是在需要数据库系统状态一致情况下，允许多个事务同时运行。并发控制是一个很好的研究领域，并在任何允许多发事务发生的数据存储中都需要。然而，在一个分布式系统中，并发控制的选择变得非常重要，因为每个线程不再跑在相同的机器上，而是在具有显著网络延迟的网络上运行。每种策略都有其优点，根据工作负载特性，选择不同策略是最好的。并发控制策略被使用最重要的因素是事务之间冲突的频率，因为一些策略是擅长处理冲突，但是以牺牲每个事务的性能作为代价。下面，我们描述四种常见的并发控制算法。

4.1、Two-Phase Commit

两阶段锁（2PL）是一个被证明可串行化的方法。两阶段锁在事务的生命周期中引入了两个锁阶段：锁的获取和锁的释放。事务在执行期间能请求新的锁，当不再需要时，释放它们。然而，需要警告的是，在任何锁被释放之后，都不允许再获取更多的锁，这就是为什么该算法被称为两阶段。

在2PL中有两个锁类型，一种是读取，另一种是写入。读锁可以与其他读锁共享，但写锁不能与其他读锁或写锁共享，即写锁需要独占访问数据项。如果为具有其他锁的数据项请求写锁，则事务需要等待。这种等待可能导致死锁，因为事务可能有循环等待依赖性。这需要通过中止一个事务来处理，解决死锁问题的另一种方法是在需要等待锁的情况下终止任何事务，然后重试整个事务。如果完全重试是高成本的，这或许代价很大，但它可以在没有使用复杂的算法情况下去解决死锁问题。

4.2、Optimistic

乐观的并发控制系统执行事务不带任何锁。但，在事务提交之前，它会执行一项检查，以检查它读取的任何数据是否被其他事务修改。如果是这种情况，则需要重试事务。如果大多是操作没有冲突，乐观并发控制可以很好的执行，但如果发生许多冲突，则有许多重试，性能会受到很大的影响。

4.3、Multi-version concurrency control

多版本并发控制（MVCC）存储每一个key的多个版本。事务在开始时会被分配一个版本或时间戳，并且只读取更低或相等的版本号。这使实现快照隔离相对容易，但仍存在一些挑战，首先，老得keys需

要被定期清理，以使数据库不会充满陈旧数据；其次，如果系统是分布式，分配单调递增的版本或时间戳会变得复杂。

4.4、Deterministic

在一个确定性的并发控制系统中，事务需要在执行前完全发送到数据库，即不允许交互事务。然后，调度器分析所有事务并将其分配给批次。该调度程序保证批处理中的事务之间不会发生冲突，因此，事务的实际执行可以以很少的开销完成。这种方法的一个缺点是调度器可能成为系统中的瓶颈，并且会引入延迟，因为每个事务都需要在执行前进行批处理。

5、Distributed transactions and consensus

在一个分布式系统中，必须实现分布式共识以保持一致性。这意味着节点必须就事务的排序达成一致。然而，并非所有事务都必须排序，但冲突的事务所在节点上必须具有相同的顺序。

分布式共识可以以多种不同的方式实现，但是两种常见的算法是两阶段提交和Raft。

6、Two-Phase Commit

两阶段提交（2PC），不要与两阶段锁（2PL）混淆，在副本数据库中使用，以确保不同副本是否就是否应提交或中止事务达成一致。这种同步非常重要，否则副本上的上的状态将发生分歧变得不一致。

两阶段提交协议分为准备阶段和提交阶段。在准备阶段，协调器将从所有副本收集关于它们是要提交还是中止的投票。收集所有投票后，提交阶段开始，协调器检查投票。如果任何副本被投票中止，事务将被中止，否则事务将被提交。最后一步是协调器向所有副本广播其决定，以便它们知道事务是中止还是提交的。该协议确保所有副本都同意所有事务的状态，并确保不会出现不一致。

7、Raft

Raft是一种分布式共识算法，被开发为Paxos的更简单、更容易理解的替代方案。该算法允许奇数组节点就日志的内容和顺序达成一致，称为Raft日志。此日志可以包含任何内容，但对于数据库，它通常包含与非分布式数据库使用的传统预写日志类似的记录。

Raft通过定期为Raft小组选举一名组长来工作。每个组还应具有奇数个节点，以便始终保证多数投票。leader是按可配置的时间间隔选出的，通常在每个时间间隔后重新当选，除非他们不可用。如果某个leader不可用，其他节点将宣布选举，并在随机延迟后尝试选举leader角色。新的日志参与只能由raft group的leader添加。然后，领导人需要发起投票，并获得多数票，然后才能认为书面声明已被提交。由于只需要大多数投票，一些节点可能会在日志中落后于领先节点。如果一个leader失败，必须选出一个新的leader，节点需要宣布其最新日志条目号。一个节点永远不会投票给另一个拥有比自己更旧日志的节点，但应该始终投票给拥有更新日志的节点。这种机制与大多数节点具有最新日志相结合，确保了在日志过期的情况下永远不会成为leader。

在Raft之上实现SQL数据库时，读写操作都应向Raft添加日志条目。如果未添加日志条目，在leader变更期间，如果从follower或leader处读取，读取可能会变得不一致。然而，由于这会给读取增加大量开销，许多数据库使用一种称为leadership lease的机制来避免为读取添加日志[21，19，6]。leadership lease由group leader持有一段给定的时间，在此时间段内，不能选择其他节点作为group

leader。这保证了只要leader仍有租约，先导服务的读取将始终具有最新数据。但是，follower仍然不能使用这种机制进行读取，因为他们可能仍然拥有过时的数据。leadership lease解决了读取数据的瓶颈，但如果leader崩溃且租约过长，也可能会导致一段时间的不可用。因此，lease期限通常很短。

