High Performance TiDB 7: transactions

Written by Hangzhi

Hi! I'm Nick Cameron, I'm an engineer working on TiKV at PingCAP. This talk was made with help from Xu Rui and Zhao Lei.

嗨！我是 Nick Cameron，我是 PingCAP 的 TiKV 工程师。这个讲座是在 Xu Rui 和 Zhao Lei 的帮助下进行的。

In this lesson we're going to talk about distributed transactions in TiDB and what makes them fast or slow.

在这节课中，我们将讨论 TiDB 中的分布式事务，以及使它们快或慢的原因。

In the first two sections, I'll cover some background - about TiDB's architecture and its consistency guarantees. Maintaining those consistency guarantees is why we need to support transactions.

在前两节，我将讨论一些关于 TiDB 基础架构以及其提供的一致性背景知识。维持这些一致性保障是我们需要支持事务的原因。

Then I'll talk about how distributed transactions are implemented in TiDB, and how that implementation has been optimised.

然后，我们将讨论 TiDB 中的分布式事务是如何实现的，以及如何对该实现进行优化。。

OK, first let's go over a few parts of TiDB's architecture. We need to understand these things in order to understand the transaction implementation.

好的，首先让我们介绍一下 TiDB 基础架构的几个部分。为了解事务的实现原理，我们需要了解这些基础内容。

In a TiDB cluster, there are three kinds of node: TiDB, TiKV, and PD.

在 TiDB 集群中，有三种类型的节点：TiDB，TiKV 和 PD。

When we're talking about transactions, the only thing which PD does is supply timestamps. So the main participants are TiDB and TiKV nodes.

当我们谈论事务时，PD 唯一要做的就是提供时间戳。所以，主要参与者是 TiDB 和 TiKV 节点。

TiDB nodes take SQL from the client and translate that into operations on keys and values. In this talk we're only interested in the last step of that translation, which is how the key-value operations are dispatched to the TiKV nodes.

TiDB 节点从客户端获取 SQL，并将其转换为对键和值的操作。在本课程中，我们仅关注事务的最后一步转换过程，也就是如何将键值操作分配给 TiKV 节点。

So, we'll only be talking about keys and values, not more complex types like tables, rows, or SQL queries.

所以，我们将讨论键和值，而不是表、行或 SQL 查询等更复杂的类型。

TiKV nodes take those key-value operations and update their own state.

TiKV 节点执行这些键值操作并更新自己的状态。

The first stage of that is implemented in the storage module, which translates those transactional operations on key-values into raw data operations (i.e., get/put/delete operations with no concept of transactions). Those are replicated by Raft and stored by RocksDB. For this lesson, we'll abstract over that and view the output of storage as raw key-value operations.

其第一阶段是在存储模块中实现的, 这个阶段将对于键-值的事务操作转化成原始数据操作（即，没有事务概念的 get / put / delete 操作）。它们由 Raft 复制并由 RocksDB 存储。在本节课中，我们将对此进行抽象，并将存储模块的输出视为原始键-值操作。

So to step back, when we're talking about transactions, these are the components involved.

退一步看，在我们谈论事务的时候，这些都是相关的组件。

It's a little bit more complex than that because the database is sharded into regions. Regions are replicated across TiKV nodes using Raft, but as we said earlier we can abstract over the Raft layer.

这会复杂一点，因为数据库被分片到各个 Region, 而 Region 在 TiKV 节点之间被 Raft 复制，但是正如我们之前所说，我们可以在 Raft 层上进行抽象。

TiDB only talks to the leader nodes in each region, so we can think of each region being hosted on just a single node.

TiDB 仅与每个 Region 中的 leader 节点对话，因此我们可以认为每个 Region 都仅托管在单个节点上。

In the next section I'm going to cover the consistency guarantees of TiDB

在下一节中，我将介绍 TiDB 的一致性保障

We often talk about the validity of transaction systems in terms of the acronym 'ACID'. That is transactions should be atomic, which means either all of a transaction is committed or none of it is; consistent which is a property of how transactions are used, so isn't relevant here; isolated which is the most interesting bit and we'll talk about in detail; and durable, which means that once a transaction is committed, it will not be lost.

