易鲸捷 DTM 技术白皮书 1.0.0

2020/1

版权

公告

本文档包含的信息如有更改，恕不另行通知。

易鲸捷产品和服务附带的正式担保声明中规定的担保是该产品和服务享有的唯一担保。本文中的任何信息均不构成额外的保修条款。

声明

Microsoft® 和 Windows® 是美国微软公司的注册商标。Java® 和 MySQL® 是 Oracle 及其子公司的注册商标。Bosun 是 Stack Exchange 的商标。

Apache®、Hadoop®、HBase®、Hive®、openTSDB®、Sqoop® 和 Trafodion® 是 Apache 软件基金会的商标。Esgyn，EsgynDB 和 QianBase 是易鲸捷的商标。

目录

前言

本文简介

本文介绍了分布式事务管理体系结构，以及如何将其用于 DTM 提供的各种功能

目标读者

本指南的目标读者为易鲸捷数据库管理员和用户。

修订历史

版本	日期	说明
1.0.0	2020 年 1 月	第一版本

批评与建议

我们支持您对本指南做出的任何批评与建议，并尽力提供符合您需求的文档。

若您发现任何错误、或有任何改进建议，请发邮件至 support@esgyn.cn。

1. 介绍易鲸捷数据库分布式事务管理

易鲸捷公司的 EsgynDB 和 QianBase 产品（以下简称易鲸捷数据库）旨在提供企业级的 SQL-on-HBase DBMS 引擎，该引擎专门针对受事务保护的运营型工作负载。 它们代表了 HBase 和事务型 SQL 技术的结合，充分利用了 Tandem Computers 和 HP 在数据库技术和解决方案方面超过 25 年的投资。Tandem NonStop 将专业知识和对分布式事务管理（DTM）的投资用于高并发，低延迟和混合工作负载环境并带给了 HBase。易鲸捷 DTM 专为无限规模而设计，可支持大量并发的短期运营更新以及长期运行的运营报告查询。

企业级的 SQL on HBase DBMS 引擎专门针对大数据事务型或运营型工作负载，而不是分析型工作负载。事务型 SQL 包含先前描述为 OLTP（在线事务处理）的工作负载。OLTP 的开发是为了支持传统企业级事务应用程序（ERP，CRM 等）和企业业务流程。此外，交易已经发展为既包括结构化又有半结构化的社交和手机数据的交互以及观察。

为了支持跨结构化和半结构化数据的事务处理工作量，易鲸捷数据库拥有分布式事务管理基础架构，该基础结构将促进在易鲸捷数据库表中的结构化数据以及 HBase 中的半结构化数据中跨越多行，多表和多语句的事务更新。HBase 是易鲸捷数据库的存储引擎。因此，易鲸捷数据库表也是 HBase 表。

易鲸捷数据库中可用的初始事务型能支持：

- 多版本并发控制（MVCC）非锁定乐观并发控制算法
  - 读取从未被阻止
  - 事务看不到其他活动事务所做的更改
  - 冲突检查在提交时完成，并且如果事务与已提交的事务发生冲突，则事务中止
  - 应用程序需要重试由于冲突而中止的事务
- 开始工作，提交工作，回滚工作，设置交易
  - 如果未使用 BEGIN WORK 发起事务，则 SQL 默认为 AUTOCOMMIT
- READ COMMITTED 作为事务隔离级别，这也是默认设置
  - 比 ANSI SQL READ COMMITTED 强，因为它禁止丢失更新:
    - 如果 T1 读取一行，然后 T2 更新该行，然后 T1 根据其先前的读取和提交更新同一行，则 T2 在尝试提交时将因冲突而中止，因为第一个提交者在 DTM 中获胜
  - 等效于 READ COMMITTED，因为它不支持重复读取和读取偏斜
    - 读取偏斜是指 T1 读取 x，然后 T2 更新 x 和 y 并提交，然后 T1 读取 y。在这种情况下，y 的值在 T2 更新之后，而 x 的值在 T2 更新之前，因此与 y 不一致。
- 多个 SQL 进程同时参与同一事务
  - 例如由事务启动进程启动的多个并行插入进程，例如用于大型并行 INSERT…SELECT 操作
- Region Server，事务管理器或节点故障后恢复

- 跨 HBase 区域的事务完整性和区域平衡

未来发布计划提供：

- 优化
  - 单一操作事务仅限于单个 region
  - 更高效的事务流，尤其是当事务只涉及一个进程时
- 隔离支持
  - 可重复读取隔离
  - 快照隔离
  - First writer wins
  - 序列化快照隔离
- 某些表的行锁定范例
  - 涉及长时间运行事务的表
  - 悲观锁定，用于高度并发的操作
- 高可用性
  - 集群的动态扩展和收缩
  - 事务期间处理区域的分割/再平衡
  - 处理复杂的故障情况
  - 数据中心之间的同步和异步主动-被动和主动-主动支持
- 用于事务管理和指标的 RESTful API

易鲸捷数据库客户和社区将推动决定以上工作的优先级。

2. 可扩展架构

分布式事务管理系统需要支持大型群集，其中潜在的数千个节点运行着各种各样的并发工作负载，其中包括:

- 短期更新事务
- 长期运行的批处理更新
- OLTP 查询，BI 查询，加载，提取和汇总

设计用于这种环境的体系结构的主要目标之一是按规模设计。规模:

- 节点数
- 不同规模的并发事务数
- 不同复杂程度的查询数
- 数据量

任何进程或节点都不能成为瓶颈。该架构在各个方面都具有可扩展性。

另一个重要的设计点是高可用性。单节点故障不会影响整个集群中运行的所有事务。恢复时间必须最短。这意味着恢复需要跨 region server 并行完成。在构建分布式事务管理系统时，应考虑最高的可用性和一致性目标。

这就是我们使用易鲸捷数据库 分布式事务管理构建的!

