电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS

论文题目 **乐观分布式事务与可拓展性验证**

专　　业 **计算机科学与技术**

学　　号 **2017020902022**

作者姓名 **杨韬**

指导教师 聂晓文　副教授

摘　要

随着互联网数据量爆发式增长，传统数据库慢慢被 NoSQL 的分布式数据库取代。但是 NoSQL 数据库不具备事务处理的能力，这使得它们在处理复杂业务逻辑时捉襟见肘，难以保证数据库的一致性。如今分布式数据库正在从 NoSQL 发展到 NewSQL 的阶段，而分布式事务处理是 NewSQL 的关键技术之一。

本文旨在为 HBase 提供可串行化的分布式事务处理支持，并且不能以牺牲系统的可拓展性为代价，最终使得 HBase 能够同时胜任 OLTP（On-Line Transaction Processing）和 OLAP（On-Line Analytical Processing）的业务场景。

在比较了各种乐观并发控制算法的原理后，本文设计了基于多版本时间戳排序（Multi-Version Timestamp Ordering）和写快照隔离（Write-Snapshot Isolation）的事务处理系统 HBS。与其他算法相比，多版本时间戳排序算法不依赖中心节点检测冲突，使得 HBS 的可拓展性良好。写快照隔离避免了不必要的冲突检测，进一步提高了 HBS 的并发度。相比于 Percolator，Omid 等系统只提供快照隔离级别，HBS 的策略提供了可串行的隔离级别，并且不以牺牲系统的可拓展性和性能为代价。

最后本文从数据一致性的角度测试并验证了系统的正确性，并讨论了该系统的可拓展性。

关键词：可串行化，多版本时间戳排序，乐观并发控制，写快照隔离，分布式事务处理，分布式数据库

ABSTRACT

With the explosive growth of Internet data, traditional databases are slowly being replaced by NoSQL distributed databases. The lack of transaction processing of NoSQL databases makes them stretched when dealing with complex application logic. Nowadays, distributed databases are turning from NoSQL to NewSQL, and transaction processing is one of the keys to NewSQL.

The paper aims to provide transaction processing support for HBase without sacrificing the scalability of the system and ultimately enables HBase to be competent for both OLTP (On-Line Transaction Processing) and OLAP (On-Line Analytical Processing).

The paper designs and implements the transaction processing system HBS based on HBase. After comparing different optimistic concurrency control algorithms, HBS chooses a concurrency control strategy of multi-version timestamp ordering and writ-snapshot isolation. Multi-version timestamp ordering does not rely on a central node to detect conflicts, making HBS have good scalability. Write-snapshot isolation reduces the occurrence of conflicts and further boosts HBS up. Compared to Percolator, Omid systems providing only snapshot isolation level, HBS strategy provides a serializable isolation level, and not at the expense of scalability and system performance for the price.

Finally, the paper tests and verifies the correctness of HBS from the perspective of consistency, and discusses the scalability of HBS.

Keywords: serializability, multi-version timestamp ordering, optimistic concurrent control, write-snapshot isolation, distributed transaction processing, distributed database management system

目 录

第一章 绪 论

1.1 研究背景

随着大数据时代的到来，传统的关系数据库没有能力处理如此巨量的数据。NoSQL 数据库正是为了解决这类问题而诞生的，它们常常以 Key-Value 为数据模型，具有非常好的可拓展性，能够同时部署在成千上万台服务器上，处理 PB 级数据量。

但是，在 NoSQL 数据库的实现中，为了良好的可拓展性，它们常常会放弃对事务处理的完整支持。例如 HBase[1]只支持 CheckAndMutate 原子操作，Bigtable[2]只提供行内事务支持。这使得它们在处理复杂业务逻辑时显得力不从心。传统的业务也难以平滑地迁移到 NoSQL 的数据库中。

为了解决这个问题，NewSQL，一类支持传统数据库接口的分布式数据库应运而生。它们为应用层提供的最关键的支持是分布式事务处理。

1.2 研究现状

在 NewSQL 的发展过程中出现了多个分布式数据库系统支持事务处理，下面简要介绍。

Percolator[3]基于 Bigtable 使用两阶段提交的算法实现了快照隔离级别。Percolator 使用一个 Timestamp Oracle 为事务分配开始时戳和提交时戳，并在事务提交的第一阶段判断是否与并发事务发生冲突。两阶段提交的策略使得 Percolator 具有很好的可拓展性，但是同时也带来了较高的延迟。

Omid[5]为 HBase 提供快照隔离级别的事务支持。不同于 Percolator 使用两阶段提交的策略，Omid 利用一个主节点保存事务执行状态，检查事务的提交。Omid 中心化冲突检查的策略避免了两阶段提交的高延迟，但是同时也使得主节点成为系统可拓展性的瓶颈。

值得注意的是，它们都只提供了快照隔离级别的事务支持。在某些业务场景之下，快照隔离并不能保证数据的一致性，可能会出现写偏序的情况。

1.3 本文的贡献与创新

设计并实现一个分布式数据库是非常复杂的，我们常常要在各种利弊得失之间做最恰当的取舍。现有的事务处理系统要么牺牲隔离级别来满足高并发的要求，

例如 Percolator；要么牺牲可拓展性来满足低延时的要求，例如 Omid。我们必须要根据我们的需求做合理的取舍。

本论文将要介绍的 HBS 系统基于 HBase 提供了可串行化的事务支持。HBS 使用 MVTO[6]乐观并发控制算法，两阶段提交的策略，实现了非常好的可拓展性。本文还学习了 CockroachDB[14]的 write intent 的概念，进一步优化两阶段提交的延迟。同时，HBS 应用了写快照隔离（write-snapshot isolation）[8]优化，提升系统的吞吐量。相比于 Percolator，Omid 不仅提供了更高的隔离等级，而且并不以牺牲巨大的系统性能和可拓展性为代价。

1.4 本文的结构安排

在第二章中，我们会讨论各种并发控制算法和事务的隔离级别等相关技术基础，并给出 HBS 使用 MVTO 算法的原因及其取舍。第三章中，我们会讨论 HBS 的需求分析。第四章将根据需求分析的结果，给出每个模块的设计。第五章将在设计的基础上实现系统，我们会详细地介绍 HBS 的各个组件的具体实现，除此之外我们还会论证 HBS 的错误恢复的能力。第六章中，我们会介绍一些对 HBS 的系统测试。

第二章 相关技术基础

2.1 事务

事务是对一组相关的数据库操作的抽象。一个事务处理系统提供这些保证：事务的执行是原子的；多个事务并行执行结果等价于它们串行执行的结果；数据库在事务执行的前后从一个一致状态转移到另一个一致状态；已提交的事务对数据库的更改是持久的。这也是所谓的事务的 ACID 特性[18]。

ACID 特性让事务成为简化编程，推理系统行为的强大抽象。使用事务，程序员可以不用考虑操作的原子性、数据的一致性，只需要把业务逻辑封装在事务之下，就能得到 ACID 的保证。事务之于数据库处理系统就如同进程之于操作系统。

事务可以分为两个关键的部分：并发控制和错误恢复。并发控制是指在多个事务并发执行的情况下，保证数据的一致性不被破坏，它们的执行结果应等价于多个事务串行的执行结果，也就是所谓的可串行化。错误恢复指的是在数据库系统崩溃重启后，数据能够恢复到一致性的状态，已经提交的事务的结果不会丢失。

2.2 隔离级别

从理论上讲，我们希望看到的是对于各个事务而言好像是执行在一个独立的数据库系统中。其达到的效果是多个并发事务的执行结果等价于这些事务串行执行的结果，并且每个事务所看到的数据库都满足一致性约束。这种要求被称为可串行化。

表 2-1 隔离级别及它们可能出现的异常

隔离级别	脏读	不可重复读	幻影读	写偏序
未提交读（Read Uncommitted）	可能发生	可能发生	可能发生	可能发生
提交读（Read Committed）	不可能	可能发生	可能发生	可能发生
可重复读（Repeatable Read ）	不可能	不可能	可能发生	可能发生
快照（Snapshot）	不可能	不可能	不可能	可能发生
可序列化（Serializable）	不可能	不可能	不可能	不可能

然而，在实现的数据库系统时，要实现可串行化的基本是非常高的，可能会极大地限制事务的并发度。并且，并不是所有的业务都需要可串行这样最强的隔离保证。因此，我们根据事务并发执行时相互干扰的程度设定了不同的隔离基本。论文[9]中讨论了 ANSI SQL 标准中定义的隔离级别和常见的隔离级别。通常而言，我们会使用到中表 2-1 列出的隔离级别。但是并不是所有系统中都实现了这些隔离级别，例如 PostgreSQL 最低的隔离级别是读已提交，它早期最高的隔离级别是快照，在后期的改进中使用 SSI（Serializable Snapshot Isolation）算法提供了可串行化的隔离级别[10]。

2.3 并发控制

通常而言，一个数据库系统中，同时有多个事务并发地执行。并发控制就是要在允许事务并发执行的基础上，保证并发的事务不会相互影响，不会导致数据的不一致。从用户的角度而言，并发控制的目的是让所有任务像是串行执行的一样，而且一个事务执行过程中好像是独占了整个数据库系统。

我们可以把并发控制算法分为两类：乐观并发控制与悲观并发控制。悲观并发控制一般使用两阶段锁的策略保证一个事务在其执行过程中，它所访问的数据不被其他事务并发地修改。乐观的并发控制会假设一个事务在执行的过程中不会发生冲突，而在事务提交时检查是否有冲突发生，如果有再来解决冲突。乐观的策略会等待冲突出现以后再处理，而悲观的策略会避免冲突的出现。