8、NewSQL

9、Percolator

2010年，谷歌发布了一个名为Percolator的分布式交易系统[18]。它支持并发事务并实现多版本并发控制。每个事务都可以看到数据库与MVCC的一致状态，并在事务之间提供快照隔离。Percolator不是一个NewSQL系统，因为它只支持简单的键值操作，但其设计已在TiDB中用于实现分布式SQL数据库。

Percolator使用时间戳“oracle”，以便为每个事务获得唯一且单调增加的版本号。oracle本质上是一个集中式系统，所有事务都必须通过该系统才能获得其读写操作的版本号。这意味着不依赖于同步的时钟，但如果集群超出了中心时间戳oracle，这也可能成为性能瓶颈。

10、Spanner

Spaner[7]是Google在2013年发布的一个NewSQL系统。Spaner通过使用MVCC实现并发分布式事务，并使用时间戳为事务分配唯一的版本号。然而，为了克服时钟不同步的问题，spanner引入了“TrueTimeAPI”，这是一种精确的时钟，可以返回当前时间和不确定性。spanner使用此时间戳作为事务的版本号，但也添加了额外的逻辑来处理时间戳的不确定性。例如，如果读取可能在事务之前或之后写入的key，则需要使用新的时间戳重试事务。通过该系统，spanner能够提供称为外部一致性的高级别一致性，这比序列化更严格，但不是最严格的形式。

TrueTime API基于原子钟和GPS时钟来获得准确的结果，通常可以将不确定性保持在10ms以下。较低的不确定性总是有利于性能，因为事务需要重试的概率较低。

Spanner为底层数据存储使用键值接口，并使用Paxos算法进行复制。如前所述，并发控制由MVCC处理，事务使用两阶段提交策略来确保一致性。

11、Calivn

Calvin[22]是一个利用确定性并发控制形式的数据库。这意味着所有事务都被放入不会发生冲突的批中，这可以显著加快事务处理速度。Calvin通过一个集中式调度程序来实现这一点，该调度程序接受所有事务并对其进行批处理。

尽管这种方法可以显著提高速度，但也有一些缺点。首先，调度器可能在一定规模上成为瓶颈。其次，所谓的依赖事务很难执行。由于调度器需要知道事务访问的所有keys，因此执行基于查询的操作（如外键查找）具有挑战性。Calvin建议，在这种情况下，应首先执行只读查询以找出实际的键，然后在实际事务中使用这些键。为了避免不一致，实际事务必须检查预取的值在此期间没有变化。如果值已更改，则需要重试事务。然而，作者认为这在大多数系统中不是一个重要问题，因为像外键这样的数据在大多数应用程序中很少发生变化。

四、CockroachDB

CockroachDB是由Cockroach Labs开发的NewSQL数据库，于2017年首次发布。CockroachDB的设计灵感来自谷歌Spanner，但与Spanner相比的一个显著差异是它不使用任何专用硬件时钟。相反，使用了**混合逻辑时钟，可以模拟TrueTime API的功能**。然而，这意味着CockroachDB的不确定性要比SPanner大得多，在时钟异常的情况下，有可能在数据库中引入不一致性。为了减轻损害，CockroachDB具有在时钟不同步的情况下停止系统的机制。CockroachDB支持**可序列化事务隔离，但不是最强大的事务隔离形式**。

CockroachDB实验室还销售Cockroach Cloud，在那里他们销售CockroachDB集群的托管产品。CockroachDB已获得许可，因此它可以用于除销售“CockroachDB作为服务”之外的任何用途。所有源代码在发布三年后也将免费发布，这意味着三年前的代码可以作为服务托管和出售[14]。除此之外，还有一个企业版的CockroachDB，它添加了一些额外的功能。

我们对CockroachDB的描述是基于他们的官方在线文件[6]。在下面的小节中，我们将描述CockroachDB的体系结构和存储引擎，以及它如何处理事务和地理复制。

1、Architecture

CockroachDB是一个同性质的节点集群，这意味着所有节点运行相同的软件，集群中没有特殊的主角色。因此，所有节点都从客户机接收请求并处理事务，而不需要客户机了解数据放置位置。

CockroachDB软件分为五个层，每个层之上构建。所有层都在同一个程序中运行，但这些层对于系统的推理非常有用。下面列出了五个层：

1、SQL layer: SQL层是客户端与之通信的层。它接收SQL查询并将其转换为键值操作，然后发送到下一层。因此，没有其他层知道SQL，而是使用键值数据进行操作。

2、Transactional layer: 该层协调事务并为数据库提供ACID支持。并发控制在这一层处理。

3、Distribution layer: 分布层将键值空间的分布特性抽象到更高层。更高层不需要知道数据的位置。相反，请求被路由到该层中的正确对等方。

4、Replication layer: 复制层复制每个分片的数据，确保每个分片一致且对等点同步。这是通过使用Raft算法实现的。

5、Storage layer: 存储层是从磁盘读写的层。如下文所述，每个节点可能是多个Raft组（即碎片）的成员，但一个节点上的所有数据都由单个存储层处理。

2、Storage engine

为了更好地比较不同的NewSQL数据库，我们将存储引擎定义为SQL层以下的所有内容。因此，对于CockroachDB，我们所称的存储引擎由事务层、分发层、复制层和存储层组成。

CockroachDB将数据分成“范围”，每个范围大约64MB大。由于分片非常小，任何节点通常都会承载许多不同的分片。所有分片都复制到可配置数量的节点，但默认为三个。同步处理复制以实现一致