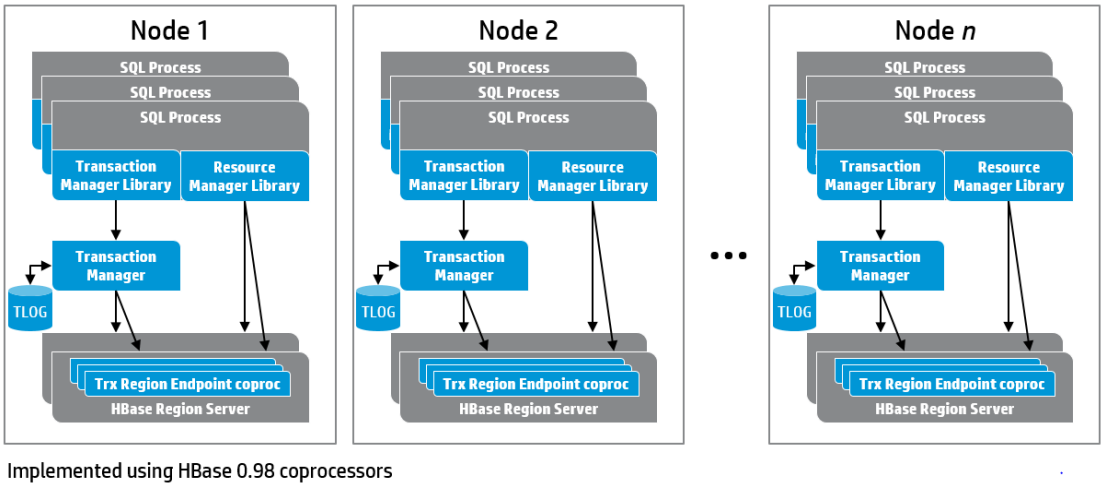

图 1. 可扩展架构

在图 1 中，您可以看到 DTM 体系结构的设计是可扩展的：

- 在单个节点上运行的多个 SQL 进程可以通过链接的**资源管理器库**和**事务管理器库**利用事务支持
- 每个节点上的**事务管理器**管理由该节点上的 SQL 进程启动的事务，从而跨多个 region servers 中托管的多个 regions 协调事务。换句话说，随集群的大小**扩展**。
- **协处理器代码**管理 region server 负责的每个 region 的事务更新和一致性。

在这种体系结构中，与事务管理相关的工作是真正分散的。发起事务的多个进程与本地**事务管理器（TM）**进行对话以协调事务。TM 仅参与协调事务的启动，提交，中止，日志和恢复操作。它不会跟踪所有事务更新或针对 HBase 区域进行读取。这些被委派给区域。所以它的重量很轻。TM 是多线程的，可以**扩展**，并根据需要通过线程并行处理尽可能多的事务。如果认为有必要进行**扩展**，则每个物理节点还可以定义多个逻辑节点，每个逻辑节点都有自己的 TM 进程。

管理事务的工作与每个事务的实际更新是分离的。SQL 进程使用资源管理器库直接调用事务中涉及的 HBase 区域的 get/put/delete/getScanner 事务版本。也就是说，TM 不会参与跟踪这些调用或更新。

HBase region 还通过运行在 region server 中的 **Endpoint 协处理器**的线程以完全分布式的方式跟踪更新本身，以托管事务调用的 region。同样，协处理器不将与交易有关的更新传达给 TM。它与 TM 的对话非常少，仅在事务协调活动（例如提交或中止）时进行对话。

这表明该体系结构旨在进行横向扩展和纵向扩展，以满足超大型集群中任何工作负载的需求。

3. 解析组件架构

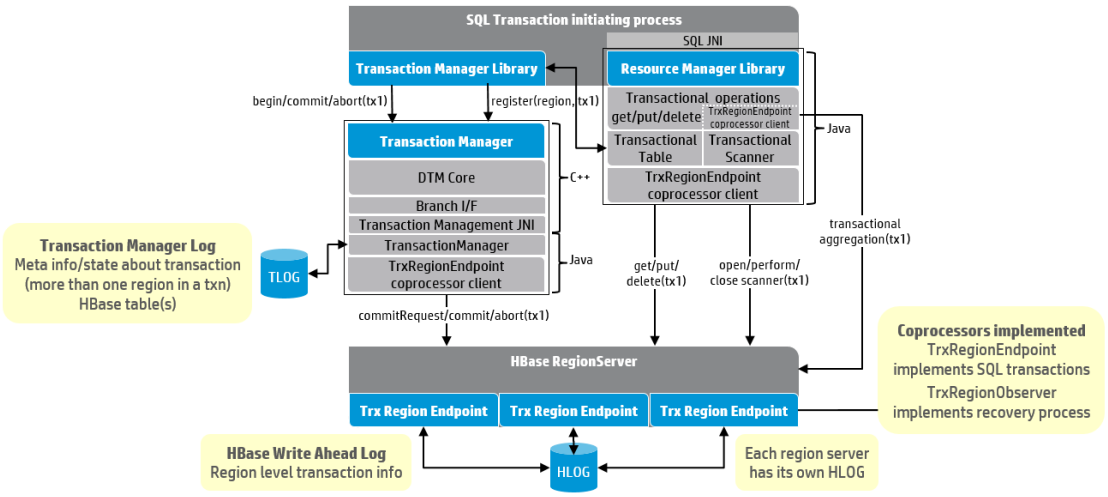

图 2. 解析组件架构

在解析组件架构时，您会看到首先是 SQL 事务启动过进程。这是易鲸捷数据库 SQL 代码（从现在开始称为 SQL），例如易鲸捷数据库命令接口 SQLCI 或 TRAFCI 或 Master Executor 服务器进程，它编译查询并代表客户端执行该查询，然后收集结果并将其返回给客户端。在易鲸捷数据库中，为了获得查询内并行性，可以在多个节点上生成 Executor 服务器进程（ESP），以代表查询运行。这些进程从主执行器继承事务 ID，并且也直接将事务性的 get / put / delete / getScanner 调用直接发送到 HBase 区域。

事务管理器库简化了从 SQL 应用程序到本地**事务管理器**（TM）进程的开始，注册，准备，提交和中止请求。将来会提供 XA 支持。 XA 是用于分布式事务处理的 Open Group 标准规范。TM 库是使用 Seabed 与 TM 进程进行通信的 C ++库。将来，可以很容易地用 Java 重写它，这样，除了易鲸捷数据库之外，那些希望在 HBase 上支持分布式事务的产品也可以利用整个分布式事务基础结构。

Seabed 用于在 SQL 和 TM 之间进行事务控制（开始/提交/中止/状态/恢复/挂起）消息，并用于参与事务的 SQL 组件之间的事务传播。例如，前面提到的 ESP。

当参与者（包括事务初学者）希望执行工作（例如事务中的 put）时，资源管理器库会通过 JNI 接口调用 TM 库中的 **registerRegion**。需要 JNI 接口，因为 RM 库在 Java 中，而 TM 库在 C ++中。然后，TM 库将 registerRegion 请求作为 seabed 消息发送给 TM。这对于维护参与交易的所有区域的主列表是必要的。提交和恢复处理需要了解参与事务的所有区域。每个事务区域组合只对一次 registerRegion 调用一次。它还为初学者以外的其他参与者（例如 ESP）提供了一种隐式方式来加入交易。如果参与的过程在事务期间失败，并且 TM 需要知道以便启动中止，这就很重要,。

资源管理器（RM）库是一个 Java 库，可简化从 SQL 到 HBase 的事务性 get / put / delete / getScanner 调用。HBase 是被管理的资源。它使用 64 位事务 ID 作为参数。

Region 协处理器 **TrxRegionEndpoint** 是 RM 的实现，它负责处理所有事务对 Region 的事务性更新请求。，并跟踪任何事务针对该区域执行的所有事务更新。 **TrxRegionObserver** 实现恢复，如稍后所述。

