Google Spanner (中文版)

Google Spanner (中文版)

摘要：Spanner是谷歌公司研发的、可扩展的、多版本、全球分布式、同步复制数据库。它是第一个把数据分布在全球范围内的系统，并且支持外部一致性的分布式事务。本文描述了Spanner的架构、特性、不同设计决策的背后机理和一个新的时间API，这个API可以暴露时钟的不确定性。这个API及其实现，对于支持外部一致性和许多强大特性而言，是非常重要的，这些强大特性包括：非阻塞的读、不采用锁机制的只读事务、原子模式变更。

中文关键词：谷歌，分布式数据库

英文关键词: Google, Spanner, Bigtable, Distributed Database

全文目录结构

1 介绍

Spanner是一个可扩展的、全球分布式的数据库，是在谷歌公司设计、开发和部署的。在最高抽象层面，Spanner就是一个数据库，把数据分片存储在许多Paxos[21]状态机上，这些机器位于遍布全球的数据中心内。复制技术可以用来服务于全球可用性和地理局部性。客户端会自动在副本之间进行失败恢复。随着数据的变化和服务器的变化，Spanner会自动把数据进行重新分片，从而有效应对负载变化和处理失败。Spanner被设计成可以扩展到几百万个机器节点，跨越成百上千个数据中心，具备几万亿数据库行的规模。

应用可以借助于Spanner来实现高可用性，通过在一个洲的内部和跨越不同的洲之间复制数据，保证即使面对大范围的自然灾害时数据依然可用。我们最初的客户是F1[35]，一个谷歌广告后台的重新编程实现。F1使用了跨越美国的5个副本。绝大多数其他应用很可能会在属于同一个地理范围内的3-5个数据中心内放置数据副本，采用相对独立的失败模式。也就是说，许多应用都会首先选择低延迟，而不是高可用性，只要系统能够从1-2个数据中心失败中恢复过来。

Spanner的主要工作，就是管理跨越多个数据中心的数据副本，但是，在我们的分布式系统体系架构之上设计和实现重要的数据库特性方面，我们也花费了大量的时间。尽管有许多项目可以很好地使用BigTable[9]，我们也不断收到来自客户的抱怨，客户反映BigTable无法应用到一些特定类型的应用上面，比如具备复杂可变的模式，或者对于在大范围内分布的多个副本数据具有较高的一致性要求。其他研究人员也提出了类似的抱怨[37]。谷歌的许多应用已经选择使用Megastore[5]，主要是因为它的半关系数据模型和对同步复制的支持，尽管Megastore具备较差的写操作吞吐量。由于上述多个方面的因素，Spanner已经从一个类似BigTable的单一版本的键值存储，演化成为一个具有时间属性的多版本的数据库。数据被存储到模式化的、半关系的表中，数据被版本化，每个版本都会自动以提交时间作为时间戳，旧版本的数据会更容易被垃圾回收。应用可以读取旧版本的数据。Spanner支持通用的事务，提供了基于SQL的查询语言。

作为一个全球分布式数据库，Spanner提供了几个有趣的特性：第一，在数据的副本配置方面，应用可以在一个很细的粒度上进行动态控制。应用可以详细规定，哪些数据中心包含哪些数据，数据距离用户有多远（控制用户读取数据的延迟），不同数据副本之间距离有多远（控制写操作的延迟），以及需要维护多少个副本（控制可用性和读操作性能）。数据也可以被动态和透明地在数据中心之间进行移动，从而平衡不同数据中心内资源的使用。第二，Spanner有两个重要的特性，很难在一个分布式数据库上实现，即Spanner提供了读和写操作的外部一致性，以及在一个时间戳下面的跨越数据库的全球一致性的读操作。这些特性使得Spanner可以支持一致的备份、一致的MapReduce执行[12]和原子模式变更，所有都是在全球范围内实现，即使存在正在处理中的事务也可以。

之所以可以支持这些特性，是因为Spanner可以为事务分配全球范围内有意义的提交时间戳，即使事务可能是分布式的。这些时间戳反映了事务序列化的顺序。除此以外，这些序列化的顺序满足了外部一致性的要求：如果一个事务T1在另一个事务T2开始之前就已经提交了，那么，T1的时间戳就要比T2的时间戳小。Spanner是第一个可以在全球范围内提供这种保证的系统。

实现这种特性的关键技术就是一个新的TrueTime API及其实现。这个API可以直接暴露时钟不确定性，Spanner时间戳的保证就是取决于这个API实现的界限。如果这个不确定性很大，Spanner就降低速度来等待这个大的不确定性结束。谷歌的簇管理器软件提供了一个TrueTime API的实现。这种实现可以保持较小的不确定性（通常小于10ms），主要是借助于现代时钟参考值（比如GPS和原子钟）。

第2部分描述了Spanner实现的结构、特性集和工程方面的决策；第3部分介绍我们的新的TrueTime API，并且描述了它的实现；第4部分描述了Spanner如何使用TrueTime来实现外部一致性的分布式事务、不用锁机制的只读事务和原子模式更新。第5部分提供了测试Spanner性能和TrueTime行为的测试基准，并讨论了F1的经验。第6、7和8部分讨论了相关工作，并给出总结。

2 实现

本部分内容描述了Spanner的结构和背后的实现机理，然后描述了目录抽象，它被用来管理副本和局部性，并介绍了数据的转移单位。最后，将讨论我们的数据模型，从而说明，为什么Spanner看起来更加像一个关系数据库，而不是一个键值数据库；还会讨论应用如何可以控制数据的局部性。

一个Spanner部署称为一个universe。假设Spanner在全球范围内管理数据，那么，将会只有可数的、运行中的universe。我们当前正在运行一个测试用的universe，一个部署/线上用的universe和一个只用于线上应用的universe。

Spanner被组织成许多个zone的集合，每个zone都大概像一个BigTable服务器的部署。zone是管理部署的基本单元。zone的集合也是数据可以被复制到的位置的集合。当新的数据中心加入服务，或者老的数据中心被关闭时，zone可以被加入到一个运行的系统中，或者从中移除。zone也是物理隔离的单元，在一个数据中心中，可能有一个或者多个zone，例如，属于不同应用的数据可能必须被分区存储到同一个数据中心的不同服务器集合中。

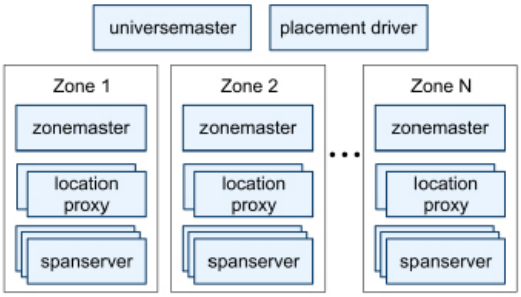

Figure 1: Spanner server organization.