实现悲观并发控制需要维护一张锁表记录每个事务访问的数据项。如果我们把悲观的协议应用到分布式系统中，我们就需要一张分布式的表。然而要正确地获取和释放分布式锁是一件非常复杂的事。例如锁表如何被持久化，一个事务崩溃后如何保证它获取的锁也被正确地释放。另外分布式场景之下，锁的获取和释放延迟代价也比较高。

2.3.1 多版本并发控制

乐观的策略有很多不同的实现，但是它们通常都会结合 MVCC[20]的思想。MVCC 是多版本并发控制，它并不是一种具体的并发控制算法，它是一种思路。MVCC 的系统中，一个逻辑数据项有多个物理版本。不同的物理版本表示一个数据库不同时段对应的状态。例如表 2-2 MVCC 示例，在 0 到 10 时段，这个数据项的值是 v1；10 到 14 时段，值是 v2；14 之后，值是 v3。这意味着一个读操作可以读到它感兴趣的版本，而不被当前的另一个写操作影响。两个写操作也可以写各自的版本，相互之间没有影响。

表 2-2 MVCC 示例

value	start-time	end-time
v1	0	10
v2	10	14
v3	14	

在 MVCC 的系统中，一个事务根据自己的时间戳，访问在这个时间戳之下对自己可见物理版本。这种增加物理版本的策略增强了事务依据时间戳的隔离性，极大地增加了事务的并发度。

2.3.2 乐观并发控制算法

虽然限定了我们的并发控制算法是乐观的，在并发控制的实现上，有各种各样的可供选择的方案。下面的策略是单机数据库系统中常见的一些策略，分布式并发控制的算法也是在单机的基础上拓展而来的。

OCC（Optimistic Concurrency Control）[12]是一种经典乐观的策略，它假设数据库中的重复发生的可能性低，因此没有必要在读写时获取锁。但是，我们会在事务提交的阶段检测事务之间是否有冲突发生。这种做法要求我们维护全局的事务状态，并且在事务提交的时候阻塞地检测是否有冲突发生。

MVTO（Multi-Version Timestamp Ordering）[6]是一个根据事务时戳计算它们可串行化顺序的并发控制算法。它同样不需要获取锁，在冲突发生以后再处理。MVTO 下，一个数据项会两个时间戳，读时戳和写时戳。每个事务在开始的时候都会被分配一个时戳，MVTO 的算法策略可以保证多个并发事务的执行可以等价于它们按照时间戳串行执行。MVTO 算法的优势在于不需要维护全局的结构，不需要阻塞地检查是否有冲突发生。在分布式系统中，MVTO 所依赖的检测冲突的信息天然地随着数据分布在各个数据节点上，因此 MVTO 天生就是可拓展的。它的劣势在于它的一次读操作需要记录读时戳，如果我们把读时戳设计为持久的，那么在分布式系统中，我们还需要备份读时戳。

SSI（Serializable Snapshot Isolation）[10]是一种对 Snapshot Isolation 改进的策略，在 PostgreSQL 中有所应用。它通过破坏 Serializable Graph 中形成环的必要条件来达到可串行的隔离级别。为了检查出形成环的比较条件，它需要在 Snapshot Isolation 的基础上添加一些全局的数据结构，然而这在分布式系统中可能成为可拓展性的瓶颈。

WSI（Write-Snapshot Isolation）[8]是一个论文对 Snapshot Isolation 的反思。普

通的 Snapshot Isolation 为所有的对操作提供了一个 Snapshot，然后在事务提交的时候，通过检查与并发事务之间的写操作的冲突来避免一些异常。但是在 MVCC 的系统中，写操作事实上是没有冲突的，不同的事务写操作完成可以写不同的数据版本而互相不干扰。在论文[8]中，作者证明了在 MVCC 的系统中，我们只需要检查读写冲突就能使事务可串行化。需要说明的是，WSI 不是一种具体的算法实现，它和 MVCC 相似，也是一种并发控制算法的思想，可以应用于其他算法之中。

2.4 错误恢复

错误是数据库系统中不可避免的一部分，不管是硬件上的还是软件上的。因此，在设计数据库系统的过程中，我们考虑的常常不是如果规避所有的错误，而是在假设错误一定会发生的情况下，设计恰当的机制保证系统能从错误之中恢复。

错误恢复的核心是要保证持久性和一致性。一个常用的策略是 WAL（Write-Ahead Log）[16]，数据库系统会在正式更新数据库之前，先把操作写进日志中，并持久化到持久化存储中。这样，在系统崩溃并重启之后，通过查询日志可以重做（redo）那些已提交但没有被写到持久化存储中的更改，撤销（undo）未提交但是被持久化的更改。

2.5 HBase 简介

HBase 是一个开源的、分布式的、多版本的、非关系型的数据库[1]，它的设计参考了 Google 的 Bigtable[2]，因此也被认为是 Bigtable 的开源版本。HBase 可以处理 PB 级的数据量，稳定地运行在上千台服务器上。HBase 能够保证数据的持久性，并且能够从局部故障中自动恢复。这极大地减轻了 HBS 实现分布式事务的负担。

HBS 的数据模型可以看作多维的 Key-Value 映射。其中 Key 是 table，row，column，version 的四元组。HBS 中有 column family 的概念，即一个 column 下会有多个子 column。同一个 column 下的内容会聚集存储，不同 column 下的内容存储位置没有关联。因此我们通常把相关性强的数据存在同一个 column 之下以加速访问的速度。HBS 的数据模型就利用了这一点。

HBase 在基本操作上提供了 Get，Put 之类的常见接口。除此之外，HBase 还提供了 CheckAndMutate 原子操作，它会先读一个数据，做一定的检查，通过检查之后才会更新数据，如果没有通过就返回失败，这整个过程是原子的。我们可以方便地利用 CheckAndMutate 实现行级事务。在后续的 HBS 实现中，我们可以看到大量的应用。

除此之外，HBase 还提供了 Coprocessor 的接口。Coprocessor 有两类，一类是

RegionObserver，它提供了数据库操作的钩子；另一类是 Endpoint，我们可以使用它为 region server 实现新的 API，它可以使用调用一个 region server 的各种函数。Coprocessor 可以在 HBase 运行过程中动态地加载到 region server 或者某张表上；也可在 HBase 启动时静态地加载。

2.6 本章小结

这章内容简单介绍了 HBS 设计实现过程中需要用到的相关技术基础。首先本章介绍了事务处理相关的技术，包括隔离等级、并发控制算法、错误恢复。本章还讨论了基于 MVCC 的几个常见乐观并发控制算法及其优劣对比。最后，本章简单介绍了构建 HBS 的基石——HBase 数据库。

第三章 需求分析

3.1 任务概述

HBS 最核心的需求在于为 OLTP（On-Line Transaction Processing）和 OLAP（On-Line Analytical Processing）应用提供事务支持，同时系统要具有良好的可拓展性。业务层的程序员能够使用 HBS 提供的 API 接口在 HBase 上做事务处理。

3.2 设计约束

HBS 的设计需要基于 HBase，所以它依赖 HBase 的接口和 HBase 的运行环境。

3.3 功能需求

根据任务概述，HBS 的功能需求可以细化为一下几方面：

基础的事务处理编程接口。用户需要能够方便地使用这些接口创建事务，提交事务，回滚事务。

基础的数据表管理操作。因为 HBS 需要屏蔽 HBase 的数据模型，为 HBase 表设置特定的参数，因此 HBS 还需要提供表的删除，表的创建等接口。

基础的数据更新、读取操作。HBS 需要至少提供读和写两个最基础的数据操作。其他复杂的数据操作可以由用户层的代码封装实现。

自动的错误恢复能力。HBS 的主要逻辑在客户端，而客户端的代码并不完全稳定可靠。因此，HBS 需要假设客户端是最容易出错的一环，并在客户端崩溃时保证整个系统的一致性不被破坏。

3.4 非功能性需求

根据任务概述，HBS 的非功能性需求可以细化为以下几方面：

系统需要有良好的可拓展性。HBS 不能以牺牲可拓展性为代价提供事务支持。

系统的延迟不能过高。因为要支持 OLTP 的业务场景，要求 HBS 能够在较大的延迟之下完成事务。但是因为 HBase 本来是强一致性的系统，可能本身具有较高的延迟，所以对延迟的要求是不能在 HBase 的基础之上增加太多

3.5 系统用例

根据上文所述的需求分析，用户视角下的用例如图 3-1 所示。

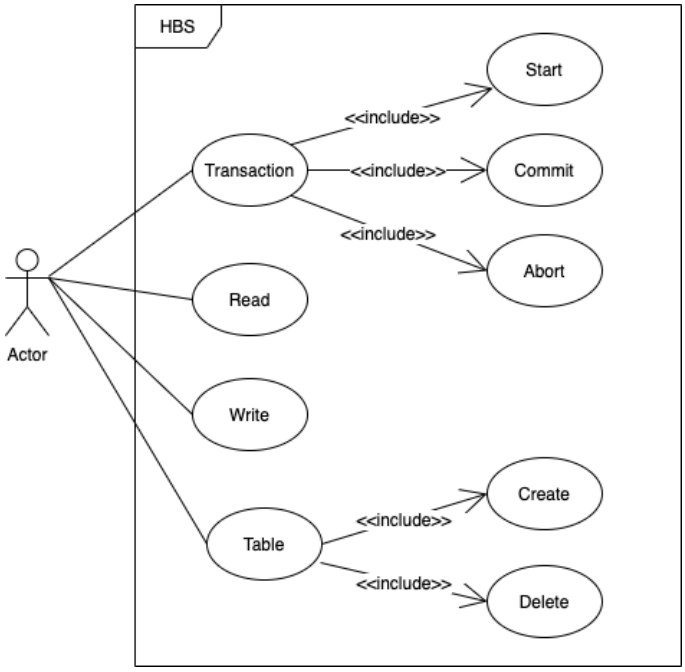

图 3-1 需求分析系统用例图

由用例图可以看到，用户能够利用 HBS 创建和删除表，开始事务，并在事务内做读写操作，最后可以选择提交或者终止事务。

3.6 本章小结

本章内容主要分析了 HBS 的用户的需求，从 HBS 最核心的 OLTP 和 OLAP 业务场景下的事务支持出发，逐步细化功能性需求和非功能性需求，最终给出了系统用例。