我们经常用缩写 "ACID" 来谈论事务系统的有效性。也就是说是事务应该是原子性的，这意味着要么所有事务都已提交，要么都不提交；一致性作为事务使用方式的一个属性，在这里并不重要；隔离性是最有趣的一点，我们将详细讨论；持久性，这意味着一旦提交了事务，它就不会丢失。

Confusingly, the 'isolation' in ACID is usually called 'consistency' in the distributed systems world, but that is not the same as the 'consistency' in ACID.

困惑的是，ACID 中的"隔离性"在分布式系统世界中通常称为"一致性"，但是与 ACID 中的"一致性"不同。

The property supported by TiDB is 'snapshot isolation'. The intuition with snapshot isolation is that you freeze time at the start of a transaction, and then every read sees the data as if it were at that point in time.

TiDB 支持的特性是"快照隔离级别"。快照隔离在直觉上是你在事务开始时冻结时间，然后在每次读取时，都将数据视作在该时间节点上。

It's a stronger property than read committed. Like read committed, you can't read another transaction's reads until they're committed. In addition, snapshot isolation prevents phantom reads and non-repeatable reads within a transaction.

这是比"读已提交"更强大的特性。就像"读已提交"一样，你无法读取其他事务未提交的数据内容。此外，快照隔离可防止事务中的幻读和不可重复读取。

Snapshot isolation is not as strong as serializability. It does not enforce a total ordering over all operations. Since all reads effectively happen at the start of a transaction, it is possible for

a transaction to write a value based on reading an old version of a different value. That anomaly is called write skew.

快照隔离不如可串行性强。它不会保证所有操作的全序关系。由于所有读取实际上都是在事务开始时发生的，因此事务可以基于一个旧版本的值来写入一个值。这种异常现象被称为写偏序。

In the next couple of sections I'll describe how snapshot isolation is implemented in TiDB. I'll start with the Percolator protocol which ensures atomicity and isolation of transactions.

接下来的两节，我将描述 TiDB 如何实现快照隔离。我将从确保事务原子性和隔离性的 Percolator 协议开始。。

Before we get into the details, the protocol here is based on Percolator from Google. The Percolator paper is a great read if you're interested in learning more about transactions and the Percolator protocol.

在开始详细介绍之前，此处的协议是指基于 Google 的 Percolator。如果您有兴趣了解有关事务和 Percolator 协议的更多信息，那么 Percolator 论文是不错的阅读材料。

TiDB transactions are collaborative, which means both TiKV and TiDB nodes take part and both must be implemented correctly for the protocol to be sound.

TiDB 事务是协作的，这意味着 TiKV 和 TiDB 节点都参与其中，并且协议的健全依赖于两者都正确实现

As we'll see, timestamps are essential to the transaction system. Timestamps are not hours-minutes-seconds or something similar, they are just an unsigned integer. They are a combination of the wall-clock time according to PD, and a counter, but what's important is that every timestamp is different and that they are monotonic and linear, which means that if you ask PD for two timestamps, the second one issued by PD will always be larger than the first.

正如我们将看到的，时间戳对于事务系统至关重要。时间戳不是小时-分钟-秒之类的，它们只是一个无符号整数。它们是根据 PD 的挂钟时间和一个计数器的组合，而重要的是每个时间戳都是不同的，并且它们是单调且线性的，这意味着如果您向 PD 请求两个时间戳，则由 PD 发出的第二个时间戳将始终大于第一个。

For this lesson, we'll consider two timestamps for each transaction, one for the start and one for when the transaction is committed. These are called startTS and commitTS, respectively.

The startTS is the time at which all reads happen and the commitTS is the time when all writes appear visible to other transactions.

在本课程中，每个事务我们都将务考虑两个时间戳，一个用于开始时，一个用于提交事务时。它们分别称为 startTS 和 commitTS。startTS 是所有读取发生的时间，commitTS 是所有写入对其他事务可见的时间。

TiDB transactions follow a two-phase commit protocol. That means that once a TiDB node has built up a transaction and is ready to commit it, there are two roundtrips between the TiKV and TIDB nodes before the transactions is considered committed. The two phases are called prewrite and commit.