图1显示了在一个Spanner的universe中的服务器。一个zone包括一个zonemaster，和一百至几千个spanserver。Zonemaster把数据分配给spanserver，spanserver把数据提供给客户端。客户端使用每个zone上面的location proxy来定位可以为自己提供数据的spanserver。Universe master和placement driver，当前都只有一个。Universe master主要是一个控制台，它显示了关于zone的各种状态信息，可以用于相互之间的调试。Placement driver会周期性地与spanserver进行交互，来发现那些需要被转移的数据，或者是为了满足新的副本约束条件，或者是为了进行负载均衡。

2.1 Spanserver软件栈

本部分内容主要关注spanserver实现，来解释复制和分布式事务是如何被架构到我们的基于BigTable的实现之上的。图2显示了软件栈。在底部，每个spanserver负载管理100-1000个称为tablet的数据结构的实例。一个tablet就类似于BigTable中的tablet，也实现了下面的映射：

(key:string, timestamp:int64)->string

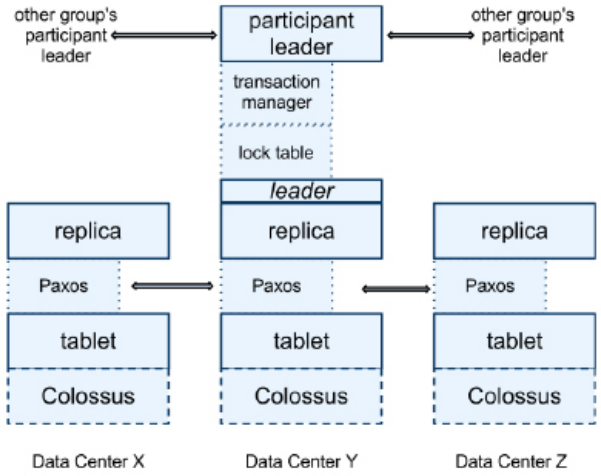

Figure 2: Spanserver software stack.

与BigTable不同的是，Spanner会把时间戳分配给数据，这种非常重要的方式，使得Spanner更像一个多版本数据库，而不是一个键值存储。一个tablet的状态是存储在类似于B-树的文件集合和写前(write-ahead)的日志中，所有这些都会被保存到一个分布式的文件系统中，这个分布式文件系统被称为Colossus，它继承自Google File System。

为了支持复制，每个spanserver会在每个tablet上面实现一个单个的Paxos状态的机器。一个之前实现的Spanner可以支持在每个tablet上面实现多个Paxos状态机，它可以允许更加灵活的复制配置，但是，这种设计过于复杂，被我们舍弃了。每个状态机器都会在相应的tablet中保存自己的元数据和日志。我们的Paxos实现支持采用基于时间的领导者租约的长寿命的领导者，时间通常在0到10秒之间。当前的Spanner实现中，会对每个Paxos写操作进行两次记录：一次是写入到tablet日志中，一次是写入到Paxos日志中。这种做法只是权宜之计，我们以后会进行完善。我们在Paxos实现上采用了管道化的方式，从而可以在存在广域网延迟时改进Spanner的吞吐量，但是，Paxos会把写操作按照顺序的方式执行。

Paxos状态机是用来实现一系列被一致性复制的映射。每个副本的键值映射状态，都会被保存到相应的tablet中。写操作必须在领导者上初始化Paxos协议，读操作可以直接从底层的任何副本的tablet中访问状态信息，只要这个副本足够新。副本的集合被称为一个Paxos group。

对于每个是领导者的副本而言，每个spanserver会实现一个锁表来实现并发控制。这个锁表包含了两阶段锁机制的状态：它把键的值域映射到锁状态上面。注意，采用一个长寿命的Paxos领导者，对于有效管理锁表而言是非常关键的。在BigTable和Spanner中，我们都专门为长事务做了设计，比如，对于报表操作，可能要持续几分钟，当存在冲突时，采用乐观并发控制机制会表现出很差的性能。对于那些需要同步的操作，比如事务型的读操作，需要获得锁表中的锁，而其他类型的操作则可以不理会锁表。

对于每个扮演领导者角色的副本，每个spanserver也会实施一个事务管理器来支持分布式事务。这个事务管理器被用来实现一个participant leader，该组内的其他副本则是作为participant slaves。如果一个事务只包含一个Paxos组（对于许多事务而言都是如此），它就可以绕过事务管理器，因为锁表和Paxos二者一起可以保证事务性。如果一个事务包含了多于一个Paxos组，那些组的领导者之间会彼此协调合作完成两阶段提交。其中一个参与者组，会被选为协调者，该组的participant leader被称为coordinator leader，该组的participant slaves被称为coordinator slaves。每个事务管理器的状态，会被保存到底层的Paxos组。

2.2 目录和放置

在一系列键值映射的上层，Spanner实现支持一个被称为“目录”的桶抽象，也就是包含公共前缀的连续键的集合。（选择“目录”作为名称，主要是由于历史沿袭的考虑，实际上更好的名称应该是“桶”）。我们会在第2.3节解释前缀的源头。对目录的支持，可以让应用通过选择合适的键来控制数据的局部性。

一个目录是数据放置的基本单元。属于一个目录的所有数据，都具有相同的副本配置。当数据在不同的Paxos组之间进行移动时，会一个目录一个目录地转移，如图3所示。Spanner可能会移动一个目录从而减轻一个Paxos组的负担，也可能会把那些被频繁地一起访问的目录都放置到同一个组中，或者会把一个目录转移到距离访问者更近的地方。当客户端操作正在进行时，也可以进行目录的转移。我们可以预期在几秒内转移50MB的目录。

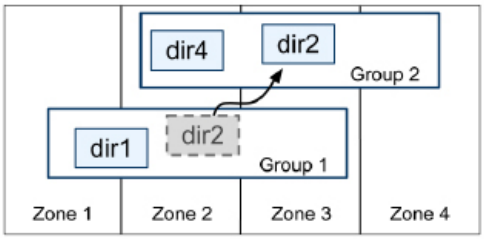

Figure 3: Directories are the unit of data movement between Paxos groups.

