通过应用侧事务合并提升数据库性能

Xueyuan Ren俄亥俄州立大学美国俄亥俄州哥伦布 ren.450@osu.edu

Frank Li俄亥俄州立大学美国俄亥俄州哥伦布 li.12634@buckeyemail.osu.edu

Yang Wang俄亥俄州立大学美国俄亥俄州哥伦布 wang.7564@osu.edu

arXiv:2601.10596v1 [cs.DB] 15 Jan 2026

摘要—**本文探讨了一种提升应用程序端事务处理性能的新方法：合并结构相似的语句或事务。具体而言，我们改写事务，以便：1）利用特定的 SQL 语义合并相似语句；2）消除冗余读取；3）通过预先计算聚合效果，合并不同事务中相互竞争的语句。基于这一思路，我们介绍了 TransactionMerger 的设计。它是一种中间件，可收集并合并来自不同客户端的事务。我们还介绍了一种静态分析工具，用于在不违反隔离性的前提下识别合并机会，并分享了我们在 TPC-C 和 Spree（一款广受欢迎的真实应用程序）中改写事务的经验。评估结果表明，这种事务合并可使 TPC-C 吞吐量最高提升 2.65 倍，使 Spree 吞吐量提升 3.52 倍。**

索引词—**事务处理、中间件、数据库应用程序**

I. 引言

我们的社区一直在从多个维度提升数据库系统的性能，包括并发控制 [1], [2], 查询规划 [3], 复制 [4],等。然而，学术界提出的技术往往需要很长且难以预料的时间才能应用于成熟的数据库系统，这可能是因为这些系统十分复杂。同样，Oracle 和 SQL Server 等商业数据库系统可能提供开源数据库中不存在的优化。

对于迫切需要提升性能、却无法更改数据库的用户，另一种选择是在应用程序端优化性能，即重写或重构其 SQL 事务。示例包括事务切分 [5],语句重新排序 [6], 事务共享 [7], 等。与数据库侧优化相比，应用侧优化可能需要数据库应用程序开发者付出额外努力，但可以在可预测的时间内完成部署。此外，应用侧优化能够利用特定于应用程序的知识，而这些知识通常很难在数据库端获取。先前的研究表明，此类应用侧优化在现实世界的应用程序中相当常见 [8], [9]。

本文探讨了一种新的应用侧优化机会：合并结构相似的语句或事务。与简单的客户端批处理方法相比，该方法将少量插入、更新或删除语句一并执行 [10], 我们的方法侵入性更强，需要修改应用程序代码，但能带来更大的性能提升。为将我们的方法与简单方法区分开来，本文称之为“事务合并”。具体而言，本文探讨了两种优化机会：

事务内合并。一个事务可以包含多条插入、更新或删除语句，以更改多行的状态；也可以包含多条查询语句，以检查多个条件。由于这些语句之间通常不存在依赖关系，因此为合并提供了天然的机会。我们发现，SQL 已经提供了支持合并的语义：我们可以串联多个查询或删除语句的条件，一并执行它们的任务；插入支持一次插入多行；还可以使用 UPDATECASEWHEN 语句，根据不同条件更新多行。

我们的微基准测试评估表明，这种事务内合并可使 MySQL 上的事务处理吞吐量最高提升 6.0 倍，在 PostgreSQL 上最高提升 4.9 倍。

跨事务合并。数据库应用程序通常会向最终用户提供多个 API，每个 API 执行一个或少数几个事务。TPC-C [11] 中的新订单和付款事务就是这样的例子。因此，如果多个用户同时访问应用程序，他们可能会通过同一个 API 发起事务；这些事务的结构相似，但参数值不同。这类事务也为合并提供了机会。例如，我们可以利用上文讨论的 SQL 语义，实现一个事务来处理来自多个客户的多个新订单或付款请求。为实现这一构想，我们构建了 TransactionMerger，这是一种中间件，可以从客户端累积相似事务，将它们合并为一个事务，执行合并后的事务，并将结果分发给客户端。

与事务内合并相比，跨事务合并至少带来了另外两个优化机会。首先，许多客户端的事务会读取相同的数据。例如，TPC-C 中的 NEW-ORDER 事务需要读取本地仓库的税率，因此同一仓库的所有客户都会读取相同的值。因此，如果我们合并来自同一仓库的 NEW-ORDER，合并后的事务只需读取一次税率，从而消除冗余的

读取。其次，跨事务合并可以将通常是事务处理瓶颈的争用转化为优化机会。例如，如果两个事务都希望将某个数值减一，那么合并后的事务只需将该数值减二——这不仅消除了争用，还减少了数据库需要执行的操作数量。

然而，跨事务合并可能会打破事务边界，引发许多问题 [12]。例如，在基于锁的并发控制机制中，如果两个事务都想更新同一行，其中一个就会被延迟——这是实现可串行化等隔离级别所必需的。如果我们只是合并两个事务的两项更新，可能会产生在可串行化隔离级别下不会出现的结果。为了解决这个问题，我们构建了一款静态分析工具，帮助开发者在不破坏隔离性的前提下识别跨事务合并的机会。