尽管这些图显示了这些组件的更详细细分，但它们与本文的相关性并不高。它们是为那些需要比本文提供更多细节的开发人员准备的。

为了确保完整性并促进恢复，涉及两个**预写日志**（WAL）：

- 事务管理器日志 **TLOG** 维护有关 TM 管理的每个事务的元信息和状态。根

据扩展要求，可以有一个或多个用于该日志记录的 HBase 表。

- HBase WAL **HLOG** 是 HBase 的一部分，它维护对 HBase 所做的所有更新。每个 region server 中有一个。属于 DTM 的 TrxRegionEndpoint 代码也可以在 HLOG 中维护区域级别的事务信息。它用于存储与事务相关的状态，例如在提交的准备阶段之前进行的更新。

  让我们看看在事务的生命周期中如何利用这些组件。

4. BEGIN WORK

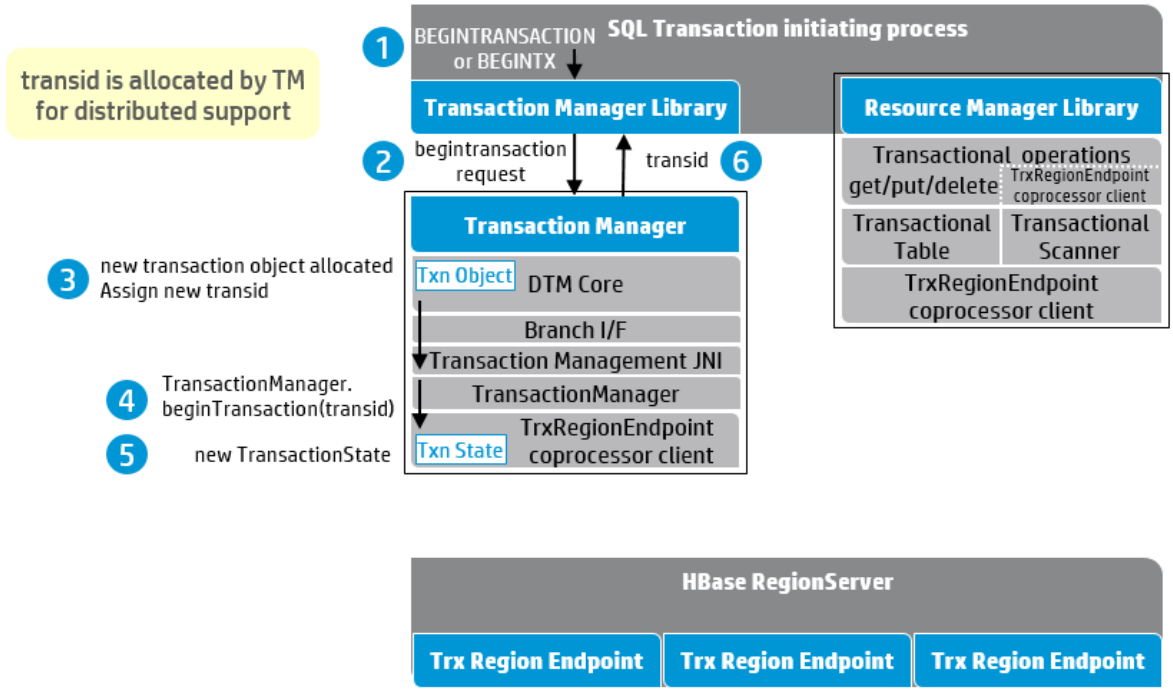

图 3. BEGIN WORK – BEGINTRANSACTION

应用程序要做的第一件事是使用 BEGIN WORK 命令开始事务。这导致 DTM 基础结构中发生以下事件序列。本质上，此过程会生成事务 ID，并在 TM 中创建事务对象和状态，以便 TM 可以在其整个生命周期内管理事务。

❶ SQL 当使用事务管理库启动 BEGIN WORK 时，SQL 代表应用程序调用 BEGINTRANSACTION

❷ Library TM 库将 begintransaction 请求发送到在其本地节点上运行的 Transaction Manager 进程

❸ TM 进程实例化一个新的事务对象并分配一个新的事务 ID（transid）。此事务 ID 用于跨区域与该交易关联的更新。它有助于将参与区域中与该交易相关的所有更新结合在一起

❹, ❺ & ❻ TM 然后处理开始事务，实例化一个新的 TransactionState 对象，以

在 TrxRegionEndpoint 协处理器客户端中表示该事务。该客户端与 HBase RegionServer TrxRegionEndpoint 协处理器进行通信，以在 HBase 区域级别管理该事务

❼ TM 将事务 ID 返回到 TM 库。现在，SQL 在执行该事务的一部分时执行 get, put 和 delete 操作时可以使用该事务 ID。

4.1. 事务型 get / put / delete

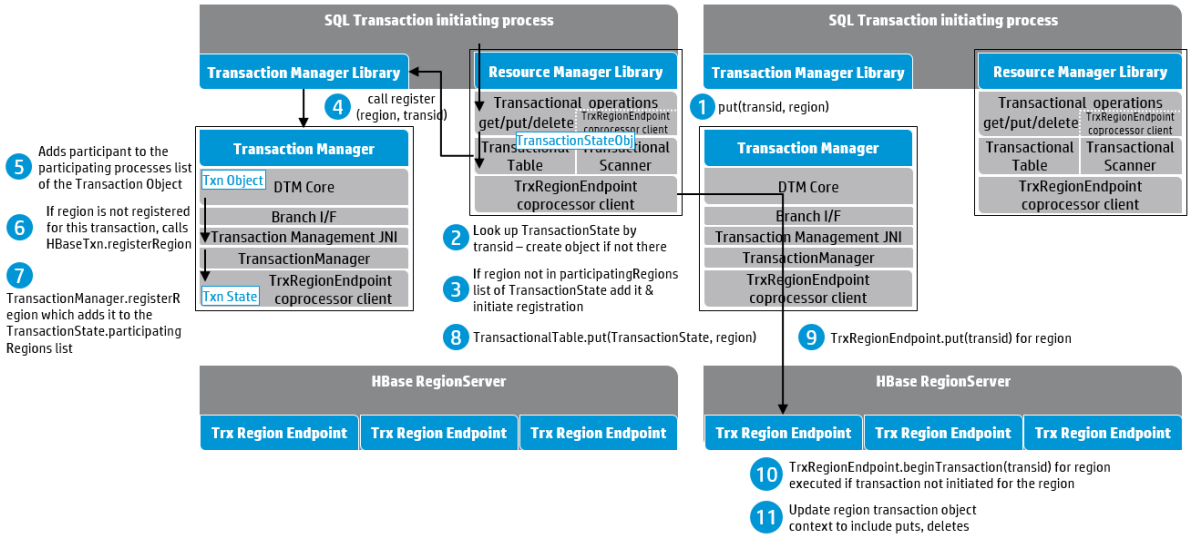

图 4. get / put / delete

现在，应用程序已开始新事务，它可以开始使用此事务 ID 进行 get, put 或 delete。它通过资源管理器库调用这些调用的事务版本。它将事务 ID 与这些调用一起传递。在此示例中，我们假设应用程序正在执行 put 操作。随后发生以下事件序列：