性，并使用Raft算法执行复制。CockroachDB还使用Raft实现leader lease，如第2.7节所述，以提高每个分片的读取性能。

CockroachDB的存储层构建在RocksDB之上，这是一个关键值存储系统，可将数据持久存储在磁盘上，并具有高性能。RocksDB支持标准的键值操作，并可高度配置以满足特殊需求。CockroachDB使用RocksDB作为MVCC存储，其中每个key可以有多个时间戳版本。为了获得key的最新版本，可以使用前缀按相反顺序执行扫描，其中第一个结果是最新版本。RocksDB高度依赖bloom过滤器来快速确定它应该在哪些块中搜索某个键，但这在前缀搜索中不起作用，因为bloom过滤器是用全键构建的。但是，RocksDB还支持前缀bloom过滤器，在生成bloom过滤器时，可以使用用户定义的key前缀。这使得MVCC能够与RocksDB一起使用，而不会造成任何重大性能损失。

RocksDB中CockroachDB使用的另一个特殊功能是快照。RocksDB可以生成数据库状态的一致快照，而不会阻塞其他操作。当一个新副本加入Raft组时，可以生成RocksDB快照并将其传输到新节点，以便它能够快速赶上状态。此外，还使用了一些数据摄取优化，这样就不需要在新节点上执行多个压缩。

3、Transaction handling

如前所述，CockroachDB基于混合逻辑时钟（HLC）使用时间戳来排序事务。节点之间的任何通信还包括其当前HLC时间戳，以便快速识别时钟值中的任何异常。如果时钟偏移量大于最大偏移量，默认情况下为500ms，则可能会出现不一致，因此如果检测到这种情况，CockroachDB会立即关闭节点。

CockroachDB中的事务通过在数据库中存储每个事务的记录来跟踪。该记录用作事务状态的真实性来源，其他事务可以查询该记录，以查看其写入的数据是否已提交。发起事务的节点称为事务协调器，它必须在运行时定期更新事务记录，以表明它没有崩溃。如果事务标记为正在进行，但在某个时间阈值后尚未更新，则所有其他事务都会将其视为已中止，可以删除。

为了区分提交的值和未提交的值，CockroachDB使用了一种称为写入意图的东西，这基本上是临时写入。事务写入的任何数据最初都是作为写意图创建的。写意图看起来类似于普通数据值，但它也包含一个指向事务的事务记录的指针。一旦事务提交，事务记录将标记为已提交，所有写入意图将异步转换为正常的MVCC值。但是，由于这种情况是异步发生的，其他事务可能会对属于已提交事务的写意图进行反向处理，因此该值应视为正常值。任何读取写意图的计数器都必须通过意图中的事务记录指针查找事务的状态。如果事务被中止，则可以删除意图，如果事务被提交，则可以将意图转换为正常值。但是，如果事务仍在进行中，则尝试读取该值的事务必须等待并添加到等待队列中。这种写入意图和事务记录系统支持原子提交，因为事务记录始终是事务状态的真实来源。更改记录会更改整个事务的状态，这是一个单独的分片操作，因为它是单个数据项，因此可以原子地执行。

事务添加到的等待队列存储在单个CockroachDB分片中。这使得检测死锁更加容易，因为所有这些数据都存储在同一硬件上。等待队列中的每个事务都用它正在等待的事务的ID注册，并且每当事务完成时，队列都会收到通知，并且可以恢复由现在完成的事务阻止的任何事务。等待队列中的死锁是通过随机终止死锁循环中的一个事务来处理的。

在创建写意图之前，事务必须检查key的现有版本。如果key的写意图已经存在，则事务必须检查是否仍在进行中，如果仍在进行，事务必须等待。接下来，事务必须检查现有MVCC值的时间戳版本是否高于当前事务的时间戳，如果是，则必须使用新的时间戳重试事务。为确保隔离性和一致性，必须检查

的最后一件事是，最近的读取必须以低于当前事务的时间戳进行。每次读取key时，都会存储读取事务的时间戳，以确保在该时间戳读取的结果以后不会更改。如果写入事务要写入具有更高读取次数的key，CockroachDB会尝试自动将写入事务的时间戳记推高。但是，由于事务的时间戳已更改，事务必须检查其以前的读取是否因新的时间戳而过时，此过程称为读取刷新。

由于CockroachDB的事务排序取决于同步时钟，并且由于时间戳带有一些不确定性，因此每个时间戳都与其他时间戳有一些重叠，无法确定哪一个先到。这导致在处理MVCC时出现一些问题，因为一致性需要绝对排序。为了解决这个问题，CockroachDB使用预定义的最大时钟不确定性，并尝试在发生冲突时将时间戳推到该不确定性之外，或者完全重试事务。例如，如果执行了读取，并且MVCC时间戳非常接近事务的时间戳，则需要将事务的时间戳记推送到一个稍后的点，在该点上它不再不确定首先发生了什么。如前所述，当按下这样的时间戳时，需要进行读取刷新，以确保以前的读取现在不会过时。如果某些键被频繁写入和读取，此功能可以降低操作速度，但也可以防止出现异常。

4、Geo-replication

CockroachDB支持地理复制工作负载，并允许配置数据放置，以便可以针对数据库用户的需求优化数据可用性和延迟。但是，地理复制功能仅在企业许可证下可用，这意味着它们需要花钱，并且不是开源产品的一部分。