一个Paxos组可以包含多个目录，这意味着一个Spanner tablet是不同于一个BigTable tablet的。一个Spanner tablet没有必要是一个行空间内按照词典顺序连续的分区，相反，它可以是行空间内的多个分区。我们做出这个决定，是因为这样做可以让多个被频繁一起访问的目录被整合到一起。

Movedir是一个后台任务，用来在不同的Paxos组之间转移目录[14]。Movedir也用来为Paxos组增加和删除副本[25]，因为Spanner目前还不支持在一个Paxos内部进行配置的变更。Movedir并不是作为一个事务来实现，这样可以避免在一个块数据转移过程中阻塞正在进行的读操作和写操作。相反，Movedir会注册一个事实(fact)，表明它要转移数据，然后在后台运行转移数据。当它几乎快要转移完指定数量的数据时，就会启动一个事务来自动转移那部分数据，并且为两个Paxos组更新元数据。

一个目录也是一个应用可以指定的地理复制属性（即放置策略）的最小单元。我们的放置规范语言的设计，把管理复制的配置这个任务单独分离出来。管理员需要控制两个维度：副本的数量和类型，以及这些副本的地理放置属性。他们在这两个维度里面创建了一个命名选项的菜单。通过为每个数据库或单独的目录增加这些命名选项的组合，一个应用就可以控制数据的复制。例如，一个应用可能会在自己的目录里存储每个终端用户的数据，这就有可能使得用户A的数据在欧洲有三个副本，用户B的数据在北美有5个副本。

为了表达的清晰性，我们已经做了尽量简化。事实上，当一个目录变得太大时，Spanner会把它分片存储。每个分片可能会被保存到不同的Paxos组上（因此就意味着来自不同的服务器）。Movedir在不同组之间转移的是分片，而不是转移整个目录。

2.3 数据模型

Spanner会把下面的数据特性集合暴露给应用：基于模式化的半关系表的数据模型，查询语言和通用事务。支持这些特性的动机，是受到许多因素驱动的。需要支持模式化的半关系表是由Megastore[5]的普及来支持的。在谷歌内部至少有300个应用使用Megastore（尽管它具有相对低的性能），因为它的数据模型要比BigTable简单，更易于管理，并且支持在跨数据中心层面进行同步复制。BigTable只可以支持跨数据中心的最终事务一致性。使用Megastore的著名的谷歌应用是Gmail,Picasa,Calendar,Android Market, AppEngine。在Spanner中需要支持SQL类型的查询语言，也很显然是非常必要的，因为Dremel[28]作为交互式分析工具已经非常普及。最后，在BigTable中跨行事务的缺乏来导致了

用户频繁的抱怨；Percolator[32]的开发就是用来部分解决这个问题的。一些作者都在抱怨，通用的两阶段提交的代价过于昂贵，因为它会带来可用性问题和性能问题[9][10][19]。我们认为，最好让应用程序开发人员来处理由于过度使用事务引起的性能问题，而不是总是围绕着“缺少事务”进行编程。在Paxos上运行两阶段提交弱化了可用性问题。

应用的数据模型是架构在被目录桶装的键值映射层之上。一个应用会在一个universe中创建一个或者多个数据库。每个数据库可以包含无限数量的模式化的表。每个表都和关系数据库表类似，具备行、列和版本值。我们不会详细介绍Spanner的查询语言，它看起来很像SQL，只是做了一些扩展。

Spanner的数据模型不是纯粹关系型的，它的行必须有名称。更准确地说，每个表都需要有包含一个或多个主键列的排序集合。这种需求，让Spanner看起来仍然有点像键值存储：主键形成了一个行的名称，每个表都定义了从主键列到非主键列的映射。当一个行存在时，必须要求已经给行的一些键定义了一些值（即使是NULL）。采用这种结构是很有用的，因为这可以让应用通过选择键来控制数据的局部性。

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

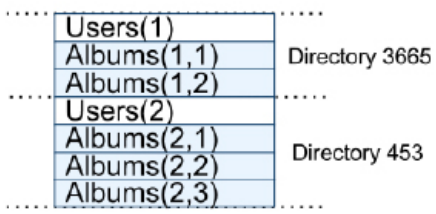

Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by INTERLEAVE IN.

图4包含了一个Spanner模式的实例，它是以每个用户和每个相册为基础存储图片元数据。这个模式语言和Megastore的类似，同时增加了额外的要求，即每个Spanner数据库必须被客户端分割成一个或多个表的层次结构（hierarchy）。客户端应用会使用INTERLEAVE IN语句在数据库模式中声明这个层次结构。这个层次结构上面的表，是一个目录表。目录表中的每行都具有键K，和子孙表中的所有以K开始（以字典顺序排序）的行一起，构成了一个目录。ON DELETE CASCADE意味着，如果删除目录中的一个行，也会级联删除所有相关的子孙行。这个图也解释了这个实例数据库的交织层次（interleaved layout），例如Albums(2,1)代表了来自Albums表的、对应于user_id=2和album_id=1的行。这种表的交织层次形成目录，是非常重要的，因为它允许客户端来描述存在于多个表之间的位置关系，这对于一个分片的分布式数据库的性能而言是很重要的。没有它的话，Spanner就无法知道最重要的位置关系。

3 TrueTime

Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(*t*)	true if *t* has definitely passed
TT.before(*t*)	true if *t* has definitely not arrived

Table 1: TrueTime API. The argument *t* is of type *TTstamp*.