❶ SQL 引擎是根据 HBase 行键确定事务中需要涉及哪个区域的引擎。SQL 用事务 ID SQL 调用 RM 库的事务性 put。

❷ RM 会查看是否已经为此事务 ID 创建了 TransactionState 对象，该对象可能

是由与该事务相关的先前更新启动的。如果不存在，它将创建它。

❸然后，RM 会查看正在更新的 HBase 区域是否已经在该事务的参与区域列表中，可能是由与该区域关联的同一交易 ID 的先前更新添加的。如果该区域不在列表中，则将其添加，然后启动注册过程，以向 TM 注册该区域作为该交易的参与区域。 TM 需要知道这一点，以便对于该事务的后续 COMMIT 或 ABORT，它可以与该事务涉及的适当区域进行通信。

❹RM 通过调用 TM 库中的注册，并向其传递交易 ID 和区域来启动此注册过程。TM 库将注册请求发送到 TM。

❺ TM 将参与的 SQL 进程添加到事务对象的参与的进程列表中

❻ & ❼如果未为此事务注册区域，则 TM 将将该区域添加到事务状态对象的参与区域列表中。该区域可能已经被另一个 SQL 进程注册，也参与了同一事务并更新了相同的 HBase 区域。

❽ & ❾一旦完成此注册过程，RM 中的 TrxRegionEndpoint 协处理器客户端就会向 TrxRegionEndpoint 发起事务型 put 到 put 操作定向到的区域。

❿如果此区域以前从未看到过此事务 ID，则 TrxRegionEndpoint 将执行开始事务。为该事务创建一个事务对象，以跟踪该事务的所有 HBase 区域级别上下文。这是一种优化，其中区域在接收到第一次更新时会包含在事务中，而不是 TM 将开始事务传达给事务中涉及的所有区域。

⓫ 事务上下文中的 writeOrdering 列表已更新，以包括 RM 代表该事务传达给该区域的所有更新。在这种情况下，与 put 相关的更新将在此内存上下文中维护。尚未将任何内容写到 HBase 表（HBase WAL 或内存存储）。这将在提交的

第一（准备）阶段完成。

4.2. 参与单个交易的多个区域和流程

DTM 的与众不同之处之一是它支持不仅跨越区域而且跨越进程的事务的能力。上面使用的示例显示了一个 put。 但是，同一 SQL 进程可以在同一事务中更新同一表的多个区域，甚至多个表的多个区域。 如下所示。

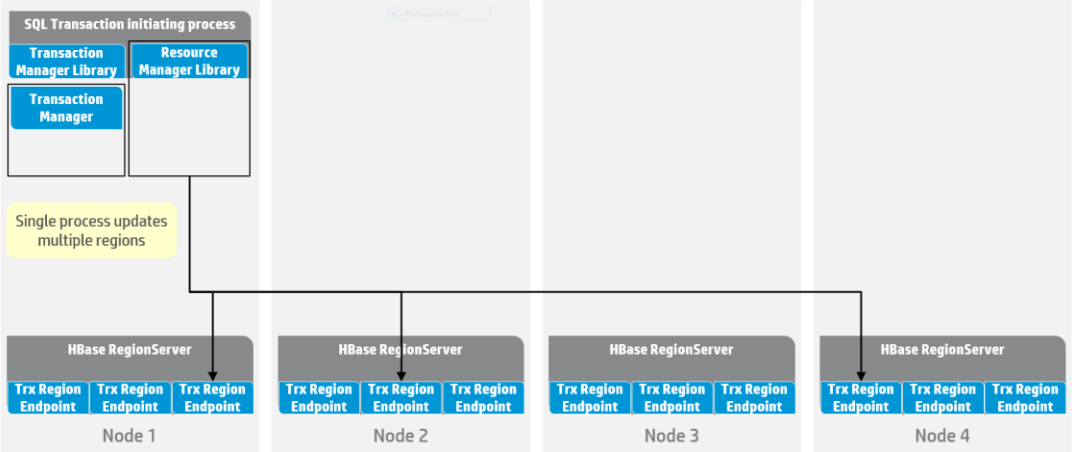

图 5. get / put / delete –单个流程完成多区域更新

您也可以让其他进程参与同一事务。如下所示，由于事务正在处理的更新量很大，因此事务启动 SQL 进程可能需要并行化更新，例如在行集更新或并行化的 INSERT…SELECT 中。 在这种情况下，如在易鲸捷数据库中一样，主执行程序进程（即事务启动进程）可以使用多个进程来并行更新。它将事务 ID 传递给这些进程。这些过程可以通过向 TM 注册自己来加入事务。在这些进程中将处理事务的提交，中止和恢复。

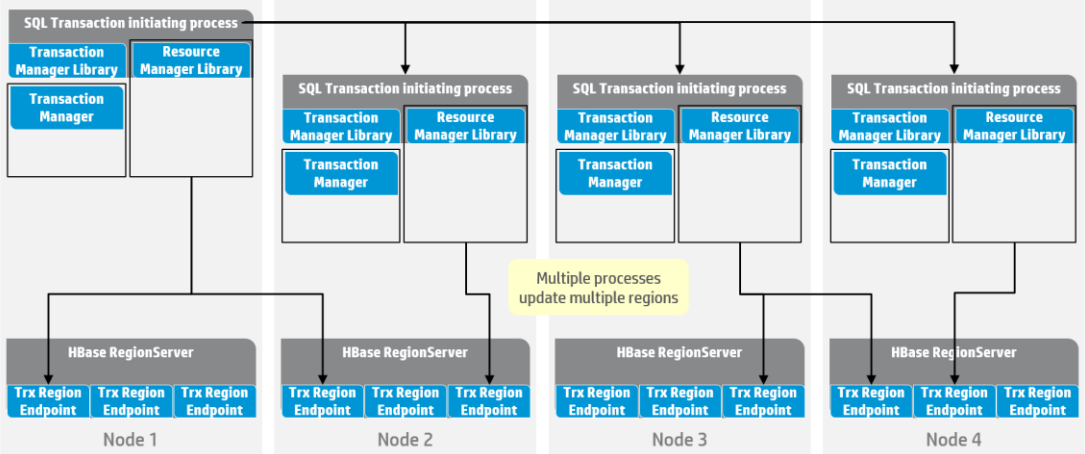

图 6. get / put / delete – 多个流程完成多区域更新

5. COMMIT WORK

COMMIT WORK（或 ENDTRANSACTION）遵循两阶段提交协议。 有一个 prepare 阶段，在此阶段，TM 从参与准备好要提交的事务的所有区域中获得承诺，然后是完成事务的 commit 阶段。