TiDB 事务遵循两阶段提交协议。这意味着，一旦 TiDB 节点建立了事务并准备提交，事务真正被提交前在 TiKV 和 TiDB 节点之间则会有两次网络交互。这两个阶段称为预写和提交。

First of all, the client tells TiDB that it's starting a transaction and TiDB fetches a startTS from PD. The transaction is built up by the client and TiDB, data is read from a snapshot of the database at the startTS and writes are buffered locally. When the client ends the transaction, TiDB sends prewrite messages to the TiKV nodes which hold the written keys. The prewrite message contains the values to be written. TiKV will lock every key and record the new values, then return success to TiDB if it locked every key and didn't meet any other error. If a node returns success, then it is promising that if TiDB decides to commit the transaction, then it will not fail.

首先，客户端告诉 TiDB 它正在开始事务，并且 TiDB 从 PD 获取一个 startTS。事务由客户端和 TiDB 建立，数据在 startTS 的数据库快照上读取，而写入操作被缓存在本地。当客户端结束事务时，TiDB将预写消息发送到保存已写键的TiKV节点。预写消息包含要写入的值。TiKV 将锁定每个键并记录新值，然后如果它锁定了每个键且未遇到任何其他错误，则返回“成功”给 TiDB。如果节点返回“成功”，那么将保证，如果TiDB决定提交事务，那么它不会失败。

TiDB collects its responses from all the nodes. If it gets success from each one, then it will begin the second phase. It will get a commitTS from PD and then send commit messages to the TiKV nodes. TiKV will then make the changes it recorded in the prewrite phase visible to other transactions from the commitTS onward. The commit should always succeed and the transaction is durably written to the database, so TiDB can return success to the client.

TiDB 从所有节点收集其响应。如果每个节点的响应都成功，那么它将开始第二阶段。它将从 PD 获取 commitTS，然后将提交消息发送到 TiKV 节点。然后，TiKV 将使它在预写阶段中记录的更改从 commitTS 开始对其他事务可见。提交始终成功并且事务持久地写入数据库，则 TiDB 可以将成功返回给客户端。

Well, when I said that TiDB sends commit messages, it's a bit more complicated than that.

当谈到 TiDB 发送提交消息时，实际上会比这个复杂一些。

In the prewrite phase, TiDB will select one of the keys to be the primary key (nothing to do with primary keys in SQL). The status of the primary key is the source of truth for the status of the whole transaction. That means that TiDB only needs to wait to commit the primary key before it can return success to the client, the other locks can later be committed asynchronously.

在预写阶段，TiDB 选择其中一个键作为主键（与 SQL 中的主键无关）。主键的状态是整个事务状态的真实来源。这意味着 TiDB 只需等待提交主键，即可将其成功信息返回给客户端，而其他锁则可以稍后异步提交。

There is a lot going on here. Here's a timeline diagram showing an example with two TiKV nodes. I'm not sure if this will actually make things clearer, to be honest.

这里有很多事情。这是一个时间线图，显示了具有两个TiKV节点的示例。老实说，我不确定这是否真的会更清楚。

Time runs top to bottom. The blue phase is preparation of the transaction. The prewrite phase is green and the commit phase is pink.

时间流向是自上而下的。蓝色是事务准备阶段。绿色为预写阶段，粉色为提交阶段。

Being a distributed system, there are lots that can go wrong. TiDB could lose messages or nodes, or the network could be partitioned, and so forth.

作为分布式系统，很多事情肯能会出现偏差。TiDB 可能会丢失消息或节点，或者网络可能被分区等等。

I'll mostly leave it as an exercise to work out how the protocol remains reliable under these conditions, but one situation we might have to deal with is keys being locked without ever being unlocked. Then some other transaction wants to write to the locked key and TiDB has to recover somehow.

我主要把弄清协议在这些情况下如何保持可靠性留作练习，但是我们可能要处理的一种情况是，被锁键从未被解锁。之后，其他一些事务想要写入锁定键，而 TiDB 必须以某种方式恢复。

The way that recovery works is that TiKV returns a specific error to TiDB with the contents of the lock. That lets TiDB look up the primary key (which might be on a different node), to find

the status of the transaction. The abandoned lock can then be committed or rolled back as appropriate and the transaction can be retried.

恢复方式是 TiKV 向 TiDB 返回包含锁的内容的具体报错信息。这让 TiDB 可以搜寻主键（可能位于其他节点上），以查找事务的状态。然后可以适当地提交或回退废弃的锁，以及重试该事务。

Alright, so that's what the transaction protocol looks like from the outside, how is that implemented in TiKV?

好了，这就是从外部看事务协议的样子，而它是如何在 TiKV 中实现的呢？

We use multi-version concurrency control (or MVCC), which is the standard way to implement snapshot isolation.