第四章 系统设计

HBS 在设计上学习了 Percolator 和 Omid。HBS 采用和 Percolator 相似的两阶段提交的策略以避免使用一个中心节点来作为协调者。同时 HBS 使用 Omid 中的全局的 Commit Table 来记录所有事务的状态。HBS 使用乐观的并发控制算法 MVTO，利用全局单调递增的时间戳对并发的事务的读写操作排序以保证可串行化。与 Percolator 和 Omid 不同的是，HBS 实现了可串行化的隔离级别。HBS 还使用了 Transaction Manager 来记录活跃的事务和生成全局单调递增的时戳。

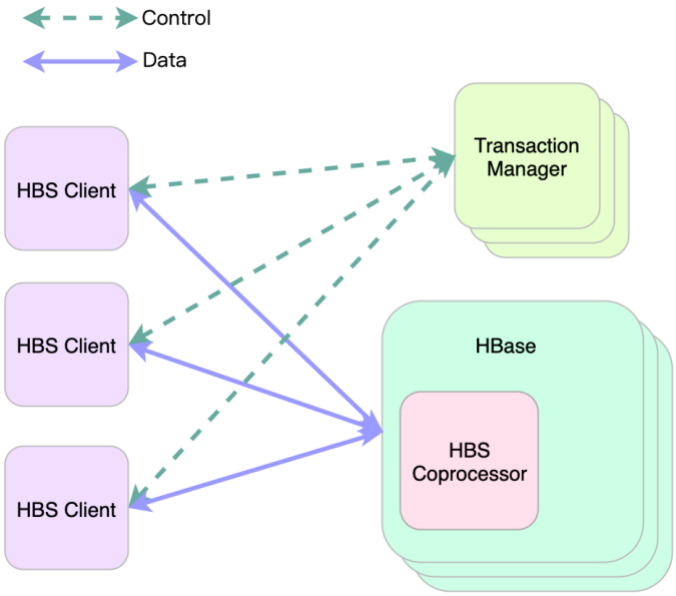

图 4-1 HBS 系统架构

HBS 的主要部分作为库的形式被用户调用，这部分被称为 HBS Client。客户端的库中完成了事务处理的大多数逻辑，包括并发控制算法和错误恢复。HBase 并没有提供和 Bigtable 类似的行级事务，一部分操作仅仅靠调用 HBase 提供的 API 难完成 HBS 可串行化的调度。为此，本文将这部分操作设计为 HBS Coprocessor。HBS Coprocessor 是运行在 HBase 的 region server 上的服务，提供 MVTO 算法需要的读写操作。

4.1 并发控制策略设计

并发控制策略是事务处理系统最核心的部分，它决定了一个事务执行的所有流程。

Percolator、Omid 之类的分布式系统为了实现系统的可拓展性，只提供了快照隔离级别。Snapshot Isolation 在实现上需要检查并发事务之间的写写冲突（write-write conflict）。因为大多数的事务的写集合并不大，所以检测写集合并不需要巨大的开销。Percolator 使用时间戳加上锁的方式检查并发事务是否有写写冲突。Percolator 的做法不依赖中心节点，可拓展性好，但是两阶段提交的代价比较大。Omid 使用的策略是无锁的，但是它依赖一个中心节点检测并发事务之间是否有冲突。虽然 Omid 实现了无锁的分布式事务，也不需要两阶段提交，但是它依赖几个中心节点极大地限制了系统的可拓展性。Snapshot Isolation 虽然为并发事务提供了一定的隔离级别，但是并不能保证事务的并发执行不相互影响。例如，写偏序（write skew）在 Snapshot Isolation 中依然会发生[9]。

HBS 设计的目标之一是提供可串行化的并发控制，并且不能以牺牲可拓展性为代价。

4.1.1 MVTO 算法

考虑分布式系统的可拓展性，并发控制算法最好不依赖中心节点或者尽量少地依赖中心节点。参照 2.3.2 关于乐观并发控制的讨论，OCC 和 SSI 都需要记录全局的事务执行状态以检查冲突。MVTO 算法的冲突检测只依赖读写时间戳，而读写时间戳都随着数据分布在各个存储节点上，因此 MVTO 并发控制可以完全地分布式化。选择 MVTO 算法的代价是每次读操作都需要更新最大读时戳，可能降低读操作的性能。

4.1.2 Write-Snapshot Isolation

论文[8]证明了在一个 MVCC 的系统中，只检查读写冲突就足以达到可串行化隔离级别。该论文把传统的检查写写冲突的 Snapshot Isolation 称为 Read-Snapshot Isolation，把只检查读写冲突的做法称为 Write-Snapshot Isolation。论文[8]还给出了基于 OCC 的 WSI 算法，但是这个实现需要记录事务的所有读集合并做相应的冲突检查。这相比 Snapshot Isolation，冲突检测的项目增多了，系统的吞吐量也相应地降低。

我们可以为 MVTO 实现 WSI 优化。在 MVTO 算法的写操作中，只需要检查读写冲突，而不再检查写写冲突。因为 MVTO 只会根据写集合做冲突检测，所以在实现 WSI 后，冲突检测的量减少一半，等同于 Snapshot Isolation 所需的冲突检测数。

在 MVTO 中应用 WSI 使得 MVTO 算法在理论上冲突检测的代价等同于

Snapshot Isolation。相比传统的 MVTO，写写冲突不再导致事务终止，极大地提升了系统的并发度。

4.1.3 两阶段提交

HBS 使用两阶段提交的策略提交事务。第一阶段，把所有写集合中的数据以 write intent 的形式写到 HBase 中，每次写操作还需要检查是否有冲突发生。Write intent 不是正式的数据，不能直接被其他事务读取。如果第一阶段没有发生冲突，那说明这个事务具备提交的条件，可以完成提交。第二阶段是把 write intent 变成正式数据。

我们之所以需要两阶段提交是因为第一阶段可能出现读写冲突，而我们必须在出现冲突的时候撤销事务提交。在一个事务还没有完成所有提交过程前，另一个并发事务可能读到它写的数据，而前者不一定能够成功提交，因此我们使用 write intent 保证并发的读操作不会读到一个未提交的数据。

在 HBS 的设计中，一个事务读到 write intent 时需要等待 write intent 的提交。HBS 设计了 TM（Transaction Manager）来维护所有活跃事务的状态。当一个事务读到 write intent 的时候，可以通过 TM 来监听 write intent 所述事务的状态。

在客户端没有崩溃的情况下，上述策略能够很好地完成事务处理的工作。但是考虑到客户端崩溃的情况，一个 write intent 可能属于一个已提交但是在第二阶段崩溃的事务，也可能属于一个未提交且在第一阶段奔溃的事务。考虑到这些情况，HBS 借鉴了 Omit 的 Commit Table 用以记录所有结束事务的状态。当一个事务读到 write intent 时，不仅需要通过 TM 判断 write intent 所属事务是否活跃，还需要查询 Commit Table 判断 write intent 是否属于已经崩溃的事务。

4.2 数据模型设计

HBS 的数据模型大体上延续了 HBase 的设计，不同的是一个 column 下不在有 column family。对用户而言一张 HBS 表就是行、列作为 Key 的 Key-Value 存储，即$(Row, Column) \rightarrow Value$。这样一个映射被称为一个 Cell，如表 4-1 中所示的是同一个数据项的两个不同版本。

从表 4-1 中我们可以看到 HBase 原来的 column family 被 HBS 内部逻辑使用，他们代表的意义如下：

@Data：这个 column 下用户真实存储的数据。如表 4-1 中“Row1”下存的内容是“Hello”。

@RT：read timestamp，即最大读时戳——读过这个数据项的事务最大的时戳。

@Uncommitted：未提交的标识。如果一个数据项有 Uncommitted 这列则说明这个数据项是一个 write intent。Write intent 的概念取自 CockroachDB[14]，它表示一个 Cell 已经写在了 HBase 中，但是没有被提交。Write intent 不能之间被读取，试图读取它的事务需要等待它的提交。把这列从 Cell 中清除就能把一个 Write intent 变成一个正式的提交。

Version：write timestamp，写这个数据项的事务的时戳。Version 在 HBase 中不是一个 Column，它是每个数据项都有的一个属性。

表 4-1 HBS 中的两个 Cell

	Column:@Data	Column:@RT	Column:@Uncommitted	Version
“Row1”	“Hello”	2	-	1
“Row1”	“World”	-	“yes”	3

4.3 Transaction Manager 设计

TM（Transaction Manager）需要为事务的正常执行提供支持，它提供分配时戳，检测事务是否为活跃状态，等待事务停止执行的接口。不同的事务可以利用 TM 来完成协调、同步的工作。

如图 4-2 所示，TM 是多节点的设计。当 leader 节点正常工作时，follower 节点监控 leader 节点的状态但不提供服务。当 follower 节点检测到 leader 不能提供正常服务时，follower 开始重新选举新的 leader 节点。