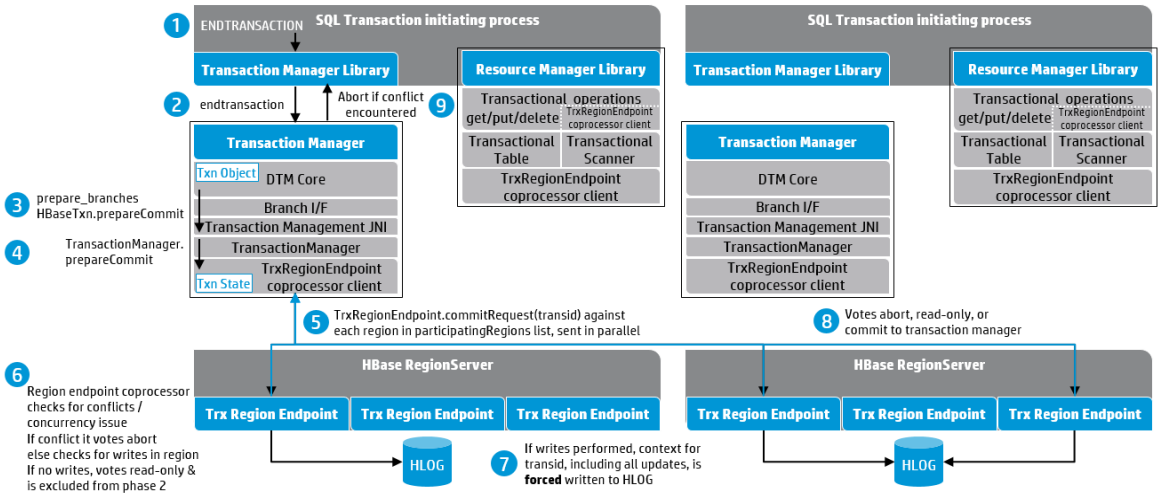

图 7. COMMIT WORK – ENDTRANSACTION – 阶段 1, prepare

以下事件序列是阶段 1 或 **prepare** 阶段的一部分：

❶ SQL 当使用事务管理库启动 COMMIT WORK 时，SQL 代表应用程序调用 ENDTRANSACTION。

❷ Library TM 库使用 Seabed 将 endtransaction 请求发送到在其本地节点上运行的 Transaction Manager 进程

❸, ❹ & ❺ TM 处理 prepareCommit，然后通过 TrxRegionEndpoint 协处理器客户端，将其 prepare 请求事务并行发送给 HBase RegionServer TrxRegionEndpoint 协处理器，以针对其参与交易的参与区域列表中的每个区域。

❻一旦 HBase 区域的 TrxRegionEndpoint 协处理器接收到 prepare 请求，它将检

查与其他事务的冲突或并发问题。在该事务读取了该行之后，它将检查是否有其他事务提交了对该事务要更新的任何行的更改。在更新冲突的事务中，首先提交的事务将先处理。

附带说明一下，由于首先提交的事务先被处理，因此系统会保留提交事务的事务上下文，直到在该提交时间之前没有任何正在运行的事务为止。这是为了便于冲突检查可能正在尝试更新此事务已更改并已提交的行的事务。

Prepare 被串行化。如果允许并发 prepare，则它们各自可以基于先前完成的事务确定它们没有任何冲突，而实际上它们可能彼此冲突。

如果存在冲突，则 TrxRegionEndpoint 投票中止并将中止传达回 TM。如果没有冲突，并且该事务在该区域中没有任何写入，则协处理器将只读表决权回给 TM，以将其自身排除在提交的第二阶段之外。

将来，如果有更新操作，RM 就有机会仅向 TM 注册一个区域。事务区域终端仍将知道事务读取操作以支持读取隔离级别。当前，涉及这些区域，因此在提交时它们可以释放这些只读事务的上下文。 SQL 进程可以使用轻量级进程将事务提交传达到这些只读区域。当区域仅参与读取时，这可以大大减少向 TM 注册区域的开销，并且 TM 可以在提交时与这些区域进行通信。

❼ 如果事务在该区域中具有更新的行，则将事务上下文以及到目前为止已跟踪的所有更新（仅在内存中）强制写入 HBase WAL（HLOG)。事务上下文使用 WAL 标记写入 HLOG，从而使其对 HBase 透明。现在可以在恢复期间使用此上下文。在写完上下文和对 HBase 的更新后，TrxRegionEndpoint 响应 TM，准备提交。

❽ & ❾ TM 整理所有异常中止的，只读的并准备提交事务涉及区域的响应。如果有任何区域投票决定中止，则启动中止交易过程。

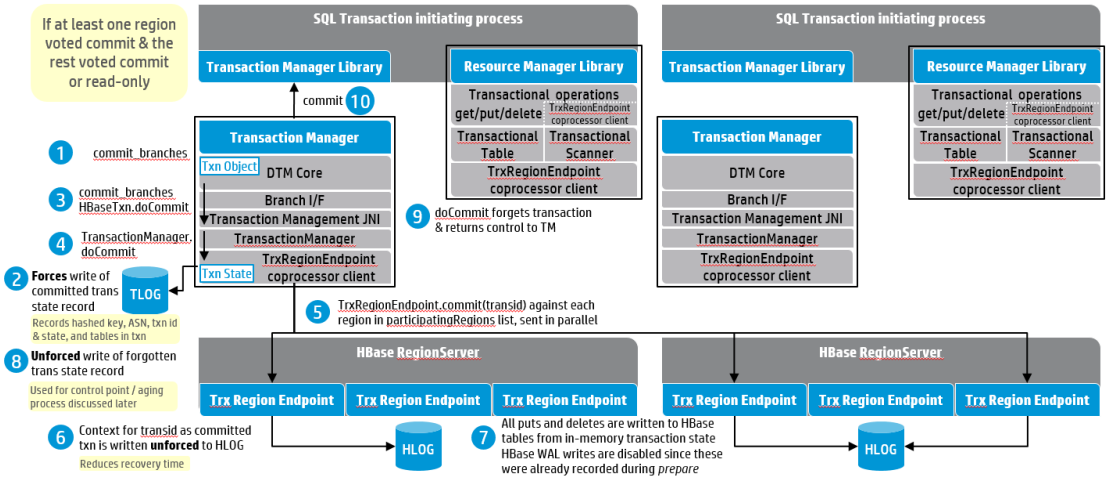

图 8. COMMIT WORK – ENDTRANSACTION – 阶段 2, commit

如果至少一个区域投票 commit，而其余区域投票 commit 或 read-only，则将启动 COMMIT WORK 的阶段 2。以下事件序列是阶段 2 或 commit 阶段的一部分：

❶, ❸ & ❹TM 已从参与事务的区域收到所有响应，评估事务可以在 commit 阶段继续进行并启动提交处理

❷ TM 作为 commit 处理的一部分要做的第一件事是强制将已提交的事务状态记录写入 TLOG。它记录事务 ID，事务状态以及事务中涉及的表，以及哈希键和指示日志写序列号（ASN）的单调递增值，如果恢复阶段使用必要。记录表而不记录区域，因为到恢复时该表的 HBase 区域可能已更改。