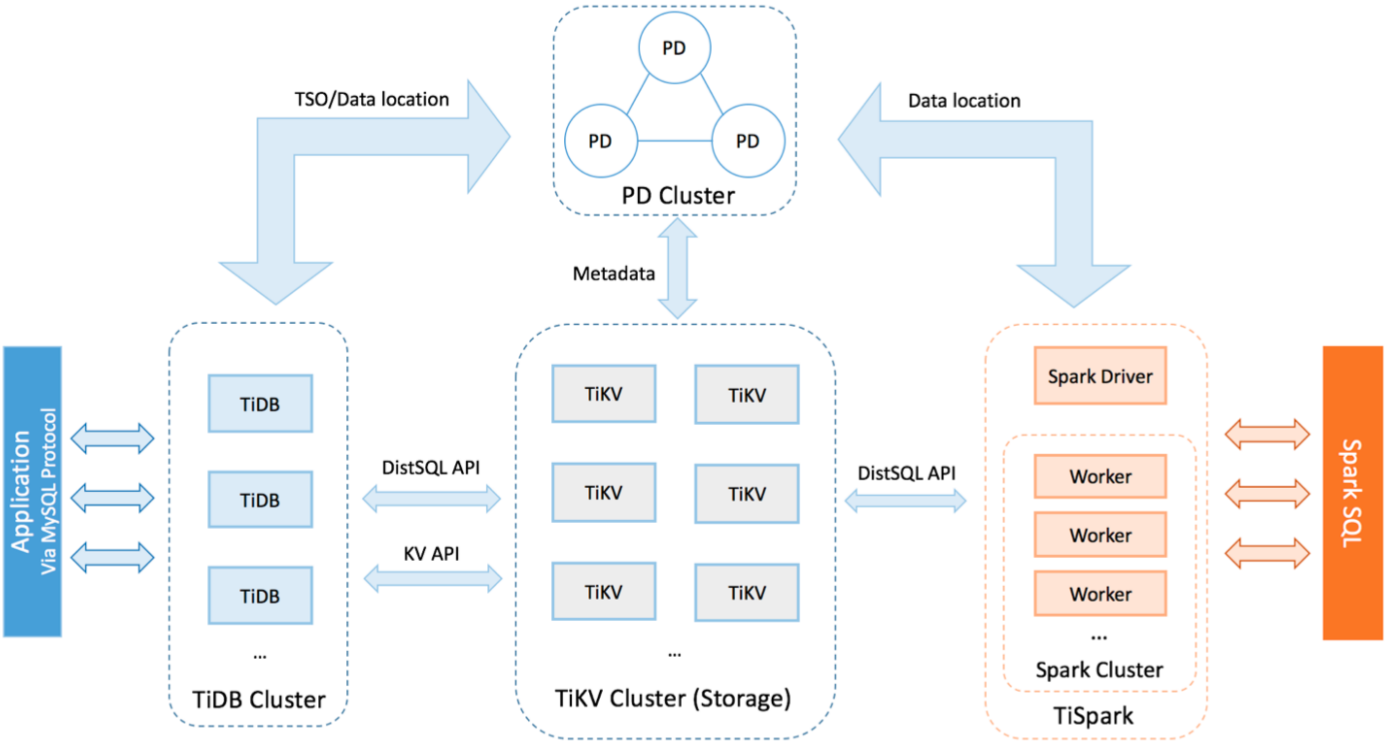
用户在CockroachDB中控制地理复制的主要方法是在每个表上设置分区键，该键将确定表行的位置。此外，分区键可以分配给区域，以便在特定区域中更快地访问数据。有许多不同的数据放置策略，其中之一是将一组分区键的所有副本放置在一个区域中，这样可以在该区域中进行快速读写。但是，这可能以故障容忍度为代价，因为该区域的停机将使所有区域中受影响的行都不可用。另一种策略是将副本放置在靠近最需要它们的位置的相邻区域，以最小化延迟，同时仍具有容错能力。

CockroachDB还允许请求在某个区域安置raft group leader。由于CockroachDB实现leaseholder读取 ，这意味着读取可以在该区域以非常低的延迟进行，而写入仍然需要跨区域通信。如果读速度比写速度更重要，这对于某些应用程序来说是一个很好的折衷。CockroachDB可能采用的另一种方法是启用follower读取，这可以在所有具有副本的区域中实现低延迟读取。然而，这并不保证读取返回最新的数据，如果基于这些读取进行写入，则可能会导致不一致。但是，如果只读事务可以容忍旧数据，并且它们需要非常低的延迟，那么它可能是应用程序的合适功能。

五、TiDB

TiDB是一个基于键值数据库TiKV的NewSQL数据库，于2017年首次发布。TiDB和TiKV均由PingCAP根据Apache 2开源许可证开发。在本节中，我们将TiKV计算为TiDB的一部分，即使它可以独立于TiDB运行。TiKV的设计主要受Percolator的启发，如第2.9节所述。这意味着TiDB使用集中分配的版本号来排序事务，而不是不确定的时间戳。Percolator设计还意味着TiDB只支持快照隔离，这比CockroachDB和YugabyteDB提供的串行化能力弱。

下面，我们将深入描述TiDB和TiKV的相关部分，以深入了解数据库的工作方式。我们的描述基于TiDB和TiKV的在线文档[23,24]。首先，我们描述了集群和软件的体系结构，然后介绍了存储引擎TiKV如何工作和处理事务。最后，我们将介绍TiDB在地理复制场景中的工作方式。



1、Architecture

TiDB集群由三个主要部分组成：时间戳oracle（称为Placement Driver，PD）、键值存储（称为TiKV）和SQL处理系统（称为TiDB）。这三个部分相互作用，形成了一个可以处理SQL事务的NewSQL系统。每个部件都支持复制，通常在单独的机器上运行。在线分析处理（OLAP）查询集群中还可能包括的第四部分是TiSpark。这四个组件如图2.1所示。本章不讨论TiSpark组件，因为它与NewSQL无关。如图所示，NewSQL系统的三个组件PD、TiKV和TiDB必须一起通信，以便为客户端执行SQL查询。

所有SQL客户端都直接或通过负载平衡器与TiDB通信。TiDB负责将用户查询转换为可由TiKV处理的键值操作。TiDB是一种无状态服务，这意味着它可以根据需求快速扩展和缩小。系统中的所有数据都存储在TiKV和放置驱动程序中。