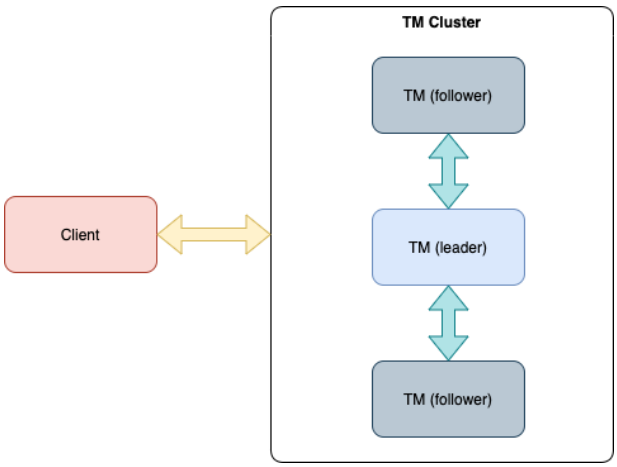

图 4-2 Transaction Manger 设计图

在一个事务开始的时候，它会向 TM 申请一个开始时戳，并使用这个时戳完

成读写操作。当一个事务需要根据其他事务决定自己的执行步骤时，它可以通过 TM 提供的 watcher 机制观测到别的事务的状态。最后，当一个事务结束时，它会释放它申请的时戳。

表 4-2 TransactionManager 接口设计

接口名称	返回类型	说明
allocate	long	TM 分配一个全局单调递增的时戳，并在 TM 中记录该时戳为活跃事务。
release	void	释放一个 TM 中记录的活跃事务。
exists	boolean	测试一个事务是否被 TM 记录为活跃事务。
waitIfExists	boolean	等待一个事务停止执行。

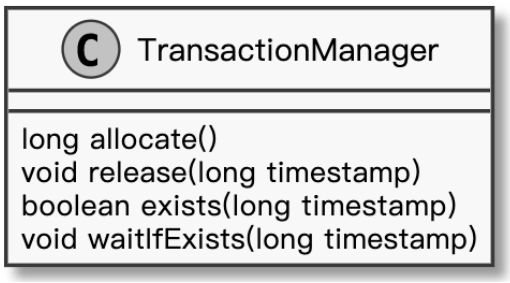

图 4-3 TransactionManager 类图

根据这些设计，我们可以绘制出 TM 的类图如图 4-3 所示。

4.4 Commit Table 设计

表 4-3 CommitTable 接口设计

接口名称	返回类型	说明
status	Status	在 Commit Table 中查询事务的状态。
commit	boolean	在 Commit Table 记录事务为已提交状态。
abort	boolean	在 Commit Table 记录事务为已终止状态。

Commit Table 就是一张普通的 HBase 表它记录了事务的状态。当一个事务在 commit table 中没有记录时，它的状态是 uncommitted；当一个事务成功提交，它

在 commit table 中记录的状态是 committed；当一个事务终止后，它在 commit table 中记录的状态是 aborted。

需要说明的是，一个事务只能自己 commit 自己，也就是说 committed 的状态只能由事务自身写，不能由别的事务写。而 aborted 的状态可能是别的事务发现了某个事务在 TM 中没有记录，认为它已经崩溃了，帮助它写下 aborted 状态的。一个事务是不会自己在 commit table 中记录自己 aborted 状态的。

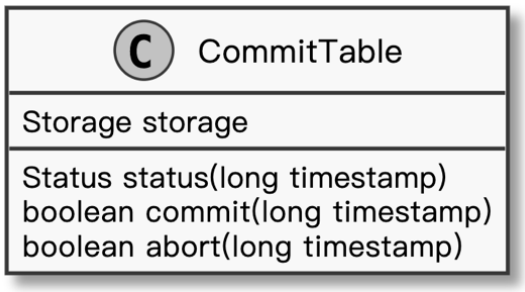

图 4-4 CommitTable 类图

根据以上设计，可以绘制 Commit Table 的类图如图 4-4 所示。

4.5 HBS Coprocessor 设计

HBS Coprocessor 是一个 HBase Coprocessor 接口的实现，它需要为 MVTO 算法提供读写能力。我们之所以把 MVTO 的读写操作设计为 coprocessor 是因为他们必须是原子操作。比如 MVTO 的读操作会把数据项的读时戳更新为最大的读时戳。如果两个读操作同时执行，那么它们可能会误认为它们的时戳就是最大的读时戳，然后都去更新读时戳。一种可能的坏的结果是时戳小的事务后于时戳大的事务更新读时戳。在这种情况下，关于读时戳的假设就被破坏了。

MVTO 的写操作也可能出现类似的影响，并且读操作和写操作之间也可能相互影响的情况。MVTO 的读写相容性如表 4-5 所示。

表 4-4 MVTO 读写相容性表

	read	write
read	不相容	不相容
write	不相容	不相容

我们把 MVTO 的读写设计为两个独立的 Coprocessor，分别叫做 GetEndpoint 和 PutEndpoint，它们分配提供读写的能力。

表 4-5 GetEndpoint 与 PutEndpoint 接口设计

接口名称	返回类型	说明
GetEndpoint.Get	byte[]	调用 GetEndpoint.Get，读取某个 Cell 的内容，并更新其最大读时戳。
PutEndpoint.Put	boolean	调用 PutEndpoint.Put，检测是否发生读写冲突。如果没有，写一个 write intent；反之，返回失败。

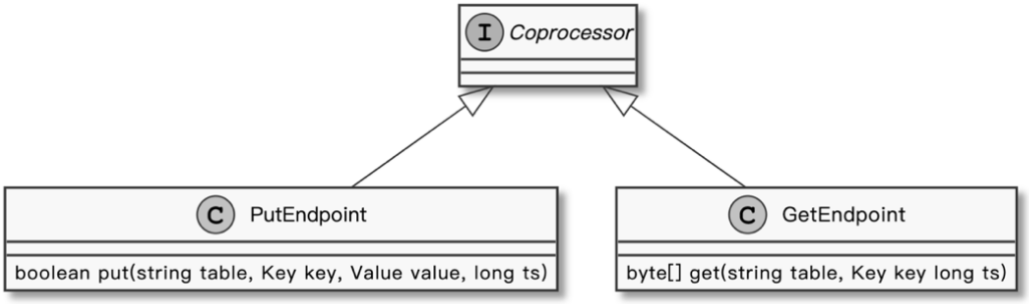

图 4-5 GetEndpoint 和 PutEndpoint 类图

根据以上设计，可以绘制 GetEndpoint 和 PutEndpoint 的类图如图 4-5 所示。

4.6 Storage 设计

表 4-6 Storage 接口设计

接口名称	返回类型	说明
createTable	void	创建一张数据表。
deleteTable	void	删除一张数据表。
mvtoGet	byte[]	MVTO 算法的读操作，需要更新最大读时戳。
mvtoPut	boolean	MVTO 算法的写操作，需要检查是否发生读写冲突。
removeCells	void	删除一组 cells，无论其是否是 write intent。
removeUncommittedFlags	void	删除一组 write intent 的 uncommitted flags，使它们变为已提交状态。

HBase 提供的能力在 HBS 中以 Storage 接口的形式呈现。Storage 定义了表格创建删除、数据读写、GetEndpoint 和 PutEndpoint 调用的能力，其具体的接口设计

如表 4-6 所示。在 Storage 接口的设计下，HBS 不再需要依赖 HBase 的具体实现，降低了 HBS 与 HBase 的耦合度。事实上，只要正确实现 Storage 的接口，HBS 也能为其他存储系统提供事务支持。

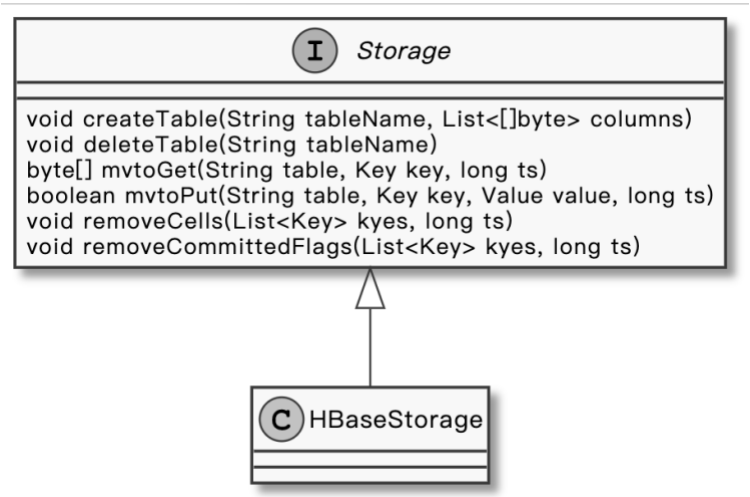

图 4-6 Storage 类图

根据上述设计，我们可以得到 Storage 的类图如图 4-6 所示。

4.7 HBS Client 设计

HBS Client 为用户提供的是一个 Transaction 所有操作的 API，用户可以使用这个类完成事务处理操作。

表 4-7 Transaction 接口设计

接口名称	返回值	说明
start	void	通过 TM 申请一个时戳，并初始化事务局部读写集合。
commit	boolean	两阶段提交事务：把局部写集合的数据写到数据库中，清除 write intent 中的 uncommitted flags。第一阶段冲突检测失败则放弃提交返回失败，如果第一阶段成功则在 Commit Table 中记录提交状态。
abort	void	终止事务：释放 TM 中的记录。
get	byte[]	调用 Storage 的读接口读取数据，并判断读到的内容是否为 write intent。如果读到 write intent，需要等待 write intent 的事务的终止。
put	void	暂存数据到局部写集合中

一个事务开始时，会通过 TM 获取时戳，在其读写的过程中会调用 Storage 提供的接口操作数据库，并通过 TM，Commit Table 完成事务相关的操作。事务有一个局部的读集合，用以缓存读过的数据，避免重复访问底层存储。事务还有一个局部写集合，用以暂存写数据，待事务提交时再写到数据库中。另外事务提供 abort 接口，当业务层的一致性约束不满足时，需要调用 abort 接口终止事务。HBS 并不需要 rollback 接口，因为所有数据只会在 commit 的时候才会被真正写到数据库中。