❺然后，TM 通过 TrxRegionEndpoint 协处理器客户端，将该 commit 并行传递给 HBase RegionServer TrxRegionEndpoint 协处理器，以用于其在该交易的参与区域列表中具有的每个区域。

❻此时，TrxRegionEndpoint 协处理器将事务 ID 的上下文（指示已提交的状态）写入 HLOG。但是这种写入是没有强制性的，因为其主要目的是减少发生故障时的恢复时间。它清楚地表明需要将事务前滚（应用更新）到已提交状态。如果此信息不可用，则在恢复时，该地区必须让 TM 参与解决此不确定交易。

❼此时，更新（puts 和 deletes）将从内存中的事务状态上下文写到 HBase 表中。不会强制执行这些写操作，因为如果发生故障，恢复过程可以从 commit 的 prepare 阶段中已写出的上下文中重新应用这些更新。 HBase 表更新是常规的 HBase 更新，带有由 HBase 记录的版本时间戳。也就是说，DTM 不会篡改时间戳来存储事务 ID，就像其他某些事务管理器一样。这些更新将记录到 HBase 的内存存储中，并与所有 HBase 更新一样，每隔一段时间就会刷新到 HFiles。对于这些更新， HBase WAL 写入操作被禁止，因为这些已在准备阶段记录在 WAL 中（请参见 prepare 阶段的步骤 7)。

❽在所有区域都答复 TM 他们已经完成步骤 7 之后，TM 会将未强制的遗忘的事务状态记录写入 TLOG。这对于恢复很重要。如果某个区域的停机时间较长，则可能导致已提交的事务长时间保留在 TM 中。该记录用于控制点/老化过程（稍后讨论)。

❾ & ❿ 此时，TM 会忘记事务，删除事务对象和状态，并向 TM 库/或 SQL 事务启动过程返回 commit 确认。

6. ABORT WORK

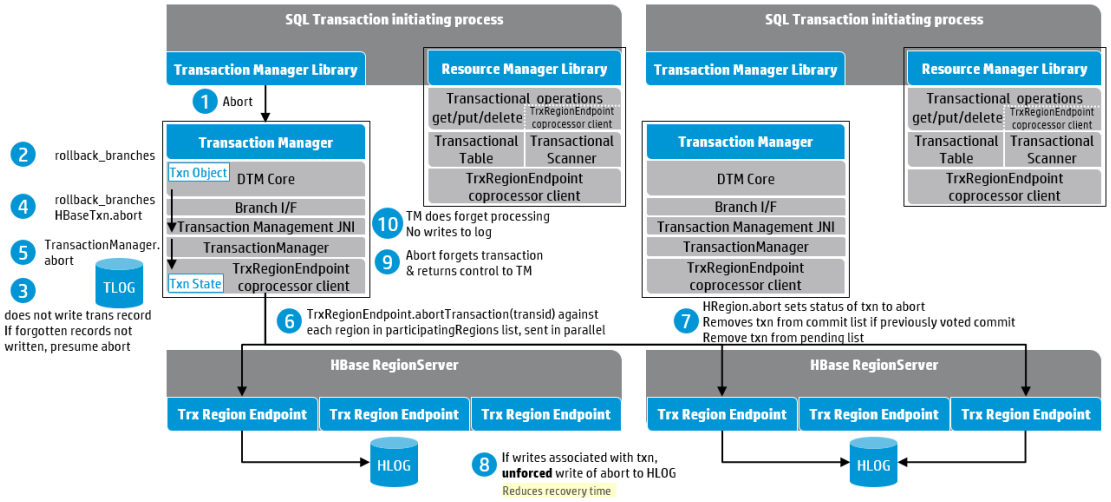

图 9. ABORT WORK – ABORTTRANSACTION

如果在提交事务时发生冲突，并且 DTM 需要中止该事务，或者应用程序发出了 ABORT WORK，则启动 ABORTTRANSACTION。以下事件序列是中止的一部分：

❶一旦应用程序启动中止，TM 库就会将该中止传达给 TM

❷, ❹ & ❺ TM 启动中止或回滚处理

❸与 commit 相反，TM 不会将事务记录写入 TLOG 以中止。这是一个优化。如果未将忘记的记录写入事务，则 TM 会假定该事务已中止。

❻然后，TM 通过 TrxRegionEndpoint 协处理器客户端，将中止事务并行传输到 HBase RegionServer TrxRegionEndpoint 协处理器，以针对其在该事务的参与区域列表中具有的每个区域。

❼此时，TrxRegionEndpoint 协处理器将事务状态设置为中止。如果该区域先前已对提交进行投票，则它将事务从提交列表和提交挂起列表中删除。

❽如果写入与事务关联，则将非强制中止的写入被写入 HLOG。如果写入，它

将减少恢复时间，因为该区域可以非常快速地确定事务已中止，并且不必将更新应用于 HBase。如果写入丢失，则必须从 TM 中找出事务的状态，这会稍微增加恢复时间。

❾ & ❿ 此时，TM 会忘记交易，并删除交易对象和状态。它不会将事务被遗忘的记录写入 TLOG。

7. 交易恢复

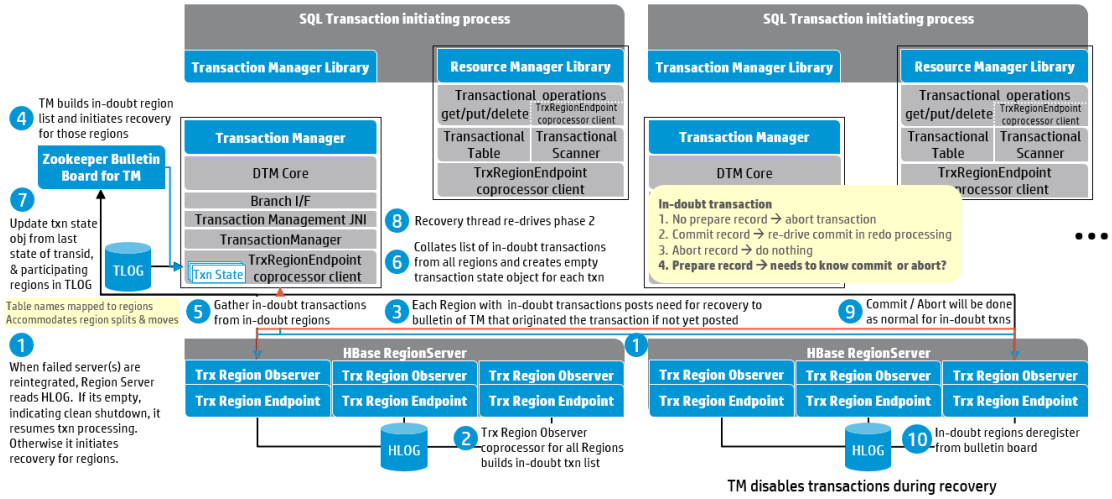

图 10. 交易恢复

如果节点或 region server 发生故障，则 HBase 会启动恢复。作为该恢复过程的一部分，将启动 DTM 恢复过程。以下是恢复步骤：

❶故障服务器修复后，region server 将读取 HBase WAL（HLOG）。如果日志为空，则表明关机是干净的，无需恢复。此时，region server 将恢复事务处理，并且不需要区域恢复。否则，将继续进行下一步。