本部分内容描述TrueTime API，并大概给出它的实现方法。我们把大量细节内容放在另一篇论文中，我们的目标是展示这种API的力量。表1列出了API的方法。TrueTime会显式地把时间表达成TTinterval，这是一个时间区间，具有有界限的时间不确定性（不像其他的标准时间接口，没有为客户端提供“不确定性”这种概念）。TTinterval区间的端点是TTstamp类型。TT.now()方法会 返回一个TTinterval，它可以保证包含TT.now()方法在调用时的绝对时间。这个时间和具备闰秒涂抹（leap-second smearing）的UNIX时间一样。把即时误差边界定义为ε,平均误差边界为（林子雨注：“ε上边一个横杠”，表示平均值）。TT.after()和TT.before()方法是针对TT.now()的便捷的包装器。

表示一个事件e的绝对时间，可以利用函数$t_{abs}(e)$。如果用更加形式化的术语，TrueTime可以保证，对于一个调用tt=TT.now()，有$tt.earliest \le t_{abs}(e_{now}) \le tt.latest$，其中，是$e_{now}$是调用的事件。

在底层，TrueTime使用的时间是GPS和原子钟。TrueTime使用两种类型的时间，是因为它们有不同的失败模式。GPS参考时间的弱点是天线和接收器失效、局部电磁干扰和相关失败（比如设计上的缺陷导致无法正确处理闰秒和电子欺骗），以及GPS系统运行中断。原子钟也会失效，不过失效的方式和GPS无关，不同原子钟之间的失效也没有彼此关联。由于存在频率误差，在经过很长的时间以后，原子钟都会产生明显误差。

TrueTime是由每个数据中心上面的许多time master机器和每台机器上的一个timeslave daemon来共同实现的。大多数master都有具备专用天线的GPS接收器，这些master在物理上是相互隔离的，这样可以减少天线失效、电磁干扰和电子欺骗的影响。剩余的master（我们称为Armageddon master）则配备了原子钟。一个原子钟并不是很昂贵：一个Armageddon master的花费和一个GPS master的花费是同一个数量级的。所有master的时间参考值都会进行彼此校对。每个master也会交叉检查时间参考值和本地时间的比值，如果二者差别太大，就会把自

已驱逐出去。在同步期间，Armageddon master会表现出一个逐渐增加的时间不确定性，这是由保守应用的最差时钟漂移引起的。GPS master表现出的时间不确定性几乎接近于0。

每个daemon会从许多master[29]中收集投票，获得时间参考值，从而减少误差。被选中的master中，有些master是GPS master，是从附近的数据中心获得的，剩余的GPS master是从远处的数据中心获得的；还有一些是Armageddon master。Daemon会使用一个Marzullo算法[27]的变种，来探测和拒绝欺骗，并且把本地时钟同步到非撒谎master的时间参考值。为了免受较差的本地时钟的影响，我们会根据组件规范和运行环境确定一个界限，如果机器的本地时钟误差频繁超出这个界限，这个机器就会被驱逐出去。

在同步期间，一个daemon会表现出逐渐增加的时间不确定性。ε是从保守应用的最差时钟漂移中得到的。ε也取决于time master的不确定性，以及与time master之间的通讯延迟。在我们的线上应用环境中，ε通常是一个关于时间的锯齿形函数。在每个投票间隔中，ε会在1到7ms之间变化。因此，在大多数情况下，ε的平均值是4ms。Daemon的投票间隔，在当前是30秒，当前使用的时钟漂移比率是200微秒/秒，二者一起意味着0到6ms的锯齿形边界。剩余的1ms主要来自到time master的通讯延迟。在失败的时候，超过这个锯齿形边界也是有可能的。例如，偶尔的time master不确定性，可能会引起整个数据中心范围内的ε值的增加。类似的，过载的机器或者网络连接，都会导致ε值偶尔地局部增大。

4. 并发控制

本部分内容描述TrueTime如何可以用来保证并发控制的正确性，以及这些属性如何用来实现一些关键特性，比如外部一致性的事务、无锁机制的只读事务、针对历史数据的非阻塞读。这些特性可以保证，在时间戳为t的时刻的数据库读操作，一定只能看到在t时刻之前已经提交的事务。

进一步说，把Spanner客户端的写操作和Paxos看到的写操作这二者进行区分，是非常重要的，我们把Paxos看到的写操作称为Paxos写操作。例如，两阶段提交会为准备提交阶段生成一个Paxos写操作，这时不会有相应的客户端写操作。

4.1 时间戳管理

表2列出了Spanner支持的操作的类型。Spanner可以支持读写事务、只读事务（预先声明的快照隔离事务）和快照读。独立写操作，会被当成读写事务来执行。非快照独立读操作，会被当成只读事务来执行。二者都是在内部进行retry，客户端不用进行这种retry loop。

Operation	Timestamp Discussion	Concurrency Control	Replica Required
Read-Write Transaction	§ 4.1.2	pessimistic	leader
Read-Only Transaction	§ 4.1.4	lock-free	leader for timestamp; any for read, subject to § 4.1.3
Snapshot Read, client-provided timestamp	—	lock-free	any, subject to § 4.1.3
Snapshot Read, client-provided bound	§ 4.1.3	lock-free	any, subject to § 4.1.3

Table 2: Types of reads and writes in Spanner, and how they compare.

一个只读事务具备快照隔离的性能优势[6]。一个只读事务必须事先被声明不会包含任何写操作，它并不是一个简单的不包含写操作的读写事务。在一个只读事务中的读操作，在执行时会采用一个系统选择的时间戳，不包含锁机制，因此，后面到达的写操作不会被阻塞。在一个只读事务中的读操作，可以到任何足够新的副本上去执行（见第4.1.3节）。

一个快照读操作，是针对历史数据的读取，执行过程中，不需要锁机制。一个客户端可以为快照读确定一个时间戳，或者提供一个时间范围让Spanner来自动选择时间戳。不管是哪种情况，快照读操作都可以在任何具有足够新的副本上执行。

对于只读事务和快照读而言，一旦已经选定一个时间戳，那么，提交就是不可避免的，除非在那个时间点的数据已经被垃圾回收了。因此，客户端不必在retry loop中缓存结果。当一个服务器失效的时候，客户端就可以使用同样的时间戳和当前的读位置，在另外一个服务器上继续执行读操作。

4.2 细节

林子雨注：上面是Google Spanner（中文版）的核心内容，第4节“并发控制”的剩余内容，没有在网页中给出，而是放在PDF文件中（请到本网页的底部“附件”中下载PDF文件），因为，第4节“并发控制”的剩余内容，公式太多，无法放入网页。而且，根据本人的阅读，上述给出的内容已经可以帮助读者基本了解Google Spanner的概貌，剩余的内容是一些琐碎的细节，个人感觉读起来比较晦涩，如果不是深入研究需要，可以不用继续阅读第4节“并发控制”的剩余内容。