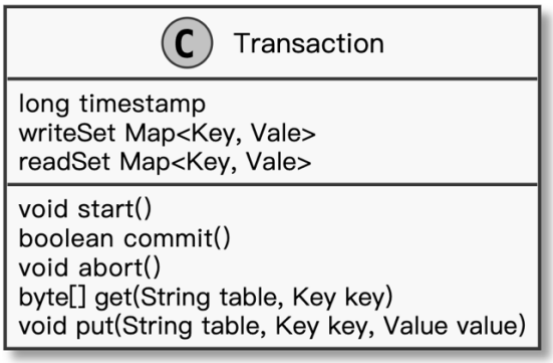

图 4-7 Transaction 类图

根据上述设计，Transaction 类图绘制如图 4-7 所示。

4.8 总体设计

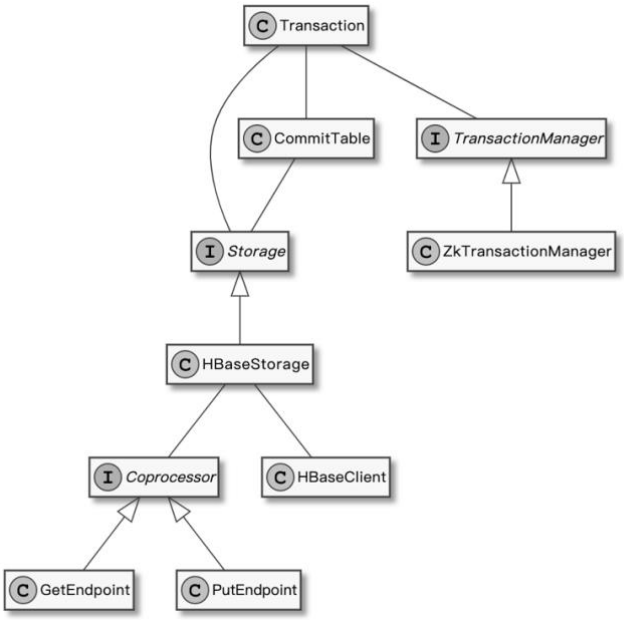

图 4-8 HBS 总体设计

在设计好前面各个子模块后，我们可以得到系统的总体设计如图 4-8 所示。

4.9 本章小结

本章首先讨论了 HBS 使用的并发控制策略，包括 MVTO、WSI、两阶段提交，并给出使用这些策略的理由。本章接着给出了在这样的并发控制策略和上一章的需求分析之下 HBS 的系统总体设计，与各个子模块的详细设计，最终给出了 HBS 系统的总体架构设计。

第五章 系统实现

在具体实现上，HBS 主要依赖 HBase 和 Zookeeper。其中，TM 模块的具体实现是利用 Zookeeper 提供各项基础服务，Commit Table 的实现是 HBase 中的一张普通的数据表。

一个事务的执行的大致流程如下：通过 HBS Client 向 TM 请求一个开始时间戳，TM 会为事务分配一个时戳，并记录该时戳为活跃事务；调用 HBS Coprocessor 读取事务需要的数据，并把事务写的数据保存到事务局部的缓存中；事务所有读写操作完成后，开始两阶段提交工作，成功后请求 TM 清除事务活跃状态。

在后面的子章节中，我们会详细讨论各个模块的实现细节，分析 HBS 如何从错误中恢复。

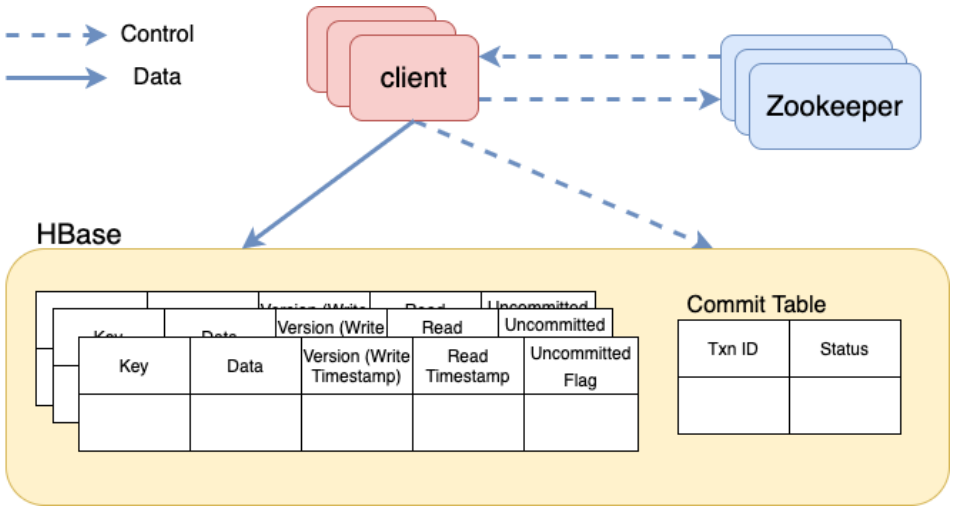

图 5-1 HBS 使用 HBase 作为底层存储

5.1 Transaction Manager 实现

TM（Transaction Manager）的实现依赖 Zookeeper，它的主要功能是分配事务时戳，维护活跃事务状态。Zookeeper 提供了强大的分布式协调服务，我们可以方便地利用 Zookeeper 的各种能力来实现 TM。

TM 时戳分配利用的是 Zookeeper 的 sequential node（node 是 Zookeeper 中的基本逻辑单元，类似与文件系统中的一个文件）。当我们请求创建一个这样的 node 的时候，Zookeeper 会自动为它分配一个整数作为后缀。这个整数是全局单调递增的，而且会备份到 Zookeeper 的多数节点上。有了这些特性，这个整数正好可以作为时戳被 MVTO 算法使用。

在一个事务开始时，它会使用 TM 分配一个全局递增的时戳，并在 Zookeeper 中创建一个临时节点，这个节点会在事务活跃的时候存在。Zookeeper 会和发起事务的客户端保持心跳通信，以保证在整个事务活跃的期间，临时节点不被删除。如果客户端崩溃了（Zookeeper 不能收到客户端发来的消息持续一段时间），Zookeeper 会删除临时节点。一个事务成功提交后也会主动删除临时节点。

一个事务还能通过 TM 感知到其他事物的状态。例如一个事务试图去读一个 write intent 的时候，它会先通过 TM 查询这个 write intent 所属的事务的状态。如果事务还在运行中，它会在 Zookeeper 中注册一个监听 node 删除的 watcher。在某个事务执行结束时，那些注册过 watcher 监听它的事务就会收到通知。

5.2 Commit Table 实现

Commit Table 可以被看作一张特殊的 HBase 的数据表，它记录了所有事务的状态。Commit Table 提供的两个主要接口，commit 和 abort 都是通过 HBase 的 CheckAndMutate 实现的。它们都会先检查在表中是否已经有操作事务的记录，如果有，则 Check 失败，返回错误。如果没有，才将事务状态记录到表中。值得注意的是，Check 和 Mutate 是一个原子操作。也就是说，在这个过程中，如果有多个对同一个事务并发的修改，只有一个会返回成功。这一点在后续的错误恢复中有重要的用途。

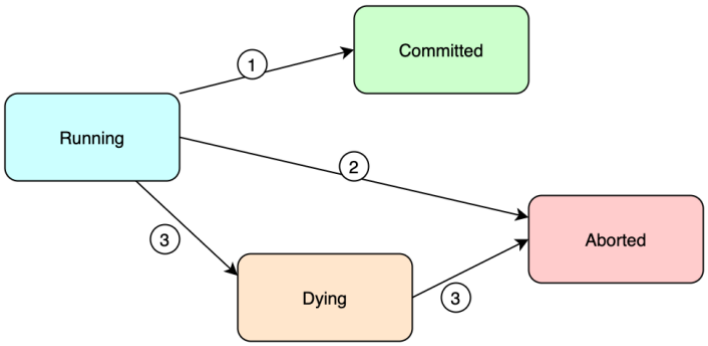

图 5-2 事务状态图

一个事务可能处于以下四种状态之中，一个事务的状态只能像图 5-2 中所展示的路径改变：

Running：事务正在执行过程中，这个事务必须在 TM 中有对应的 node。

Dying：事务已经终止执行，一个事务在 TM 中没有对应 node，且它的状态还没有记录在 Commit Table 中。

Committed：事务在 Commit Table 中写下它已提交后的状态。

Aborted：事务在 Commit Table 记录的状态是 Aborted。

5.3 HBS Coprocessor 实现

Coprocessor[15]是 HBase 提供的把客户端的代码卸载到 region server 上执行的接口。我们要实现的 HBS Coprocessor 需要为 MVTO 算法提供基本的读写能力。

Percolator 实现基本的读写操作时会依赖 Bigtable 行级事务的特性。但是 HBase 只提供了 CheckAndMutate 这种更为基础的操作。虽然我们可以简单地将其拓展为行级事务，但是这样的行级事务需要多次与 HBase 通信，性能上会大打折扣。幸运的是 HBase 提供了 coprocessor 的接口，我们可以把 MVTO 的读写操作卸载到 HBase 的 region server 中去执行。我们会为 MVTO 分别实现读写操作的 coprocessor，它们分别被称为 GetEndpoint 和 PutEndpoint，客户端可以使用 RPC 调用这两个 Endpoint。

5.3.1 GetEndpoint

代码 5-1 GetEndpoint 伪代码

```
1. # key 是要读的数据的 key
2. # TS 是这个读操作的事务的时戳
3. get(key, TS):
4.      # 获取 key 所在 row 的锁
5.      lock = rowLock(key)
6.      # 找到要读的版本号
7.      version = max(c.WT if c.WT < TS for c in data[key])
8.      # 读该版本
9.      cell = for c in data[key] where c.WT == version
10.     # 更新读时戳
11.     if cell.RT < TS:
12.         cell.RT = TS
13.     # 释放锁
14.     lock.release()
15.     return cell
16.
```