Placement driver充当集群的一种主机，是一个单分片复制系统。PD通过Raft复制，只有leader代表PD集群执行操作。PD跟踪TiKV中的不同分片，并知道哪些TiKV节点负责哪些数据。除此之外，leader还充当时间戳oracle为集群分配时间戳，并为所有事务分配时间戳以进行排序。由于这是由单个系统完成的，因此不需要对所有时钟进行同步。PD在多个节点上复制，但它们作为一个系统运行，这意味着添加节点不会提高PD的性能，而额外的节点只能通过增加更多的故障容限来提供帮助。为了确保在leader崩溃时不分配两次时间戳，领导者必须在分配时间戳之前通过提交Raft日志来保留一个时间戳块，这确保了下一个leader不能在同一个块中分配任何时间戳。

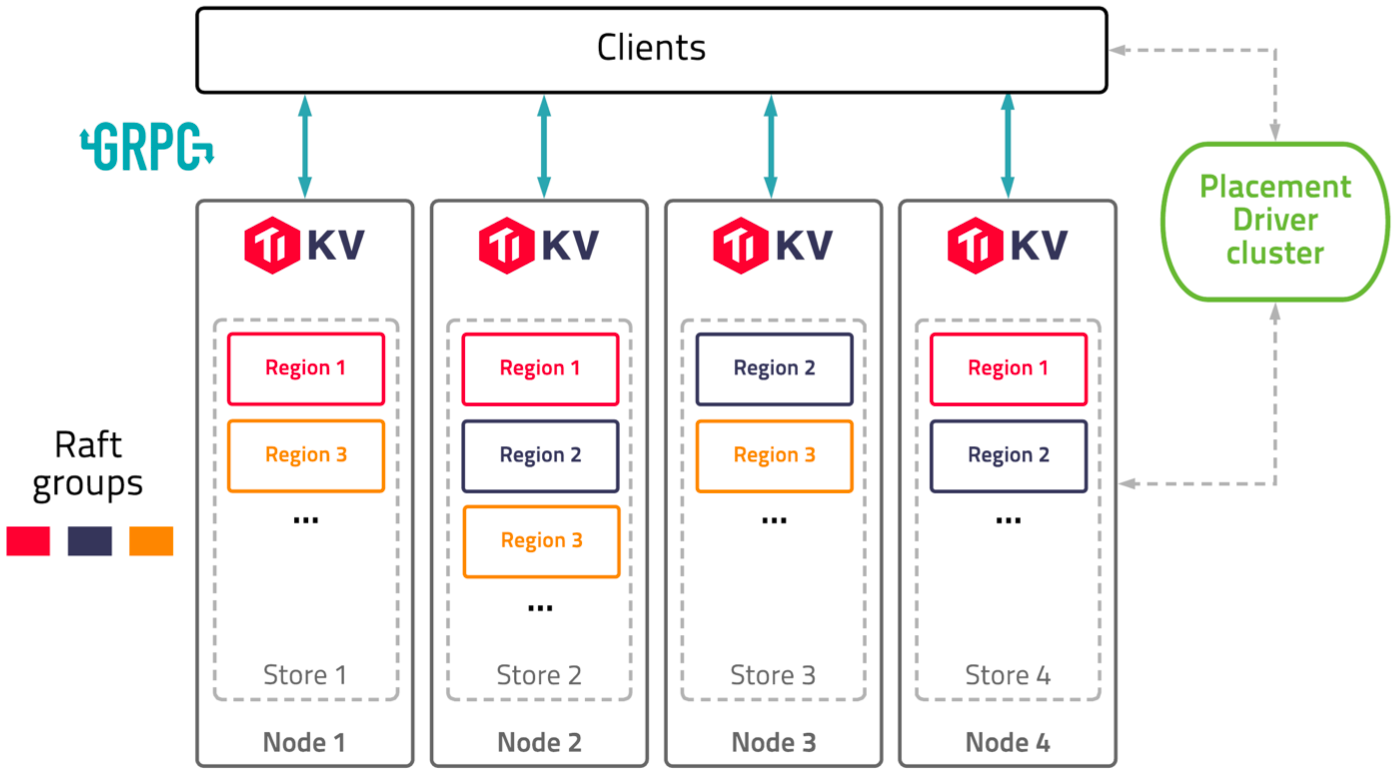
TiKV是TiDB的键值接口，它还具有完全的事务支持。TiKV的设计将在下面的存储引擎一节中进一步描述。TiKV主要响应来自TiDB的请求，但它也可以与希望使用键值接口而不是SQL接口的客户端通信。TiKV节点还与PD集群通信，例如，为了知道它们应该为哪些分片服务。由于在TiKV节点之间均匀分布，因此将TiKV节点添加到集群将提高大多数工作负载的性能，因为有更多节点可以共享工作。

2、Storage engine

如前一节所述，TiKV是TiDB的存储引擎。TiKV的灵感来自Google Percolator，它支持使用快照隔离的事务。

TiKV使用Raft同步复制数据，默认情况下，将数据复制到三个节点。键值空间被范围划分为“区域”，其中每个区域通常都很小，这样可以在节点之间快速移动区域。由于区域较小，群集中的每个节点通常为其中许多区域提供服务。图2.2显示了Raft组、区域和节点如何相互作用。该图显示了一个集群，其中数据分为三个区域，分布在四个节点之间。没有节点存储每个组的多个副本，因为这将减少冗余。当使用TiDB作为NewSQL系统时，图顶部的客户机将是将客户机SQL请求转换为键值请求的TiDB节点。