5. 实验分析

我们对Spanner性能进行了测试，包括复制、事务和可用性。然后，我们提供了一些关于TrueTime的实验数据，并且提供了我们的第一个用例——F1。

5.1 微测试基准

表3给出了一个用于Spanner的微测试基准(microbenchmark)。这些测试是在分时机器上实现的：每个spanserver采用4GB内存和四核CPU（AMD Barcelona 2200MHz）。客户端运行在单独的机器上。每个zone都包含一个spanserver。客户端和zone都放在一个数据中心集合内，它们之间的网络距离不会超过1ms。（这种布局是很普通的，许多数据并不需要把数据分散存储到全球各地）。测试数据库具有50个Paxos组和2500个目录。操作都是独立的4KB大小的读和写。All reads were served out of memory after a compaction，从而使得我们只需要衡量Spanner调用栈的开销。此外，还会进行一轮读操作，来预热任何位置的缓存。

replicas	latency (ms)			throughput (Kops/sec)		
	write	read-only transaction	snapshot read	write	read-only transaction	snapshot read
1D	9.4±.6	—	—	4.0±.3	—	—
1	14.4±1.0	1.4±.1	1.3±.1	4.1±.05	10.9±.4	13.5±.1
3	13.9±.6	1.3±.1	1.2±.1	2.2±.5	13.8±3.2	38.5±.3
5	14.4±.4	1.4±.05	1.3±.04	2.8±.3	25.3±5.2	50.0±1.1

Table 3: Operation microbenchmarks. Mean and standard deviation over 10 runs. 1D means one replica with commit wait disabled.

对于延迟实验而言，客户端会发起足够少量的操作，从而避免在服务器中发生排队。从1个副本的实验中，提交等待大约是5ms，Paxos延迟大约是9ms。随着副本数量的增加，延迟大约保持不变，只具有很少的标准差，因为在一个组的副本内，Paxos会并行执行。随着副本数量的增加，获得指定投票数量的延迟，对一个slave副本的慢速度不会很敏感。

对于吞吐量的实验而言，客户端发起足够数量的操作，从而使得CPU处理能力达到饱和。快照读操作可以在任何足够新的副本上进行，因此，快照读的吞吐量会随着副本的数量增加而线性增加。单个读的只读事务，只会在领导者上执行，因为，时间戳分配必须发生在领导者上。只读事务吞吐量会随着副本数量的增加而增加，因为有效的spanserver的数量会增加：在这个实验的设置中，spanserver的数量和副本的数量相同，领导者会被随机分配到不同的zone。写操作的吞吐量也会从这种实验设置中获得收益（副本从3变到5时写操作吞吐量增加了，就能够说明这点），但是，随着副本数量的增加，每个写操作执行时需要完成的工作量也会线性增加，这就会抵消前面的收益。

表4显示了两阶段提交可以扩展到合理数量的参与者：它是对一系列实验的总结，这些实验运行在3个zone上，每个zone具有25个spanserver。扩展到50个参与者，无论在平均值还是第99个百分位方面，都是合理的。在100个参与者的情形下，延迟开始明显增加。

participants	latency (ms)	
	mean	99th percentile
1	17.0 ±1.4	75.0 ±34.9
2	24.5 ±2.5	87.6 ±35.9
5	31.5 ±6.2	104.5 ±52.2
10	30.0 ±3.7	95.6 ±25.4
25	35.5 ±5.6	100.4 ±42.7
50	42.7 ±4.1	93.7 ±22.9
100	71.4 ±7.6	131.2 ±17.6
200	150.5 ±11.0	320.3 ±35.1

Table 4: Two-phase commit scalability. Mean and standard deviations over 10 runs.

5.2 可用性

图5显示了在多个数据中心运行Spanner时的可用性方面的收益。它显示了三个吞吐量实验的结果，并且存在数据中心失败的情形，所有三个实验结果都被重叠放置到一个时间轴上。测试用的universe包含5个zone Zi，每个zone都拥有25个spanserver。测试数据库被分片成1250个Paxos组，100个客户端不断地发送非快照读操作，累积速率是每秒50K个读操作。所有领导者都会被显式地放置到Z1。每个测试进行5秒钟以后，一个zone中的所有服务器都会被“杀死”：non-leader杀掉Z2，leader-hard杀掉Z1，leader-soft杀掉Z1，但是，它会首先通知所有服务器它们将要交出领导权。

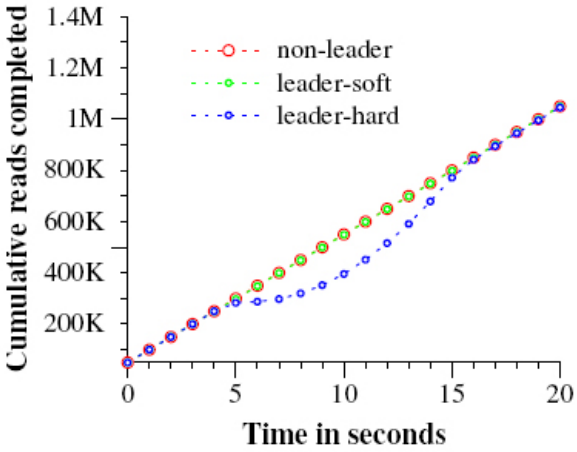

Figure 5: Effect of killing servers on throughput.