在 GetEndpoint 的实现中，我们先获取 row 的锁，保证在整个 Get 的过程中没有别的并发的读和写。我们根据时间戳读取一个最新的版本。然后我们检查并更新读到的版本的读时戳。最后，我们释放 row lock 并且返回读到的 cell。值得注意的是，这里的读操作也需要获取普通的锁而非读锁。因为 MVTO 的读操作是互斥的，不能同时进行。

5.3.2 PutEndpoint

PutEndpoint 首先读一个比要写的版本早的一个版本。如果这个 cell 的读时戳大于当前的时戳，说明之前有一个事务读了老版本，而且这个事务的时戳大于当前事务的时戳。这就产生了一个读写冲突——前一个事务的读操作本应该读到的内容是当前事务写的内容才对。如果发生了冲突，我们必须返回失败，然后客户端收到写失败的信息后会主动终止事务。如果没有冲突发生，我们可以正确地写数据，然后释放 row lock 并返回成功。

代码 5-2 PutEndpoint 伪代码

```
1. # key 是要写的数据的 key
2. # value 是要写的值
3. # TS 是这个读操作的事务的时戳
4. Put(key, value, TS):
5.     # 获取 key 所在 row 的锁
6.     lock = rowLock(key)
7.     # 读 TS 之前的一个版本
8.     version = max(c.WT if c.WT < TS for c in data[key])
9.     cell = for c in data[key] where c.WT == version
10.    # 检查读写冲突
11.    if cell.RT > TS:
12.        # 如果有比当前事务更年轻的事务读过老版本，那么这个写操作失败
13.        lock.release()
14.        return false
15.    data[key] = {Data: value, WT: TS, RT: 0, Uncommitted: yes}
16.    lock.release()
17.    return true
```

5.4 HBase Storage 实现

这部分的实现主要是把 HBase 的接口映射为 Storage 接口的要求。其中 MVTO 的读写操作需要分别调用 GetEndpoint 和 PutEndpoint 实现。

5.5 HBS Client 实现

HBS 客户端完成了分布式事务的大部分逻辑，包括调用 GetEndpoint，PutEndpoint，两阶段提交，和 TM 通信等等。在讨论这部分实现时，我们会暂时避开一些可能由崩溃导致的错误以便于我们理解。对于错误处理的部分，我们会在错误恢复章节详细讨论。

HBS 事务的定义很简单——包含一个时间戳，写集合和读集合，如代码 5-3 所示。在事务开始的时候，我们会使用 TM 为事务分配一个时戳，并初始化读集

合和写集合为空。

代码 5-3 Transaction 定义伪代码

```
1. class Transaction:
2.     timestamp long.          # 事务时戳
3.     writeSet   Map<Key, Value> # 写集合
4.     readSet    Map<Key, Value> # 读集合
5.
6.     New():
7.         return Transaction {
8.             timestamp: TransactionManager.allocate(),
9.             writeSet:  {},
10.            readSet:   {},
11.        }
12.
```

5.5.1 Get

Client 的 Get 不仅仅是调用 GetEndpoint。Get 会先在 writeSet 和 readSet 中查找要读的值。使用 readSet 是为了避免重复地访问 HBase，以减轻 HBase 的负担；而使用 writeSet 是为了读到事务自己写的内容。writeSet 后 readSet 是因为我们需要读到修改后的内容而不是修改前的内容。Get 在获取 GetEndpoint 的结果后，还需要检查这是否是未提交的 write intent。如果是，那么它需要等待 write intent 所属事务的结束。

这里的等待是利用 TM 实现的。当前事务的执行会阻塞直到 write intent 所属事务执行结束时。但是我们并不清楚事务结束的原因，可能是成功提交，也可能是提交失败或者客户端崩溃。因此当前事务会查询 Commit Table 获取该事务的状态，查询结果可能有三种：

事务已提交（Committed）：write intent 所在事务成功提交。在这种情况下，当前事务会清除 write intent 中的 uncommitted flag，使其变为已提交，并返回 write intent 中的值。

事务已终止（Aborted）：这种情况下说明这个 write intent 提交失败，当前事务会清除 write intent，调用 GetEndpoint 重读。

事务未提交（Uncommitted）：这种情况可能是事务已经失去了和 TM 的连接，但是还没有把自己的状态写入到 Commit Table 中，也就是所谓的 Dying 状态。当前事务会帮助该事务写 aborted 状态，并清除它的 write intent。最后，按照事务终止的情况处理。

代码 5-4 Transaction Get 伪代码

```
1.     Get(key):
2.         # 尝试在写集合和读集合中找到 key
3.         if key in writeSet:
4.             return writeSet[key]
5.         if key in readSet:
6.             return readSet[key]
7.
8.         while true:
9.             cell = GetEndpoint.Get(key, timestamp)
10.            if cell.Uncommitted:
11.                # 如果读到的是 write intent，等待该写操作的事务结束
12.                if waitAndClean(key, cell.WT): # write intent 被成功提
   交
13.                    readSet[key] = cell.Data
14.                else: # write intent 提交失败，重读一个更老的或者更新的版
   本
15.                    continue
16.            else:
17.                readSet[key] = cell.Data # 读的数据已提交
18.            break
19.        return readSet[key]
20.    # 等待时戳为 ts 的事务提交，返回它是否提交成功
21.    # 根据提交结果清理 write intent，或者 write intent 的 uncommitted
   flag
22.    waitAndClean(key, ts):
23.        TransactionManager.waitWhileExsits(ts) # 等待事务 ts 结束
24.        # 事务已经结束执行，但是可能还没有在 CommitTable 中记录状态
25.        while true:
26.            status = CommitTable.getStatus(ts)
27.            if status = Uncommitted:
28.                # 事务已断开了和 TM 的连接，尝试写终止状态
29.                if CommitTable.Abort(ts):
30.                    status = Aborted
31.                 else:
32.                     continue # 终止失败，可能有并发的操作，重试
33.            if status = Aborted:
34.                removeCell(key, ts) # 清除终止事务的 write intent
35.                return false
36.            else:
37.                # 事务提交成功
38.                removeUncommittedFlag(key, ts) # roll-forward: 把
   write intent 变成普通数据
39.                return true
40.
```

Get 和 waitAndClean 中都有 while 循环。Get 中的 while 循环是为了在读到的 write intent 提交失败的情况下重读另一个版本（必须在成功清理 write intent 后才能读到新版本）。waitAndClean 中的 while 循环是因为 Abort 的操作可能会失败。Abort

的实现是一个 CheckAndMutate 的原子操作，只有当事务在 Commit Table 中没有记录的时候才会成功。这样的设计可以避免并发操作带来的潜在问题，这一点会在下一节错误恢复中详细分析。

5.5.2 Put

Put 不会直接把数据写到 HBase 中，而是在事务内的局部缓存结果，等待提交的时刻再写到 HBase 中。这么做是为了缩短两阶段提交的时间，减少其他事务需要等待 write intent 的概率。如果我们每次 Put 都向 HBase 写 write intent，那么两阶段提交的时间区间会跨越整个事务执行的流程。在这个区间内访问这个事务写的数据的其他事务都需要通过 TM 等待当前事务，这对 TM 来说负载压力会比较大，同时也使得事务的读性能下降。

代码 5-5 Transaction Put 伪代码

```
1.     Put(key, value):
2.         writeSet[key] = value
3.
```

5.5.3 Commit

Commit 使用的策略是分布式系统中常见的两阶段提交。

第一阶段，调用 PutEndpoint 把 writeSet 中缓存的数据作为 write intent 写到 HBase 中。PutEndpoint 会检查是否有读写冲突发生。如果第一阶段 writeSet 中的数据都成功写到 HBase 中了，就来到了 commit point。Commit point 事务会把已提交的状态写到 Commit Table 中。在 commit 成功后，这个事务就被认为已经提交了。即使它的 write intent 中的 uncommitted flag 没有清除，这些 write intent 已经变成可见的了。第二阶段，事务清理自己 write intent 中的 uncommitted flag。这一阶段的工作对于 HBS 系统的正确性是没有影响的，因为即使 write intent 有 uncommitted flag，别的事务依然可以通过查询 Commit Table 中的状态知道这个 write intent 是否已提交的。但是如果不清除 uncommitted flag，每次读 write intent 的代价会比较大。因此第二阶段的工作可以异步地执行，两阶段提交的代价可以近似于一阶段提交。

如果第一阶段提交失败，或者在写 Commit Table 的时候失败，我们都认为这个事务不能被正确提交。这种情况下，我们会清理已经写到 HBase 中的 write intent，并释放 TM 中的 node，让其他事物感知到当前事务结束执行，最后返回失败给应用层。

代码 5-6 Transaction Commit 伪代码

```
1.     # two phase commit
2.     Commit():
3.         # 第一阶段：prewrite
4.         # 写所有的 write intent，检查是否有冲突发生
5.         written = {}
6.         for k, v in writeSet:
7.             if PutEndpoint.Put(k, v, timestamp):
8.                 written.add(k)
9.              else:
10.                fail(written)
11.                return false
12.        # 提交点：
13.        if !CommitTable.commit(timestamp):
14.            fail(written)
15.            return false
16.        # 第二阶段：清理
17.        TransactionManager.release(timestamp) # 清除 Zookeeper 中的
   znode
18.        removeUncommittedFlags(written, timestamp) # 清理写集合中的
19.        return true
20.
21.    # 清除 prewrite
22.    fail(written):
23.        # 清除所有 write intent
24.        removeCells(written, timestamp)
25.        # 在 commit table 中记录状态
26.        CommitTable.abort(timestamp)
27.        TransactionManager.release(timestamp)
```