❷ 事务区域观察者协处理器是作为恢复过程的一部分启动的。对于每个需要恢复的区域，观察者都会列出所有不确定且需要解决的交易。

- 如果 HLOG 中没有事务的 prepare 记录，则要不该区域从不参与事务，要不在提交的准备阶段，在写入事务上下文记录之前发生故障。该地区将对该交易不予理会，因此不可能将其放入其不确定交易列表中。
- 如果在 HLOG 中找到事务的 commit 记录，则该区域知道它应该重新驱动提交重做处理，以确保实际上已将更新提交给 HBase。
- 请记住，commit 记录是在 commit 处理期间进行的非强制写入，并且如果

失败阻止了该写入，则在 HLOG 中可能不存在该记录。在这种情况下，如果观察者遇到 prepare 记录但没有 commit 记录，则它必须通过不确定事务列表将其传达给 TM，以查明该事务是被提交还是中止。

- 如果遇到 abort 记录，则该区域不必为此事务执行任何操作，也不会使其进入不确定列表。同样，中止记录不是强制写入的，也可能没有写 HLOG。因此，与未遇到 commit 记录一样，该区域将需要找出事务是已提交还是中止，并将通过不确定事务列表将此信息传达给 TM。
- 总而言之，将事务添加到不确定事务列表中以传达给 TM 的唯一解决方法是遇到事务的 prepare 记录，但没有该事务的 commit 或 abort 记录。

❸ 如果某个区域遇到任何需要解决的交易，则它会通过发起这些交易的 TM 发出恢复需求。每个 TM 仅执行一次。然后，它在共享由这些 TM 发起的不确定事务之前，等待 TM 发起恢复。

❹ TM 将构建需要解决的区域列表，以进行不确定的交易并启动恢复。

❺ 然后，TM 通过其发起的不确定交易与每个区域进行通信，并从那些参与区域中获取这些交易的列表。

❻ 然后，TM 整理此不确定事务列表和需要解决这些事务的区域，并为每个事务创建空的事务状态对象。

❼然后，TM 读取 TLOG 并从为每个事务编写的最后一个状态记录中更新事务状态对象。在事务提交的准备阶段，当写入 TLOG 事务状态记录时，将写入与事务有关的表。写出的不是参与区域。其原因是，到恢复之时，该地区可能已经分裂或移动。因此，在恢复时，这些表名将映射到当前区域，以便可以使用

正确的参与区域来更新事务状态。

❽ & ❾此时，TM 的恢复线程可以根据结果（提交或中止）驱动每个事务。它将对每个不确定事务及其参与区域调用提交或中止处理，就像通常处理提交和中止一样。

❿每个地区完成其不确定交易清单后，都会从 TM Zookeeper 公告板上注销。TM 会在此恢复阶段禁用所有事务。一旦完成了每个不确定事务的恢复过程，它就会恢复新的事务处理。

8. 控制点处理

DTM 将控制点用于两个目的：

- 确保区域保留所有交易更新
- 老化不再需要的 TLOG 条目

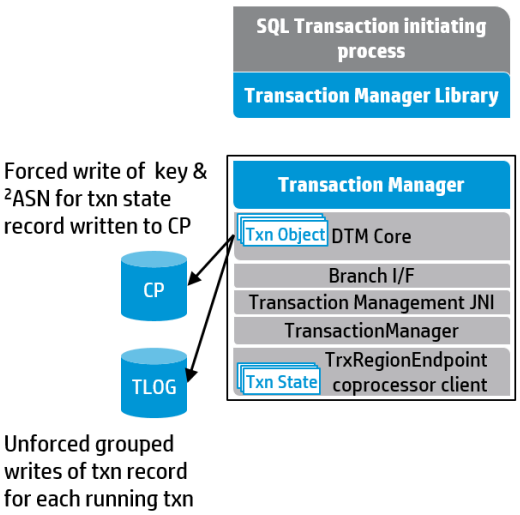

图 11. 交易恢复 – 控制点处理

8.1. 确保区域保留所有交易更新

控制点处理可确保在所有区域都执行了提交工作并将其持久化之前，不会忘记已提交的事务。为方便起见，将控制点写到 TLOG 随附的控制点 HBase 表中。这些是在可配置的时间间隔（默认为两分钟）内被强制写出的。作为强制写入 TLOG 的提交事务记录的一部分，还将写出指示审计写序列号（ASN）的单调递增值。控制点间隔由两个连续的控制点记录表示，其中第一个记录从间隔的开头定义 ASN，第二个记录在间隔的末尾定义 ASN。因此，实质上，此间隔涵盖了 TLOG 中的许多已提交事务记录。

控制点记录由 ASN 和密钥组成。同时，每个正在运行的事务的事务记录也被非强制写入 TLOG。这些可以加快事务历史的重建，或促进重做过程。如果不存在这些文件，则恢复过程必须返回更远的位置以驱动恢复。如前所述，这些事务提交的重新驱动是由区域的恢复需求驱动的。

8.2. 老化

可以老化两个控制点之前的 TLOG 条目，并且在控制点表中仅需要维护两个条目。 在老化旧条目时，在该控制点之前遇到的所有提交记录以及所有正在运行的事务的事务状态都将被重写到 TLOG 中。 仅当遇到被遗忘的交易记录时，交易状态记录才会过时，无需重写。 这样可确保 TLOG 仅包含所有驱动重新启动恢复过程所需的 TLOG 事务条目。

9. 高可用性

高可用性一直是我们传承的标志，也是我们所做工作不可或缺的一部分。因此，DTM 体系结构在构建时就考虑了高可用性也就不足为奇了。我们还进行了广泛的高可用性测试，以模拟所有可能的故障和恢复方案。高可用性是根据事务工作负载在出现故障时所能看到的 up time 来衡量的。因此，该体系结构不仅从可用性的角度，而且从对工作负载性能的影响的角度来评估和解决节点故障、DTM 进程故障、群集故障、磁盘丢失、region server 故障、HBase 区域拆分和重新平衡以及其他情况的影响。

在所有 TM 中，一个被指定为 Lead TM。当节点上的任何其他 TM 进程失败时，Lead TM 会立即知道它，并停止该节点上的所有易鲸捷数据库进程，从而模拟节点关闭情况。由该节点发起的所有事务，或涉及该节点的事务（即使在另一个节点上发起的事务）也将中止。 TM 进程将自动重新启动，然后负责解决该节点的任何不确定事务。如果 Lead TM 本身受到影响，则 Lead 序列中的下一个 TM 将接管 Lead 并启动此过程。