杀掉Z2对于读操作吞吐量没有影响。杀掉Z1，给领导者一些时间来把领导权交给另一个zone时，会产生一个小的影响：吞吐量会下降，不是很明显，大概下降3-4%。另一方面，没有预警就杀掉Z1有一个明显的影响：完成率几乎下降到0。随着领导者被重新选择，系统的吞吐量会增加到大约每秒100K个读操作，主要是由于我们的实验设置：系统中有额外的能力，当找不到领导者时操作会排队。由此导致的结果是，系统的吞吐量会增加直到到达系统恒定的速率。

我们可以看看把Paxos领导者租约设置为10s的效果。当我们杀掉这个zone，对于这个组的领导者租约的过期时间，会均匀地分布到接下来的10秒钟内。来自一个死亡的领导者的每个租约一旦过期，就会选择一个新的领导者。大约在杀死时间过去10秒钟以后，所有的组都会有领导者，吞吐量就恢复了。短的租约时间会降低服务器死亡对于可用性的影响，但是，需要更多的更新租约的网络通讯开销。我们正

在设计和实现一种机制，它可以在领导者失效的时候，让slave释放Paxos领导者租约。

5.3 TrueTime

关于TrueTime，必须回答两个问题：ε是否就是时钟不确定性的边界？ε会变得多糟糕？对于第一个问题，最严峻的问题就是，如果一个局部的时钟漂移大于200us/sec，那就会破坏TrueTime的假设。我们的机器统计数据显示，坏的CPU的出现概率要比坏的时钟出现概率大6倍。也就是说，与更加严峻的硬件问题相比，时钟问题是很少见的。由此，我们也相信，TrueTime的实现和Spanner其他软件组件一样，具有很好的可靠性，值得信任。

图6显示了TrueTime数据，是从几千个spanserver中收集的，这些spanserver跨越了多个数据中心，距离2200公里以上。图中描述了ε的第90个、99个和99.9个百分位的情况，是在对timemaster进行投票后立即对timeslave daemon进行样本抽样的。这些抽样数据没有考虑由于时钟不确定性带来的ε值的锯齿，因此测量的是timemaster不确定性（通常是0）再加上通讯延迟。

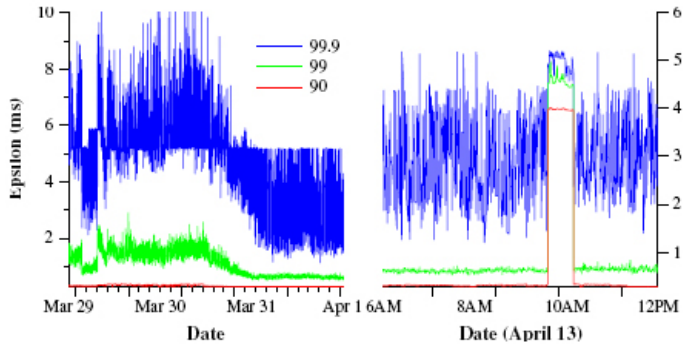

Figure 6: Distribution of TrueTime ϵ values, sampled right after timeslave daemon polls the time masters. 90th, 99th, and 99.9th percentiles are graphed.

图6中的数据显示了，在决定ε的基本值方面的上述两个问题，通常都不会是个问题。但是，可能会存在明显的拖尾延迟问题，那会引起更高的ε值。图中，3月30日拖尾延迟的降低，是因为网络的改进，减少了瞬间网络连接的拥堵。在4月13日ε的值增加了，持续了大约1个小时，主要是因为例行维护时关闭了两个time master。我们会继续调研并且消除引起TrueTime突变的因素。

5.4 F1

Spanner在2011年早期开始进行在线负载测试，它是作为谷歌广告后台F1[35]的重新实现的一部分。这个后台最开始是基于MySQL数据库，在许多方面都采用手工数据分区。未经压缩的数据可以达到几十TB，虽然这对于许多NoSQL实例而言数据量是很小的，但是，对于采用数据分区的MySQL而言，数据量是非常大的。MySQL的数据分片机制，会把每个客户和所有相关的数据分配给一个固定的分区。这种布局方式，可以支持针对单个客户的索引构建和复杂查询处理，但是，需要了解一些商业知识来设计分区。随着客户数量的增长，对数据进行重新分区，代价是很大的。最近一次的重新分区，花费了两年的时间，为了降低风险，在多个团队之间进行了大量的合作和测试。这种操作太复杂了，无法常常执行，由此导致的结果是，团队必须限制MySQL数据库的增长，方法是，把一些数据存储在外部的Bigtable中，这就会牺牲事务和查询所有数据的能力。

F1团队选择使用Spanner有几个方面的原因。首先，Spanner不需要手工分区。其次，Spanner提供了同步复制和自动失败恢复。在采用MySQL的master-slave复制方法时，很难进行失败恢复，会有数据丢失和当机的风险。再次，F1需要强壮的事务语义，这使得使用其他NoSQL系统是不实际的。应用语义需要跨越任意数据的事务和一致性读。F1团队也需要在他们的数据上构建二级索引（因为Spanner没有提供对二级索引的自动支持），也有能力使用Spanner事务来实现他们自己的一致性全球索引。

所有应用写操作，现在都是默认从F1发送到Spanner。而不是发送到基于MySQL的应用栈。F1在美国的西岸有两个副本，在东岸有三个副本。这种副本位置的选择，是为了避免发生自然灾害时出现服务停止问题，也是出于前端应用的位置的考虑。实际上，Spanner的失败自动恢复，几乎是不可见的。在过去的几个月中，尽管有不在计划内的机群失效，但是，F1团队最需要做的工作仍然是更新他们的数据库模式，来告诉Spanner在哪里放置Paxos领导者，从而使得它们尽量靠近应用前端。

Spanner时间戳语义，使得它对于F1而言，可以高效地维护从数据库状态计算得到的、放在内存中的数据结构。F1会为所有变更都维护一个逻辑历史日志，它会作为每个事务的一部分写入到Spanner。F1会得到某个时间戳下的数据的完整快照，来初始化它的数据结构，然后根据数据的增量变化来更新这个数据结构。

表5显示了F1中每个目录的分片数量的分布情况。每个目录通常对应于F1上的应用栈中的一个用户。绝大多数目录（同时意味着绝大多数用户）都只会包含一个分片，这就意味着，对于这些用户数据的读和写操作只会发生在一个服务器上。多于100个分片的目录，是那些包含F1二级索引的表：对这些表的多个分片进行写操作，是极其不寻常的。F1团队也只是在以事务的方式进行未经优化的批量数据加载时，才会碰到这种情形。

# fragments	# directories
1	>100M
2–4	341
5–9	5336
10–14	232
15–99	34
100–500	7

Table 5: Distribution of directory-fragment counts in F1.