5.6 错误恢复

错误是一个分布式系统中无可避免的，我们系统设计的原则不是考虑如何规避错误，而是数据库的状态从错误中恢复。HBS 即使在部分组件发生错误的情况下，数据库的一致性也不会被破坏，而且能快速地从错误中恢复。

在单机版的数据库中，我们一般使用 WAL 策略[16]来完成错误恢复和持久化的工作。所有的数据库操作会先写在日志中，并持久化。当一个系统崩溃重启后，我们根据日志中事务是否提交 redo 或者 undo 日志中的操作就能恢复数据库的状态。

HBase 中也使用了类似的策略来保证数据的一致性和持久性。但是这对保证分布式事务的正确性是不够的，一个事务执行的过程中可能会产生很多导致数据库不一致的数据。HBS 虽然使用了 HBS Coprocessor 把部分工作交给了 region server，但是客户端依然承担了相当一部分可能影响数据库状态的工作。客户端是作为库被业务代码使用的，我们非但不能保证业务代码的可靠性，相反我们要假设客户端是整个系统中最容易出错的一环。

在 HBS 的设计中，我们会假设客户端可能在任何一行代码处崩溃。我们要提供的保障是无论客户端处于何种执行状态，整个数据库的一致性不被破坏，同时其他并发执行的事务受到的影响最小。在 HBS 中，我们依赖 TM(Transaction Manager)和 Commit Table 来提供这样的保证。

一个事务只会通过 GetEndpoint、PutEndpoint 和一些关于 write intent 的操作来改变数据库的状态。因此，只要我们保证在这些会改变数据库的地方，客户端崩溃并不会导致数据不一致就可以了。

5.6.1 Get 过程中的错误恢复

GetEndpoint 会改变读时戳，但是这个改变是原子地增大读时戳。读时戳的增大不会影响数据库的状态的一致性，它只是可能导致其他并发事务提交时候发生冲突。

在事务开始提交以前，一个事务只有读操作。在这个过程中 GetEndpoint 永远不会破坏事务的正确性。但是如果这个过程中遇到 write intent，当前事务可能会提交或者清理这个 write intent。首先提交 write intent 只会在 Commit Table 中记录了这个 write intent 所属的事务的状态为已提交后才会进行。而一个事务只会在没有冲突的情况下才会提交，因此这种情况下永远不会出错。

表 5-1 非原子的 Commit Table 下两个事务的执行状态

T1	time	T2
	1	读到 T1 的 write intent
断开和 TM 的连接	2	收到 T1 结束的信息
第一阶段提交成功，尝试在 Commit Table 中记录已提交的状态	3	尝试在 Commit Table 中把 T1 的状态记录为 Aborted
提交成功	4	清理 T1 的 write intent

另外一种情况是一个事务在 TM 中没有记录，被认为停止执行了，但是 Commit。Table 中还没有它的状态记录。这种情况比较复杂。一种可能性是这个事务的客户端崩溃了。在这种情况下，遇到它的 write intent 的事务会帮助它在 Commit Table 记录 Aborted 状态，并清理 write intent。另外一种棘手的状态是这个事务和 TM 断开连接了，但是它依然可以和 HBase 正常通信。一种极端的情况是除了自己还有别的事务同时更新自己在 Commit Table 中的状态。这种情况下，系统的正确性就必须依赖 Commit Table 的原子操作。

我们可以先假设 Commit Table 的操作不是原子的，那么就会出现表 5-1 的可能性。事务 T1 断开了和 TM 的连接，但是依然和 HBase 保持连接。T1 会成功地完成第一阶段提交，并把已成功提交的状态返回给业务层。但是 T2 同时会认为 T1 已经被终止，然后清除了已提交事务写的数据。这就导致了数据库状态的不一致。

如果我们 Commit Table 的操作修改为 CheckAndMutate 的原子操作，那么在表 5-1 的时刻 3 时，T1 或者 T2 的操作只有一个会成功。操作失败的事务会放弃后续导致数据库状态不一致的操作。从这个例子中，我们可以看到 Commit Table 的原子操作在保证系统正确性中的重要作用。

5.6.2 两阶段提交过程中的错误恢复

错误恢复中值得讨论的还有两阶段提交的策略。我们之所以在第一阶段只写 write intent，而不写正式数据是因为第一阶段承担了冲突检测的工作。如果不是所有冲突检测都成功，那么这个事务的数据就不应更被别的事务读到。因此两阶段提交是系统正确性最重要的保证。事实上，在 HBS 中，我们只需要一阶段就足以保证系统的正确性。因为 HBS 中的事务可以利用 Commit Table 来判断一个 write intent 是否是被提交的。但是这会极大地增加读操作的负担，每次读操作都需要至少两次访问 HBase。

两阶段提交常常被认为是系统性能的瓶颈，研究人员千方百计地想要去掉它。例如 Omid 没有使用两阶段提交的策略。但是其代价是 Omid 必须依赖一个中心节点检查并发事务之间的冲突。是否使用两阶段提交在分布式事务中是一个取舍的问题：两阶段提交可以保证系统的可拓展性，但是它同时带来的是高延迟；去除两阶段提交能带来低延迟的事务，但是为此我们常常需要牺牲系统的可拓展性。

5.7 本章小结

在上一章系统设计的基础上，本章完整地实现了 HBS。本章内容详细讨论了 HBS 的各个子模块的实现细节。我们还讨论了在这些实现之下，HBS 如何能从错误中恢复，如何保证分布式事务的正确性。

第六章 系统测试

系统的测试主要内容是测试 HBS 事务处理的正确性，保证它不会导致数据库不一致。事实上，我们更加应该测试的是 HBS 在真实的分布式场景之下的性能表现，但是这至少需要数十台服务器才能提供可用的测试环境。因此，我们在系统测试章节将更加聚焦于测试的方法和系统的正确性上。

6.1 事务测试

我们使用的测试依据主要来源于论文[9]中提到的多种并发中出现的异常。测试方法主要分为两类。

一类是模拟事务并发执行过程中可能出现的执行顺序。我们会模拟一些导致冲突的执行顺序，以检查 HBS 能否正确检测出冲突，并终止导致冲突的事务。这类测试可以有效地模拟并发中出现的各种极端情况，同时事实上串行的执行顺序也便于我们找到错误，推测系统行为。

另一类测试是并发地执行大量的事务。这些事务是精心设计的，可能导致各种并发错误的，业务场景中常见的事务。这类测试最大地还原了真实业务场景下事务的执行状态，进一步验证了系统的正确性和稳定性。

表 6-1 是 HBS 的测试结果，可以看到 HBS 通过了所有测试，可以提供可串行化的隔离级别。

表 6-1 测试内容和结果

异常	是否通过
脏读（dirty read）	是
脏写（dirty write）	是
不可重读（fuzzy read）	是
更新丢失（update loss）	是
读偏序（read skew）	是
写偏序（write skew）	是

6.2 可拓展性

我们可以把 HBS 按照设计分成几个部分讨论可拓展性的问题。

首先，HBS 底层的存储系统 HBase 是一个在设计上有非常好的可拓展的存储系统，它可以处理 PB 级的数据量，运行在上千个服务器上。因此，HBase 决不可能成为 HBS 的可拓展性的瓶颈。

HBS 在算法上使用的 MVTO，这个算法在检测冲突时只依赖和数据一起存储的读写时戳。HBS 大部分逻辑都在 client 和 region server 中实现，而这两部分本身就是具有很好的可拓展性的。

HBS 唯一的中心化依赖是 TM，目前 TM 的实现依赖 Zookeeper。TM 要负责时戳的分配，维护活跃事务的状态。可以说目前的 HBS 实现中，整个系统的可拓展性瓶颈就在于 Zookeeper。根据 Zookeeper 官网[13]的数据，在 2Ghz Xeon and two SATA 15K RPM drives 的服务器上，三个 Zookeeper 副本每秒钟写的吞吐量是 20000，读的吞吐量是 80000。即便是作为一个需要多数备份的系统，Zookeeper 依然提供了足够事务系统处理的吞吐量。

根据以上的分析，我们可以得出 HBS 可拓展性良好，并且至少能够支持上千甚至上万个事务的并发执行。

6.3 本章小结

本章主要从保证数据一致性的角度给出了对 HBS 的测试结果，可以看到 HBS 正确地通过了所有并发异常的测试，能够提供可串行化的事务支持。最后我们简单地讨论了一下系统的可拓展性，并得出了 HBS 具有良好的可拓展性的结论。

第七章 总结与展望

7.1 全文总结

本文讨论了分布式数据库中事务处理的乐观并发控制算法，并详细地介绍了本文设计并实现的基于 HBase 的分布式事务处理系统 HBS。HBS 目前实现了它基本的分布式事务处理的能力，可以为 HBase 提供 ACID 语义的事务支持。它最大的创新在于应用 MVTO 的并发控制算法和 WSI 的优化，它们使得 HBS 在实现可串行化隔离级别的同时具有良好的可拓展性。这些特性使得 HBS 能够满足 OLTP（On-Line Transaction Processing）的需求。同时 HBS 不会阻塞只读事务的特性使得它同时能够满足 OLAP（On-Line Analytical Processing）的需求。

最后，本文从数据一致性的角度测试并验证了 HBS 可串行化事务处理的能力。

7.2 思考与展望

尽管 HBS 完成了基本的分布式事务处理的能力，它依然有许多值得进一步完善的地方：

垃圾回收机制：目前的 HBS 实现中没有垃圾回收的机制，所有物理版本都保存在 HBase 中。如果没有查询某个时段内数据的需求，一些过期的数据永远不可能被访问到，但是却占用了大量的存储空间。