为了将键值数据存储到磁盘，TiKV使用RocksDB。这一选择是基于该技术的高性能和成熟度[24]。与CockroachDB类似，TiKV还将RocksDB用作MVCC存储，因此需要前缀bloom过滤器支持。TiKV利用的另一个RocksDB特性是多列族支持，这基本上意味着RocksDB实例上存储了多个数据库，并且支持跨这些数据库的原子写入。



3、Transaction handling

TiDB中的所有事务处理都发生在TiKV内部，因为TiDB是数据库中的无状态层。TiKV实现了Percolator的并发控制机制，因此支持事务之间的快照隔离。但是，如果需要避免写倾斜异常，可以使用一种特殊的SQL语法：select..for update。与Percolator一样，TiKV中的交易顺序取决于中央时间戳oracle分配的时间戳，在TiKV的情况下，中央时间戳是PD。

TiKV中的每个键都有三个单独的列，分别包含数据、锁和写值。为了支持这三个单独的值，TiKV为每种值类型使用一个RocksDB列族。同一个键在每个列族中可能具有不同的值，从而启用Percolator模型所需的多值功能。每个键的数据列包含数据的多个版本，但不包含有关是否提交此数据的任何信息。lock列存储行的单个锁，最后，write列包含键的提交，其中每个提交都存储有时间戳，并指向数据列中的一个值。MVCC key的时间戳以这样的方式编码，即最新版本将始终是在扫描操作中出现的第一个版本。

当TiKV中的事务启动时，它向timestamp oracle请求时间戳。对于写事务，下一步是获取它要写入的所有数据项的锁。如果任何键的时间戳高于当前事务的值，或者锁已经存在，则事务需要释放其锁，并使用稍后的时间戳重试。事务创建的第一个锁被指定为primary key，所有其他锁都包含指向主锁的指针。创建每个锁后，还将向key的数据列添加一个数据值。以这种方式执行所有写操作后，事务可以通过更新主锁进行提交，同时在相应的写列中创建一个条目。主锁是事务状态的真实来源，如果遇到次锁，其他事务可以使用主锁。删除主锁后，所有次锁都可以转换为写值，但这是异步完成的。

当事务想要读取值时，它使用从时间戳oracle接收的时间戳。然后必须首先检查是否存在时间戳低于当前事务的锁。如果没有锁，它可以从时间戳低于当前事务的写入列中获取最新版本，然后读取写入指向的值。但是，如果密钥上有锁，则读取事务必须检查写入事务的状态。这是通过查找主锁来完成的，如果它仍然处于活动状态，则需要等待或重试读取事务。如果主锁不存在，则必须确定事务是中止还是提交的。这可以通过检查write列中是否有对应于主锁的值来实现。如果存在写入，则认为事务已提交，如果没有，则认为已中止。这种处理二级锁的机制确保提交是原子的，因为更改主锁的操作可以保证整个事务被提交或中止

4、Geo-replication

TiDB对地理复制有一些支持，但其设计中的一个大问题是，TiDB高度依赖于集中式PD。只有一个PD副本可以向事务分发时间戳，因此，从不同于PD领导者所在区域启动的任何事务都将获得高延迟。但是，将PD复制到多个区域将提高系统的故障容限，因为其他区域的副本可以在区域停机时接管。

五、YugabyteDB

由Yugabyte开发的Yugabybyte数据库是一个开源的NewSQL数据库，其灵感来自谷Spanner。YugabyteDB是根据Apache2开源许可证获得许可的，其背后的公司还销售该数据库的托管云版本。与CockroachDB一样，YugabyteDB没有用于TrueTime协议的专用硬件时钟，而是使用混合逻辑时钟。该数据库支持快照隔离和串行化，并允许用户选择所需的事务隔离级别。YugabyteDB是一个全新的数据库，其SQL接口将于2019年底投入生产。

以下各节以与CockroachDB和TiDB描述相同的格式深入描述了YugabyteDB：体系结构、存储引擎、事务处理和地理复制。我们的描述基于YugabyteDB在线文档[26]。

1、Architecture

YugabyteDB集群由具有两个独立角色的节点组成：主节点和存储实例，称为tablet Server（TServer）。主节点用Raft复制，并作为单个节点，即无分片。Tserver也可以通过Raft进行复制，但也可以分片以支持水平可伸缩性。

YugabyteDB中的主节点负责放置分片并存储整个系统的元数据。然而，与TiDB不同的是，主节点并不参与每个事务，因为它们只控制数据的放置和移动。相反，YugabyteDB依赖混合逻辑时钟来排序事务。