表6显示了从F1服务器来测量的Spanner操作的延迟。在东海岸数据中心的副本，在选择Paxos领导者方面会获得更高的优先级。表6中的数据是从这些数据中心的F1服务器上测量得到的。写操作延迟分布上存在较大的标准差，是由于锁冲突引起的肥尾效应（fat tail）。在读操作延迟分布上存在更大的标准差，部分是因为Paxos领导者跨越了两个数据中心，只有其中的一个是采用了固态盘的机器。此外，测试内容还包括系统中的每个针对两个数据中心的读操作：字节读操作的平均值和标准差分别是1.6KB和119KB。

operation	latency (ms)		count
	mean	std dev	
all reads	8.7	376.4	21.5B
single-site commit	72.3	112.8	31.2M
multi-site commit	103.0	52.2	32.1M

Table 6: F1-perceived operation latencies measured over the course of 24 hours.

6. 相关工作

Megastore[5]和DynamoDB[3]已经提供了跨越多个数据中心的一致性复制。DynamoDB提供了键值存储接口，只能在一个region内部进行复制。Spanner和Megastore一样，都提供了半关系数据模型，甚至采用了类似的模式语言。Megastore无法获得高性能。Megastore是架构在Bigtable之上，这带来了很高的通讯代价。Megastore也不支持长寿命的领导者，多个副本可能会发起写操作。来自不同副本的写操作，在Paxos协议下一定会发生冲突，即使他们不会发生逻辑冲突：会严重影响吞吐量，在一个Paxos组内每秒钟只能执行几个写操作。Spanner提供了更高的性能，通用的事务和外部一致性。

Pavlo等人[31]对数据库和MapReduce[12]的性能进行了比较。他们指出了几个努力的方向，可以在分布式键值存储之上充分利用数据库的功能[1][4][7][41]，二者可以实现充分的融合。我们比较赞同这个结论，并且认为集成多个层是具有优势的：把复制和并发控制集成起来，可以减少Spanner中的提交等待代价。

在一个采用了复制的存储上面实现事务，可以至少追述到Gifford的论文[16]。Scatter[17]是一个最近的基于DHT的键值存储，可以在一致性复制上面实现事务。Spanner则要比Scatter在更高的层次上提供接口。Gray和Lamport[18]描述了一个基于Paxos的非阻塞的提交协议，他们的协议会比两阶段提交协议带来更多的代价，而两阶段提交协议在大范围分布式的组中的代价会进一步恶化。Walter[36]提供了一个快照隔离的变种，但是无法跨越数据中心。相反，我们的只读事务提供了一个更加自然的语义，因为，我们对于所有的操作都支持外部语义。

最近，在减少或者消除锁开销方面已经有大量的研究工作。Calvin[40]消除了并发控制：它会重新分配时间戳，然后以时间戳的顺序执行事务。HStore[39]和Granola[11]都支持自己的事务类型划分方法，有些事务类型可以避免锁机制。但是，这些系统都无法提供外部一致性。Spanner通过提供快照隔离，解决了冲突问题。

VoltDB[42]是一个分片的内存数据库，可以支持在大范围区域内进行主从复制，支持灾难恢复，但是没有提供通用的复制配置方法。它是一个被称为NewSQL的实例，这是实现可扩展的SQL[38]的强大的市场推动力。许多商业化的数据库都可以支持历史数据读取，比如Marklogic[26]和Oracle' Total Recall[30]。Lomet和Li[24]对于这种时间数据库描述了一种实现策略。

Faresite给出了与一个受信任的时钟参考值相关的、时钟不确定性的边界[13]（要比TrueTime更加宽松）：Farsite中的服务器租约的方式，和Spanner中维护Paxos租约的方式相同。在之前的工作中[2][23]，宽松同步时钟已经被用来进行并发控制。我们已经展示了TrueTime可以从Paxos状态机集合中推导出全球时间。

7. 未来的工作

在过去一年的大部分时间里，我们都是F1团队一起工作，把谷歌的广告后台从MySQL迁移到Spanner。我们正在积极改进它的监控和支撑工具，同时在优化性能。此外，我们已经开展了大量工作来改进备份恢复系统的功能和性能。我们当前正在实现Spanner模式语言，自动维护二级索引和自动基于负载的分区。在未来，我们会调研更多的特性。以最优化的方式并行执行读操作，是我们追求的有价值的策略，但是，初级阶段的实验表明，实现这个目标比较艰难。此外，我们计划最终可以支持直接变更Paxos配置[22][34]。

我们希望许多应用都可以跨越数据中心进行复制，并且这些数据中心彼此靠近。TrueTime ε可能会明显影响性能。把ε值降低到1ms以内，并不存在不可克服的障碍。Time-master-query间隔可以继续减少，Time-master-query延迟应该随着网络的改进而减少，或者通过采用分时技术来避免延迟。

最后，还有许多有待改进的方面。尽管Spanner在节点数量上是可扩展的，但是与节点相关的数据结构在复杂的SQL查询上的性能相对

较差，因为，它们是被设计成服务于简单的键值访问的。来自数据库文献的算法和数据结构，可以极大改进单个节点的性能。另外，根据客户端负载的变化，在数据中心之间自动转移数据，已经成为我们的一个目标，但是，为了有效实现这个目标，我们必须具备在数据中心之间自动、协调地转移客户端应用进程的能力。转移进程会带来更加困难的问题——如何在数据中心之间管理和分配资源。

8. 总结

总的来说，Spanner对来自两个研究群体的概念进行了结合和扩充：一个是数据库研究群体，包括熟悉易用的半关系接口，事务和基于SQL的查询语言；另一个是系统研究群体，包括可扩展性，自动分区，容错，一致性复制，外部一致性和大范围分布。自从Spanner概念成形，我们花费了5年以上的时间来完成当前版本的设计和实现。花费这么长的时间，一部分原因在于我们慢慢意识到，Spanner不应该仅仅解决全球复制的命名空间问题，而且也应该关注Bigtable中所丢失的数据库特性。

我们的设计中一个亮点特性就是TrueTime。我们已经表明，在时间API中明确给出时钟不确定性，可以以更加强壮的时间语义来构建分布式系统。此外，因为底层的系统在时钟不确定性上采用更加严格的边界，实现更强壮的时间语义的代价就会减少。作为一个研究群体，我们在设计分布式算法时，不再依赖于弱同步的时钟和较弱的时间API。

致谢

许多人帮助改进了这篇论文：Jon Howell，Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. 我们的管理层对于我们的工作和论文发表都非常支持：Aristotle Balogh, Bill Coughran, Urs H¨olzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman.