我们使用多版本并发控制（MVCC），这是实现快照隔离的标准方法。

MVCC means that for every key, we don't just store the current value, but we store a history of values. In theory, we store every value the key has ever held. We keep the value and the timestamp when it was committed.

MVCC 意味着对于每个键，我们不仅存储当前值，而且还存储历史值。从理论上讲，我们存储键曾经拥有的每一个值。我们保留提交时的值和时间戳。

When reading from keys at a given timestamp, we read the most recent value of the key, before that timestamp.

在给定时间戳下读取键时，我们会读取该时间戳之前的最新值。

In this diagram, time runs from left to right and the black dots are values.

在这张图中，时间是从左到右，黑点是值

When we take a snapshot at time T, we'll get values from different times for each key, or maybe no value at all, as for key three.

当我们在时间 T 拍摄快照时，我们将获得每个键在不同时间的值，或者，对于键 3 来说，可能根本没有值

How are these different values for each key, and locks, represented on disk? The underlying storage is just keys and values, so we must encode all those higher-level concepts.

每个键的值和锁是如何在磁盘上表示的呢？本质存储只是键和值，所以我们必须对这些更高级别的概念进行编码。

We use three column families. You can think of these as separate key-value databases, the only difference being that writes to multiple column families can be made atomically.

我们使用三个列族。你可以将它们视为单独的键值数据库，唯一的区别是可以原子方式写入多个列族。

We store user values in the default column family. The keys we use in the default CF are a combination of the user key and the start timestamp of the transaction. This is where we record values during the prewrite phase.

我们将用户值存储在默认列族中。我们在默认 CF 中使用的键是用户键和事务开始时间戳的组合。这是我们在预写阶段记录值的地方。

The write CF stores metadata which indicates the status of a value. For the key in the write CF we use the user key and the commit timestamp, and for the value we keep the kind of write - whether it is a put or a delete, etc, and the startTS of the transaction. TiKV puts an entry in the write CF in the commit phase of the two-phase commit.

写入 CF 存储指示值状态的元数据。对于写入 CF 中的键，我们使用用户键和提交时间戳，对于值，我们保留写入类型-无论是 put 还是 delete 之类的，以及事务的 startTS。TiKV 在两阶段提交的提交阶段在写入 CF 中 put 一个条目。

A neat trick is that the keys are encoded in such a way that keys are stored in descending order of the commit timestamp, so the most recent version of a key is stored first.

一个秘诀是，键的编码方式使得键能够以提交时间戳的降序进行排列。因此，键的最新版本会被排序在最前面。

Finally there is the lock column family where we store locks. These are written in the prewrite phase and removed in the commit phase. For locks we only store one value per key, not multiple versions. When a key is locked, it is locked for all timestamps.

最后是我们存储锁的锁列族。它们在预写阶段写入，并在提交阶段删除。对于锁，我们每个键仅存储一个值，而不存储多个版本。当一个键被锁定时，它在所有时间戳中都被锁定。

Next we'll see how the MVCC representation is actually used. I'll go over a few actions and how they're represented.

接下来，我们将了解 MVCC 表现形式是如何被使用的。我将介绍一些操作以及它们的表示方式。

I'll start with the prewrite action. In response to a prewrite message, TiKV will first check the write column family for a write conflict. If there is no conflict, it will check for a lock; if the key is not locked, it will write a lock and then write the user value into the default CF.

我将从预写动作开始。作为对预写消息的响应，TiKV 将首先检查写列族是否存在写冲突。如果没有冲突，它将检查锁定。如果键未锁定，它将写入一个锁定，然后将用户值写入默认的 CF。

On commit, TiKV checks a key's lock, the key should have been locked in prewrite. It will then write an entry to the write CF and delete the lock from the lock CF.

提交时，TiKV 将检查键的锁定，该键应已被锁定为预写。然后它将一个入口写入 write CF，并从 lock CF 中删除该锁。

If a transaction is to be rolled back, TiKV deletes the lock and value from the lock and default CFs and writes a rollback entry into the write CF. That is so that if TiKV later gets another prewrite or rollback message, it knows that this transaction has already been rolled back.