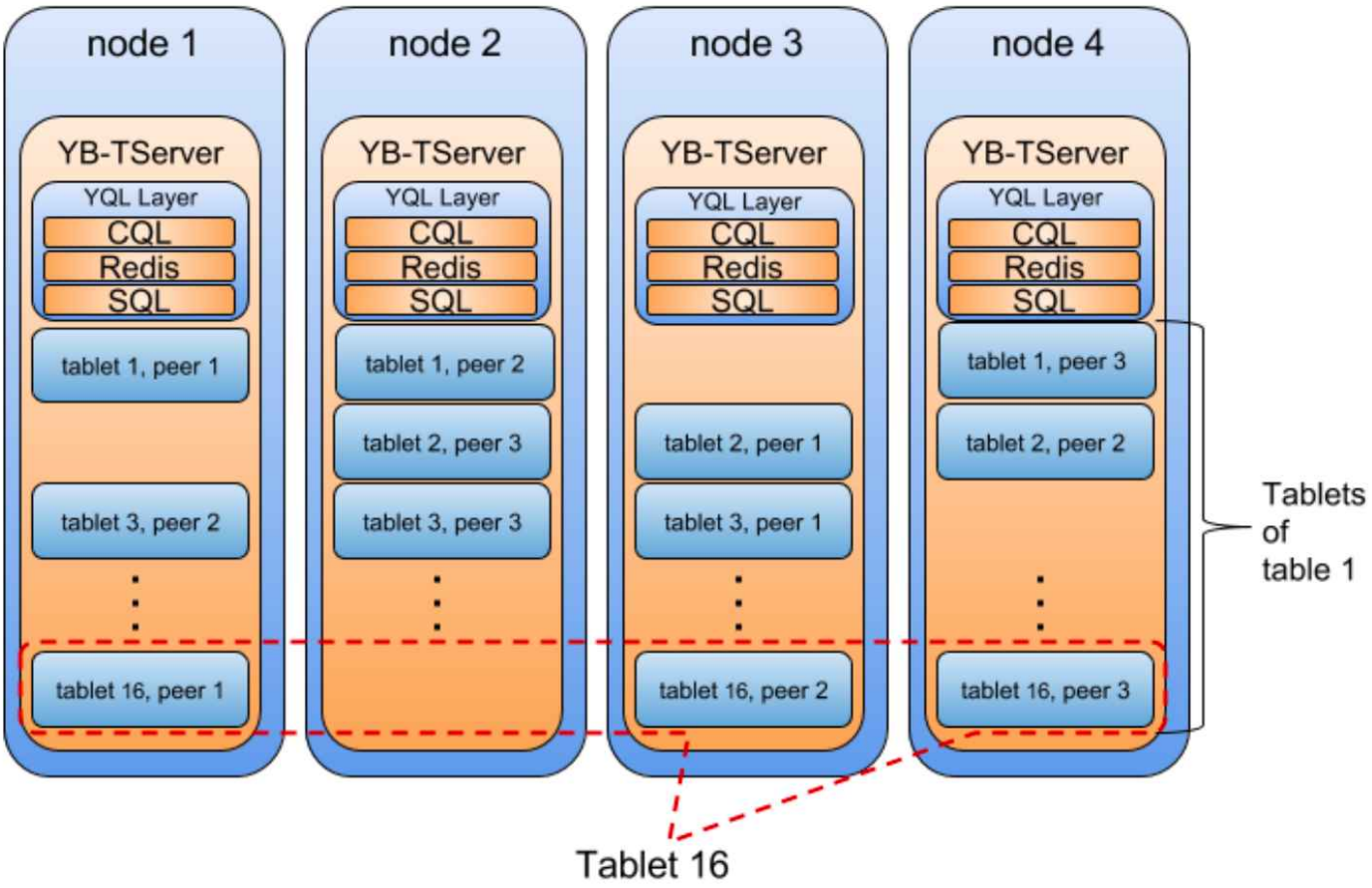
YugabyteDB中的TServer通过主键的散列进行分片，并且它们还被复制到可配置数量的节点，通常为三个。Tserver从SQL客户端接收SQL请求，然后将这些请求转换为由文档层（称为DocDB）处理的键值操作。任何TServer都可以处理查询，客户端不需要知道数据的位置。除了SQL客户端之外，YugabyteDB还支持Redis和Cassandra查询接口，它们都使用相同的底层键值存储来完成查询。

2、Storage engine

YugabyteDB的存储引擎称为DocDB，事务在此层中处理。DocDB是一个键值数据库，是YugabyteDB的SQL、Redis和Cassandra接口的底层存储，但在本节中，我们将只关注支持SQL的特性。DocDB是一个分片系统，每个分片都称为tablet，这就是为什么YugabyteDB中的存储节点称为Tablet Servers（Tserver）。每个tablet都很小，并与Raft同步复制以确保一致性，通常复制到三个节点。DocDB的设计使不同的SQL表永远不会分配给同一个tablet，但一个表可能由多个tablet组成。图2.3显示了DocDB的分片示例，其中有一个由16个tablet组成的表，分布在四个节点之间。

对于Raft读取，DocDB实现了第2.7节所述的leader lease。这大大提高了DocDB的读取性能，同时仍确保了一致性。DocDB还支持follower读取，但这并不保证客户端接收最新数据。

DocDB的底层存储系统是RocksDB，它用作MVCC存储。DocDB选择为每个tablet使用一个RocksDB实例，这意味着每个TServer将运行许多独立的RocksDB实例。这样做的原因是，将tablet复制到另一个节点非常简单，因为磁盘上的原始SSTable文件可以复制到其他节点。此外，表删除意味着可以简单地删除RocksDB实例，而无需创建tombstone记录并等待压缩以释放空间。DocDB使用RocksDB作为MVCC存储，每个key都有一个时间戳版本，因此还利用了RocksDB对前缀bloom过滤器的支持。



3、Transaction handling

YugabyteDB与TiDB和CockroachDB一样，使用MVCC进行并发控制。每个密钥都有一个时间戳，因此同一密钥可以有多个版本。为了确保适当的隔离，事务s不写正常记录，而是写所谓的临时记录。临时记录通过在键上有一个特殊前缀来标记，但它们始终存储在实际记录存储的同一个tablet中，以实现对临时记录的原子替换。