当 region server 发生故障时，所有关联的区域将由 HMaster 分配给另一个 region server。在新的 region server 中重新打开区域时，将启动标准区域恢复。作为恢复的一部分，也将启动 DTM 恢复，如前所述。在易鲸捷数据库中，可以大大减少 HBase 完成恢复之后 DTM 允许新事务之前的时间。 DTM 可以立即开始接受新事务。如果这些交易不受仍由 DTM 进行恢复的不确定交易的影响，则这些交易将成功完成。否则，他们将因冲突而中止。

物理节点故障类似于 TM 故障和 region server 故障。如果整个群集出现故障，则所有 region server 都将进行标准恢复。

区域的事务状态处于内存中，并且不会在重新平衡操作的区域划分中持续存在。当前，DTM 将延迟该操作，直到发现该区域没有任何事务运行，然后再启动该操作。但是，对于 TM 将延迟拆分时间有一个阈值。如果超过此阈值，它将禁用新事务，等待活动事务耗尽，进行拆分并重新启用该区域的事务。在将来的版本中，事务状态将被复制到新区域。在重新平衡时，负载平衡器将指定应在同一 region server 上并置哪些区域。

10. 竞争性角度

HBase 还有其他事务管理实现。分布式事务管理与以下其他开源实现之间有许多区别：

- 尽管其他实现是使用 Java 的，~~但~~当前大多数 DTM 是使用 Java 的，但是部分事务管理器仍然是 C ++。将其转换为 Java 的工作非常有限，并且任何其他希望利用 DTM 的客户端都将提供足够的动力来投入这项工作。
- 虽然 DTM 当前支持 READ COMMITTED 隔离，而且快照隔离已纳入产品线路图，其他实现也已经支持快照隔离
- DTM 具有完全分布式的体系结构，而其他实现则具有针对整个集群的集中式单个事务管理器或事务状态 Oracle（TSO）。这意味着该单一过程必须：
  - 协调整个集群中的每个读取或写入事务
  - 必须维护集群上运行或最近提交的每个事务的状态、读/写
  - 事务性元数据必须复制到每个客户端
  - 通过消息与每个 region server 来回通信
  - 管理事务之间的所有冲突
  - 管理事务的整个生命周期

  这意味着这个体系结构会：
  - 无法跨非常大的集群扩展非常多的并发事务
  - 无法处理长时间运行的事务，并且仅适用于短时间运行的事务
  - 尚不清楚单个节点或是单个进程失败对整个区域集群中的单个节点的所

有消息传递和事务协调的影响，对所有运行事务有什么影响，事务型负载需要多久才能恢复。尚不清楚如何处理单个节点处于繁忙状态的偏斜。这个影响可能会非常大。

- 目前尚不清楚在节点或进程故障影响事务管理器的情况下，整个集群将受到什么影响。这对所有正在运行的事务有什么影响？需要多长时间才能恢复事务工作负载？这对高可用性的影响可能会非常大。

- DTM 允许多个进程加入一个事务。因此，如果有多个进程代表同一个事务跨集群协调读/写，则为了提供横向扩展的高并行度，DTM 透明地促进了这一过程。其他实现不提供此功能，并且使用该体系结构的实现在横向扩展查询内/事务内并行性方面将受到限制。
- DTM 利用 HBase 协处理器来利用最高效和集成的事务实现来进行事务管理和恢复。日志与 HBase WAL 集成在一起，而 TLOG 是 HBase 表。其他实现未在同一级别与 HBase 集成。因此，尚不清楚事务的性能开销是多少。还不清楚这些实现如何处理恢复。在各种故障情况之后，它是否与 HBase 自动恢复集成在一起？在故障发生后可以恢复事务工作负载的恢复速度有多快。
- 虽然其他事务管理器在客户端管理事务，但在 DTM 中，客户端仅管理事务的某些方面，仅在需要时才产生进程间消息。服务器端（HBase 协处理器）管理事务的数据方面。 DTM 还管理易鲸捷数据库的元数据表的事务更新，并且在将列删除或添加到参与活动事务的表中时能够检测服务器端的冲突。客户端 TM 需要存储引擎中的功能来执行此操作。上面介绍了客

户端 TM 的其他缺点：

- 很多网络消息
- 有限的可扩展性
- 无法支持参与同一事务的多个客户

尽管从某种意义上说效率是可伸缩性和可用性的一部分，但它是性能的一个方面，对于任何分布式事务管理系统都是必不可少的。如果存在过多的消息传递，或者单个节点可能成为性能瓶颈，那么这将增加系统的开销，从而缩短客户应用程序的周期。我们已经尝试使用“订单输入”基准（TPC-C 等基准）来描述这种开销。我们鼓励其他事务管理实现也这样做，以便更完整地体现其功能。

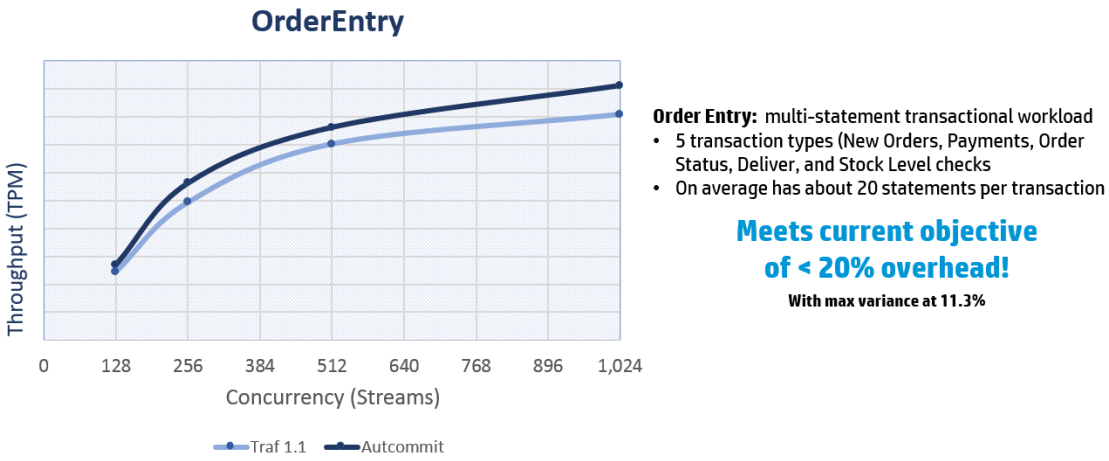

图 12. 交易费用符合当前目标

11. 总结

希望本文能使您对分布式事务管理体系结构有很好的了解，以及如何将其用于 DTM 提供的各种功能，例如完整的 ACID 事务支持和故障时的恢复。 希望它还证明了该体系结构具有很好的可扩展性，并致力于提供最高级别的可用性和一致性，并提供快速有效的恢复。与任何新产品部署一样，仍然存在一些提高效率的空间。但是基础是坚实的，这是建立新功能的基础。

了解更多信息:

www.esgyn.com。