如果要回滚事务，则 TiKV 从 lock 和 default CF 中删除锁和值，并将回滚条目写入 write CF 中。这样一来，如果 TiKV 以后收到另一个预写或回滚消息，它就会知道该事务已被回滚。

To read a key at a given timestamp, TiKV starts with the write CF and scans until it finds the first write before the timestamp. The write contains the start timestamp for the transaction which wrote it, and using the key and start TS, TiKV can find the value from the default CF.

为了在给定时间戳读取键，TiKV 从 write CF 开始并进行扫描直到找到时间戳之前的第一个写入。这个写入包含写入它的事务的开始时间戳，而且通过键和 startTS，TiKV 可以从 default CF 中找到该值。

OK, so that's an introduction to TiDB's transactions. In the next section I'll start talking about the performance of the transaction system. This is important because with all those network trips and disk accesses, the performance of the transaction system makes a big impact on the performance of the database as a whole.

好的，这就是 TiDB 事务的基础介绍。在下一节中，我将开始讨论事务系统的性能。这很重要，因为在进行所有这些网络通信和磁盘访问后，事务系统的性能会对整个数据库的性能产生重大影响。

Before we start, we should consider what are the hot and cold paths. Of course it will depend on the specific workload, but usually we can expect many reads and writes, the majority of which won't interfere with another transaction. Exactly how many writes end up as conflicts depends a lot on the workload. Failures are rare, and so recovery actions are rare too, they are very cold paths in the code. However, we can't completely ignore them because they are often the cause of high peak latency.

在开始之前，我们应该考虑什么是冷热路径。当然，这将取决于特定的工作负载，通常我们会有许多读写操作，但其中大多数不会干扰其他事务。到底有多少次写入最终会冲突很大程度上取决与工作负载。故障很少发生，因此恢复操作也很少发生，它们是代码中非常冷的路径。但是，我们不能完全忽略它们，因为它们通常是高峰值延迟的原因。

I'll start by going over some improvements to the protocol itself, then later go over some optimisations of the implementation.

我会首先回顾协议本身的一些改进，然后再是一些实现的优化。

The transaction protocol I described earlier is an optimistic protocol. It assumes there won't be a write conflict and in the prewrite phase we check that. Transactions can also be pessimistic, then we can lock keys as we're building up the transaction. If locking succeeds, then prewrite will succeed too. As well as avoiding more conflicts, pessimistic transactions also permit partial retries. The downside is that with all that locking, transactions can now deadlock, so we need a deadlock detector.

我之前描述的事务协议是一种乐观协议。它假设不会发生写冲突，并且在预写阶段我们会进行检查这个。事务也可能是悲观的，那么我们在建立事务时可以锁定键。如果锁定成功，则预写也将成功。除了避免更多冲突外，悲观事务还允许部分重试。不利的一面是，有了所有这些锁定，事务现在可能死锁，因此我们需要一个死锁检测器。

Async commit is an improvement which we are currently implementing. If you remember the transaction protocol, I said that after prewrite succeeds, TiKV promises that committing will succeed. So if all prewrites succeed, then the commit phase must succeed too. So then why do we need to wait for the commit to complete (and a network round trip) before returning success to the client? Essentially it is in case the commit message gets lost and the commit is never recorded, then later the transaction might be rolled back even though the client thinks its committed.

异步提交是我们当前正在实现的一项优化。如果您还记得事务协议，我说过在预写成功之后，TiKV 保证提交将成功。因此，如果所有预写都成功，那么提交阶段也必须成功。那么，为什么我们需要在返回成功信息给客户端之前等待提交完成（以及一个网络往返）？本质上，万一提交消息丢失并且从未记录过提交，则以后即使客户认为已提交，事务也可能回滚。

With async commit, we can return early to the client, once all prewrites succeed. As you can see from the diagram, that can really improve latency.

有了异步提交，一旦所有预写成功，我们就可以尽早返回客户端。从图中可以看出，这确实可以改善延迟。

Of course we still have to handle the case that the commit messages are lost. We do this by treating the status of the transaction as distributed amongst all the locks, rather than being recorded only with the primary key. This makes lots of things a lot more complex, but we hope it's worth it for the latency improvement.

当然，我们仍然必须处理提交消息丢失的情况。为此，我们将事务状态视为分布在所有锁中，而不是仅使用主键进行记录。这使很多事情变得更加复杂，但是我们希望在延迟方面进行的改进是值得的。

We're getting into some smaller scale optimisations now. These are all details of the implementation, but they can still make a decent difference.