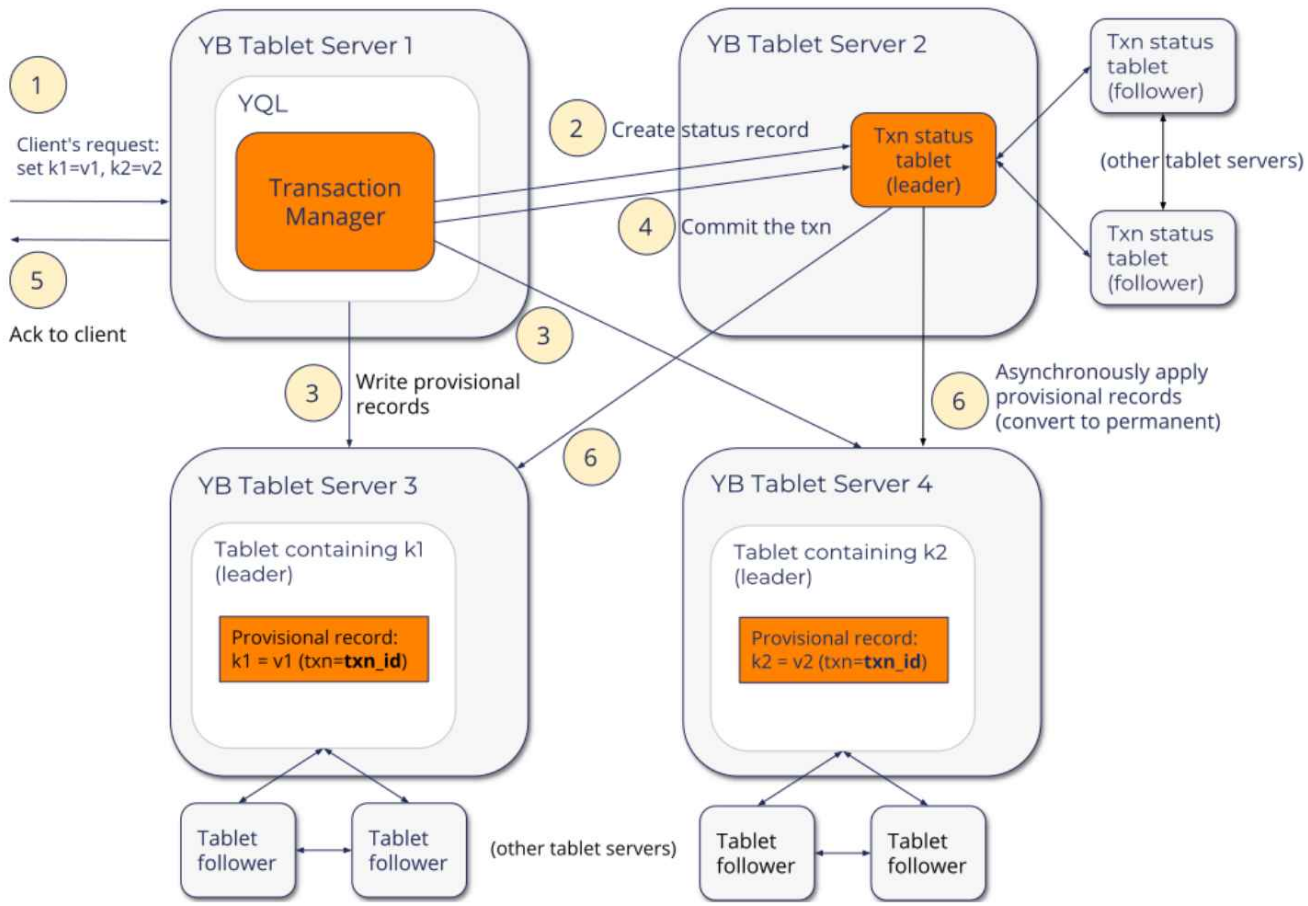
YugabyteDB中的事务，或者更具体地说，DocDB中的事务在事务表中进行跟踪。由事务创建的任何临时记录都指向事务表中的相应记录，这确保了其他事务始终可以查找临时记录的状态。对事务记录的任何更改都将作为整个事务的原子操作，这将启用YugabyteDB中的原子性。每当事务提交时，它将首先更新事务记录，然后将所有临时记录异步转换为正常记录。但是，一旦事务完成，客户端就会收到一个结果，不需要等待异步清理。发起事务的TServer成为该事务的管理者，负责协调事务的执行并将结果返回给客户端。

当事务想要写入数据时，它首先需要获取相关数据的锁。然而，锁不是显式存储的，而是临时记录被视为锁[20]。此外，raft将把所有这些锁保存在内存中，以便快速访问。如果事务想要写入数据，但临时记录已经存在，则必须根据优先级中止其中一个事务。为了中止具有锁的事务，可以删除该事务的临时记录。但是，这也意味着所有事务在提交之前都需要检查其所有临时记录是否仍然存在，以确保事务不会因冲突而中止。

当事务读取数据时，它使用自己的时间戳选择要读取的正确MVCC版本。始终选择低于事务时间戳的最高版本。但是，由于HLC具有一些不确定性，因此无法始终确定密钥是在当前事务之前还是之后写入

的。在这种情况下，整个事务被中止，并使用稍后的时间戳重试，以确保一致的读取。如果读取遇到临时记录，则需要从事务表中查找事务状态。如果事务已提交，则该值视为正常值，而如果事务中止，则忽略该值。但是，如果事务仍在进行中，则需要中止事务并使用稍后的时间戳重试。

图2.4显示了涉及写入多个tablet事务的写入路径。首先，客户端向TServer发送请求（1）。在图中，这是一个键值请求，但当使用YugabyteDB作为NewSQL系统时，它也可能是一个SQL请求。然后，该TServer成为事务管理器，并在事务表（2）中创建一条记录。接下来，事务管理器通过联系每个受影响平板电脑的Raft负责人来创建临时记录（3）。最后，通过更新事务记录（4）执行提交，客户端接收响应（5）。提交事务后，临时记录将自动转换为正常记录（6）。该图还显示，每个平板电脑有两个追随者，这是一种典型的部署，即所有数据都有三个副本。



4、Geo-replication

YugabyteDB目前没有明确支持地理复制，但他们的路线图1上有一些分区功能。Yugabetedb集群目前可以在不同的区域运行，但操作可能会有很高的延迟，并且用户无法控制数据放置。

YugabyteDB还支持follower读取，这可能有助于某些需要高可用性的应用程序，并且可以通过允许读取获取陈旧数据来牺牲一些保证。