读时戳优化：目前的读时戳是持久化保存到 HBase 中的，这使得一次读操作的代价可能等于一次写操作。事实上，读时戳没有必要保存在 HBase 中，它只需要保存在内存中。但是这需要一定的策略来保证服务器崩溃重启好，数据库的一致性不被破坏。

时间戳服务：目前分配时戳利用的是 Zookeeper，但是当系统中执行的事务足够多时，Zookeeper 可能成为系统的性能瓶颈。实现一个独立的时间戳服务能够提高提供时戳分配效率，使之不再成为性能瓶颈。

致　谢

本论文的工作是在我的导师聂晓文老师悉心指导下完成的，感谢聂老师孜孜不倦的指引和教导。我还需要感谢在我毕业设计过程中给予我各种帮助和支持的同学们和朋友们。另外，我要感谢在分布式数据库方向辛勤耕耘的科研人员，我的工作如果不是站在巨人的肩膀上是不可能完成的。最后我要感谢那些无偿为开源社区做贡献的开发人员，HBS 是建构在非常多优秀的开源项目之上的，尤其是 HBase 和 Zookeeper。

参考文献

[1] The Apache Software Foundation. Apache HBase[EB/OL]. https://hbase.apache.org, May 11, 2021

[2] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 1-26

[3] D. Peng, F. Dabek. Large-scale incremental processing using distributed transactions and notifications[C]. USENIX Symposium on Operating Systems Design and Implementation, 2010

[4] J.C. Corebett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database[C]. USENIX Symposium on Operating Systems Design and Implementation, 2012, 261-264

[5] O. Shacham, F. Perez-Sorrosal, E. Bortnikov et al. Omid, Reloaded: Scalable and Highly-Available Transaction Processing[C]. 15th USENIX Conference on File and Storage Technologies, 2017, 167-180

[6] D.P. Reed. Naming and synchronization in a decentralized computer system[D]. Commonwealth of Massachusetts: Massachusetts Institute of Technology, 1978

[7] D.P. Reed. Implementing atomic actions on decentralized data[J]. ACM Transactions on Computer Systems (TOCS), 1983, 1(1): 3-23

[8] M. Yabandeh, D. Gómez Ferro. A critique of snapshot isolation[C]. Proceedings of the 7th ACM European conference on Computer Systems, 2012, 155-168

[9] H. Berenson, P. Bernstein, J. Gray, et al. A critique of ANSI SQL isolation levels[J]. ACM SIGMOD Record, 1995, 24(2): 1-10

[10] A. Fekete, D. Liarokapis, E. O'Neil, et al. Making snapshot isolation serializable[J]. ACM Transactions on Database Systems (TODS), 2005, 30(2): 492-528

[11] D.R.K Ports, K. Grittner. Serializable Snapshot Isolation in PostgreSQL[J]. Proceedings of the VLDB Endowment, 2012, 5(12): 1850-1861

[12] H.T. Kung, J.T. Robinson. On optimistic methods for concurrency control[J]. ACM Transactions on Database Systems (TODS), 1981, 6(2): 213-226

[13] The Apache Software Foundation. Apache Zookeeper[EB/OL]. https://zookeeper.apache.org/, May 11, 2021

[14] Cockroach Labs. CockroachDB[EB/OL]. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md, May 11, 2021

[15] M.J. Lai, E. Koontz, A. Purtell. Apache HBase Coprocessor Introduction[EB/OL]. https://blogs.apache.org/hbase/entry/coprocessor_introduction, May 11, 2021

[16] C. Mohan, D. Haderle, B. Lindsay, et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging[J]. ACM Transactions on Database Systems (TODS), 1992, 17(1): 94-162

[17] Y. Wu, J. Arulraj, J. Lin, et al. An empirical evaluation of in-memory multi-version concurrency control[J]. Proceedings of the VLDB Endowment, 2017, 10(7): 781-792

[18] T. Haerder, A. Reuter. Principles of transaction-oriented database recovery[J]. ACM computing surveys (CSUR), 1983, 15(4): 287-317

[19] J. Gray. The transaction concept: Virtues and limitations[C]. VLDB, 1981, 81: 144-154

[20] P.A. Bernstein, N. Goodman. Concurrency control in distributed database systems[J]. ACM Computing Surveys (CSUR), 1981, 13(2): 185-221

外文资料原文

Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google's indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It's easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

Now, consider how to update that index after recrawling some small portion of the web. It's not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce's scalability makes this approach feasible, and, in fact, Google's web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can't handle the sheer volume of data: Google's indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don't provide tools to help programmers maintain data invariants in the face of concurrent updates.

An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-

外文资料译文

基于分布式事务和通知的大规模增量处理，作者 Daniel Peng 和 Frank Dabek，1-2 页。

在抓取文档时更新 Web 索引需要随着新文档的到来不断转换现有文档的大型仓库。这个任务是一类数据处理任务的一个示例，这类任务通过小的、独立的更新来转换大型数据仓库。这些任务不在现有基础架构提供的功能之中。数据库不满足这些任务的存储或吞吐量要求：Google 的索引系统每天存储数十 PB 的数据，在数千台计算机上处理数亿次的更新。 MapReduce 和其他批处理系统无法单独处理较小的更新，因为它们需要创建较大的批次来提高效率。

我们构建了 Percolator，这是一个用于增量处理大型数据集更新的系统。它被用以创建 Google Web 搜索索引。通过使用基于 Percolator 的增量处理的索引系统替换基于批处理的索引系统，我们每天可以处理相同数量的文档，同时将 Google 搜索结果中文档的平均使用期限降低了 50％。

考虑建立可用于查询的网络索引的任务。索引系统对 Web 上的每个页面进行爬虫访问并进行处理，同时在索引上保留了一组不变集合。例如，如果在多个 URL 下抓取相同的内容，则只有 PageRank 最高的 URL 才会出现在索引中。每个链接也被反转，以便每个出站链接的锚文本都附加到该链接所指向的页面。链接反转可以消除重复项：如果有必要，指向页面重复项的链接应转发到最高的 PageRank 副本。

这是一项批量处理任务，可以表示为一系列 MapReduce 操作：一个用于聚集重复项，一个用于链接反转，等等。由于 MapReduce 限制了并行计算，保持不变集合并不容易；所有文档都需要先被一个处理步骤，然后再开始下一个处理步骤。例如，当索引系统正在写一个链接到当前最高 PageRank URL 的反向链接时，我们不必担心其 PageRank 会同时发生变化。因为先前的 MapReduce 步骤已经确定了它的 PageRank。

现在，请考虑在重新爬取一小部分 Web 后如何更新该索引。仅仅为新页面运行 MapReduces 是不够的，例如，新页面与网络的其余部分之间存在链接。我们必须在整个存储库（包括新页面和旧页面）上再次运行 MapReduces。只要有足够的计算资源，MapReduce 的可扩展性就可以使这种方法可行。实际上，Google 的网络搜索索引是在此处所述工作之前就是以这种方式生成的。但是，重新处理整个 Web 会丢弃早期已经完成的工作，并这使系统的延迟与存储库的大小成正比，而

不是与更新的大小成正比。

索引系统可以将文档仓库存储在 DBMS 中，并在使用事务维护不变集的同时更新各个文档。但是，现有的 DBMS 无法处理这样庞大的数据量：Google 的索引系统在数千台计算机上存储了数 PB 的数据。只有 Bigtable 这样的分布式存储系统才能容纳下我们的文档仓库，但是没有提供工具来帮助程序员面对并发更新时保持数据不变性。

一个理想用于维护 Web 搜索索引的任务的数据处理系统，需要针对增量处理进行优化。也就是说，我们既能够维护非常大的文档存储库，又能在抓取到新文档时候有效地对其进行更新。这个系统将同时处理许多小更新，并且将提供机制以便在并发更新的同时保持不变性，和跟踪已处理的更新。

本文的其余部分描述了一种特定的增量处理系统：Percolator。Percolator 为用户提供了对多 PB 仓库的随机访问能力。随机访问使我们可以分开地处理文档，避免 MapReduce 所需的全局扫描。为了获得高吞吐量，许多机器上的许多线程需要并发地处理数据，因此 Percolator 提供了 ACID 语义的事务支持，从而使程序员更容易推断数据库的状态。我们目前实现了快照隔离语义。

除了对并发进行推理外，使用增量系统的程序员还需要跟踪增量计算任务的状态。Percolator 提供了观察者以便完成这样的工作：每当用户指定的列更改时，系统都会调用的对应代码段。Percolator 应用程序由一系列观察器组成。每个观察者通过把结果写在表中来触发“下游”的观察者开始工作。外部程序通过将初始数据写入表中来触发链中的第一个观察者。

Percolator 专为增量处理而构建，无意取代大多数数据处理任务的现有解决方案。 MapReduce 可以更好地处理无法将结果细分为较小更新（例如，对文件进行排序）的计算。另外，Percolator 处理的计算应具有很强的一致性要求；否则，Bigtable 就足够了。最后，计算处理的数据在某些维度上应该非常大（总数据大小，转换所需的 CPU 等）；否则，传统 DBMS 可以处理不适合 MapReduce 或 Bigtable 的较小计算。

在 Google 内部，Percolator 的主要应用程序是处理实时 Web 搜索索引。通过将索引系统转换为增量系统，我们可以在抓取单个文档时对其进行处理。这将平均文档处理延迟减少了 100 倍，并且出现在搜索结果中的文档的平均年龄下降了近 50%（搜索结果的年龄包括除索引之外的其他延迟，例如在文档被更改到被抓取之间的时间）。该系统还用于将页面渲染为图像。Percolator 跟踪网页和它们依赖的资源之间的关系，因此，当任何依赖的资源发生变化时，都可以对页面进行重新处理。