我们的工作是在Bigtable和Megastore团队的工作基础之上开展的。F1团队，尤其是Jeff Shute，和我们一起工作，开发了数据模型，跟踪性能和纠正漏洞。Platforms团队，尤其是Luiz Barroso和Bob Felderman，帮助我们一起实现了TrueTime。最后，许多谷歌员工都曾经在我们的团队工作过，包括Ken Ashcraft, Paul Cychosz, Krzysztof Ostrowski, Amir Voskoboynik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Suvorov, and Marcel van der Holst.

参考文献

[1] Azza Abouzeid et al. “HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads”. Proc. of VLDB. 2009, pp. 922–933.

[2] A. Adya et al. “Efficient optimistic concurrency control using loosely synchronized clocks”. Proc. of SIGMOD. 1995, pp. 23–34.

[3] Amazon. Amazon DynamoDB. 2012.

[4] Michael Armbrust et al. “PIQL: Success-Tolerant Query Processing in the Cloud”. Proc. of VLDB. 2011, pp. 181–192.

[5] Jason Baker et al. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. Proc. of CIDR. 2011, pp. 223–234.

[6] Hal Berenson et al. “A critique of ANSI SQL isolation levels”. Proc. of SIGMOD. 1995, pp. 1–10.

[7] Matthias Brantner et al. “Building a database on S3”. Proc. of SIGMOD. 2008, pp. 251–264.

[8] A. Chan and R. Gray. “Implementing Distributed Read-Only Transactions”. IEEE TOSE SE-11.2 (Feb. 1985), pp. 205–212.

[9] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. ACM TOCS 26.2 (June 2008), 4:1–4:26.

[10] Brian F. Cooper et al. “PNUTS: Yahoo!’s hosted data serving platform”. Proc. of VLDB. 2008, pp. 1277–1288.

[11] James Cowling and Barbara Liskov. “Granola: Low-Overhead Distributed Transaction Coordination”. Proc. of USENIX ATC. 2012, pp. 223–236.

[12] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: a flexible data processing tool”. CACM 53.1 (Jan. 2010), pp. 72–77.

[13] John Douceur and Jon Howell. Scalable Byzantine-Fault-Quantifying Clock Synchronization. Tech. rep. MSR-TR-2003-67. MS Research, 2003.

[14] John R. Douceur and Jon Howell. “Distributed directory service in the Farsite file system”. Proc. of OSDI. 2006, pp. 321–334.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system”. Proc. of SOSP. Dec. 2003, pp. 29–43.

[16] David K. Gifford. Information Storage in a Decentralized Computer System. Tech. rep. CSL-81-8. PhD dissertation. Xerox PARC, July 1982.

[17] Lisa Glendenning et al. “Scalable consistency in Scatter”. Proc. of SOSP. 2011.

[18] Jim Gray and Leslie Lamport. “Consensus on transaction commit”. ACM TODS 31.1 (Mar. 2006), pp. 133–160.

[19] Pat Helland. “Life beyond Distributed Transactions: an Apostate’s Opinion”. Proc. of CIDR. 2007, pp. 132–141.

[20] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects”. ACM TOPLAS 12.3 (July 1990), pp. 463–492.

[21] Leslie Lamport. “The part-time parliament”. ACM TOCS 16.2 (May 1998), pp. 133–169.

[22] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. “Reconfiguring a state machine”. SIGACT News 41.1 (Mar. 2010), pp. 63–73.

[23] Barbara Liskov. “Practical uses of synchronized clocks in distributed systems”. Distrib. Comput. 6.4 (July 1993), pp. 211–219.

[24] David B. Lomet and Feifei Li. “Improving Transaction-Time DBMS Performance and Functionality”. Proc. of ICDE (2009), pp. 581–591.

[25] Jacob R. Lorch et al. “The SMART way to migrate replicated stateful services”. Proc. of EuroSys. 2006, pp. 103–115.

[26] MarkLogic. MarkLogic 5 Product Documentation. 2012.

[27] Keith Marzullo and Susan Owicki. “Maintaining the time in a distributed system”. Proc. of PODC. 1983, pp. 295–305.

[28] Sergey Melnik et al. “Dremel: Interactive Analysis of Web-Scale Datasets”. Proc. of VLDB. 2010, pp. 330–339.

[29] D.L. Mills. Time synchronization in DCNET hosts. Internet Project Report IEN–173. COMSAT Laboratories, Feb. 1981.
[30] Oracle. Oracle Total Recall. 2012.
[31] Andrew Pavlo et al. “A comparison of approaches to large-scale data analysis”. Proc. of SIGMOD. 2009, pp. 165–178.
[32] Daniel Peng and Frank Dabek. “Large-scale incremental processing using distributed transactions and notifications”. Proc. of OSDI. 2010, pp. 1–15.
[33] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. “System level concurrency control for distributed database systems”. ACM TODS 3.2 (June 1978), pp. 178–198.
[34] Alexander Shraer et al. “Dynamic Reconfiguration of Primary/Backup Clusters”. Proc. of SENIX ATC. 2012, pp. 425–438.
[35] Jeff Shute et al. “F1—The Fault-Tolerant Distributed RDBMS Supporting Google’s Ad Business”. Proc. of SIGMOD. May 2012, pp. 777–778.
[36] Yair Sovran et al. “Transactional storage for geo-replicated systems”. Proc. of SOSP. 2011, pp. 385–400.
[37] Michael Stonebraker. Why Enterprises Are Uninterested in NoSQL. 2010.
[38] Michael Stonebraker. Six SQL Urban Myths. 2010.
[39] Michael Stonebraker et al. “The end of an architectural era: (it’s time for a complete rewrite)”. Proc. of VLDB. 2007, pp. 1150–1160.
[40] Alexander Thomson et al. “Calvin: Fast Distributed Transactions for Partitioned Database Systems”. Proc. of SIGMOD.2012, pp. 1–12.
[41] Ashish Thusoo et al. “Hive — A Petabyte Scale Data Warehouse Using Hadoop”. Proc. of ICDE. 2010, pp. 996–1005.
[42] VoltDB. VoltDB Resources. 2012.
（厦门大学计算机系数据库实验室教师 林子雨 翻译 2012年9月17日-21日）