我们现在来看一些较小规模的优化。这些都是实现的细节，但是它们仍然可以发挥大的作用。

Pipelining of pessimistic transactions is an optimisation of the prewrite phase. If a key has been locked, then it should be impossible for the prewrite of that key to fail. Therefore, TiKV can return success to TiDB after doing some basic validity checking, and can make the prewrite changes to the raftstore asynchronously. That saves on both disk and network IO.

悲观事务的流水线化是预写阶段的优化。如果某个键已被锁定，则该键的预写不应当失败。因此，TiKV 可以在进行一些基本的有效性检查之后将成功返回给 TiDB，并且可以异步地对 Raft存储进行预写更改。这样可以节省磁盘和网络 IO。

Any time there is IO happening, batching that IO is likely to improve performance. In TiDB, IO is batched together in multiple places - commands sent between nodes are batched together, reads and writes to Raft are batched together, and so forth.

每当发生 IO 时，对该 IO 进行批处理会提高性能。在 TiDB 中，IO 批处理发生在不少地方-节点之间发送的命令一起批处理，对 Raft 的读写也一起批处理，依此类推。

If you recall the MVCC implementation, I said we store the values in the default column family and a record of the transaction in the lock and write column families. If the value is small, there is no advantage to doing that, and we can store the value directly in the write or lock CF. That saves a disk read or write on every access. We don't do this for all values because with larger values there is a cost to copying the value and it would make scanning the write and lock CFs slower.

如果你还能记得 MVCC 实现，我说过我们将值存储在默认列族中，并将事务记录存储在锁和写列族中。如果该值很小，则这样做没有什么好处，我们可以将值直接存储在 write或者lock CF中。这样可以节省每次访问时的磁盘读或写操作。我们不对所有值都执行此操作，因为值较大时，复制该值会产生成本，这会使扫描 write和 lock CF 的速度变慢。

When reading, we scan the write CF to find the first non-rollback write. We can make that scan faster if we collapse rollbacks together, then we only need to skip one entry rather than n entries. Unfortunately, we have to be a bit careful because some rollbacks cannot be skipped or it could cause correctness problems.

读取时，我们扫描 write CF 以找到第一个非回滚写入。如果将回滚折叠在一起，我们可以使扫描更快，这样我们只需要跳过一个条目，而不是 n 个条目。不幸的是，我们必须谨慎一些，因为某些回滚不能被跳过，否则可能会导致正确性问题。

Okay. The following few issues are transaction optimisations for which you can score points for the course.

好的。以下是你们能够获得分数的事务优化问题，

I'll just let you read through these slides, I don't think there's much point in reading them all.

This one, issue 8653 is also part of the tidb challenge program. The high-performance TiDB challenge contest started last Friday, and there are special prizes.

我只是让您浏览这些幻灯片，我认为阅读所有幻灯片没有多大意义。

Issue 8653 也是 TiDB 性能竞赛的一部分。 TiDB 性能竞赛于上周五开始，而且有丰富的奖励。

I think you've seen this slide before. If you have any questions about how to pick issues and submit your work, please check the 'ti-challenge-bot introduction' video.

我想你们已经看过这张幻灯片。如果您对如何挑选 issue 和提交作业有任何疑问，请查看 'Ti-Challenge-Bot' 的简介视频。

If you have any questions, please scan this QR code and join our wechat group. We also welcome suggestions and feedback from you.

如果有任何问题，请扫描这张二维码并且加入我们的微信群。我们也非常欢迎你们提出建议和反馈。

There are more courses available at PingCAP University. You can scan this QR code or visit the website, the URL on the slide to find those lessons.

PingCAP University上还有很多课程。你们可以扫描二维码或者访问幻灯片上的网站链接来找到这些课程。

Thanks for attending the lesson. I hope you found it interesting and useful. And good luck if you choose to implement one of the issues.

非常感谢你们来参加这个课程。希望你们觉得课程有用且有趣。如果你们选择实现其中的一个issue，祝你们好运！

专业词汇对照：

以下是我对一些专业词汇的翻译，如有不妥，请指正

Transaction: 事务

Architecture: 架构

Consistency guarantee: 一致性保证。

Region：分区

Read Committed: 读已提交

Phatom read: 幻读

Serializability 可串行性

Write skew 写偏序