本文的贡献如下：

- 我们设计并实现了 TransactionMerger，这是一种用于收集并合并来自多个客户端的事务的中间件。其设计遵循两项原则。首先，TransactionMerger 应当是无状态的，以简化容错、负载均衡和扩展。幸运的是，我们发现数据库事务的中止支持大大简化了这项任务，因为当事务执行到一半失败时，TransactionMerger 可以随时中止该事务，而不必尝试持久化其中间状态以恢复执行。其次，TransactionMerger 应支持在线重新配置，因为最优设置（例如要合并的事务数量、要合并哪些事务等）往往取决于工作负载，因此可能随时间变化。为避免代价高昂的同步配置更新，我们的设计确保 TransactionMerger 的正确性不会因不同组件之间的暂时不一致而受到破坏。

- 我们设计了一种静态分析工具，帮助开发者识别不会破坏隔离性的跨事务合并机会。基于此，我们介绍了为 TPC-C（一种标准数据库基准测试）和 Spree [13]，（一种流行的在线购物框架）实现合并事务的经验。我们的经验表明，首先，尽管这项工作需要付出相当大的努力，但开发者可以循序渐进地开展，例如从一部分性能关键型事务或语句入手。其次，除了静态分析方法所发现的机会外，开发者还可以利用应用程序信息进行进一步优化。这是应用程序端优化的一项优势，而在数据库端几乎无法实现这种优化。

- 我们的评估表明，事务合并无需对数据库后端进行任何更改，即可将 TPC-C 的吞吐量提升至原来的 2.65 倍，将 Spree 的吞吐量提升至原来的 3.52 倍。因此，我们认为 TransactionMerger 为提升数据库性能提供了一种有价值的替代方案，尤其是在开发者无法控制数据库后端时。

II. 优化机会

A. 事务内合并

一个事务包含多个彼此独立的插入、更新、删除或查询语句的情况并不少见，这为批处理创造了天然的机会（示例见第 IV 节）。

实际上，如今的客户端 SQL 连接器（例如 JDBC）已经提供了 ADDBATCH 和 EXECUTEBATCH 函数，以支持批处理。当应用程序调用 ADDBATCH 时，连接器会缓冲后续的 INSERT、UPDATE 和 DELETE 语句（请注意，JDBC 批处理不支持 SELECT）。随后，当应用程序调用 EXECUTEBATCH 时，连接器会将所有缓冲的语句一并发送到数据库。稍后将会说明，这种方法在某些数据库实现中可以提升性能，但我们的方法能够带来更大的提升。

我们的方法使用下文讨论的特定 SQL 语义，将这些事务重写为一个事务。我们将重写前的语句称为原始语句，将重写后的语句称为合并语句。具体来说，

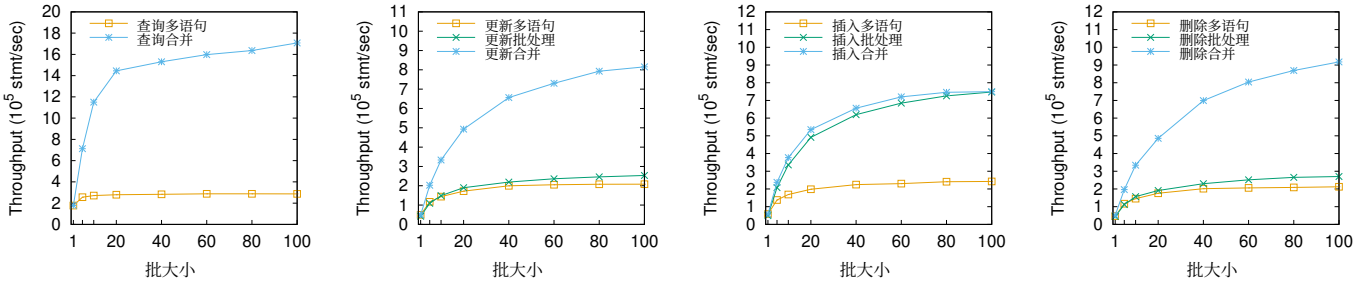
查询和删除。对于这些语句，我们可以通过两种方式合并。首先，可以使用 OR 关键字合并它们的 WHERE 子句。其次，如果它们的 where 子句仅包含等值关系，那么可以使用 IN 关键字合并等值关系所比较的值。我们的评估表明，在可以使用 IN 时，其性能优于 OR，因此我们会尽可能使用 IN。

对于查询，我们还需要扫描合并语句的结果集，查看哪些行符合哪个 WHERE 子句，并相应地分发这些行。由于结果集通常不大，因此额外开销很低。在更复杂的场景中，如果结果无法轻易分离（例如 SELECT SUM），我们发现通常可以使用 GROUP BY 关键字自动分离结果。

更新。我们使用 UPDATE CASEWHEN 语句更新多行。CASE-WHEN-THEN-ELSE 块类似于 IF-THEN-ELSE，会评估条件并返回相应的值。我们将 WHERE 子句中的所有条件放入多个 WHEN 块中，每个 THEN 块中的返回值正是对应条件下要更新的值。我们将 ELSE 块中的值设为原始值，以防没有任何条件满足。

插入。带有 VALUES 关键字的 INSERT 语句可以通过逗号分隔来插入多行。如果有多个可以合并的 INSERT 语句，我们会将它们串接成一条语句，并附加所有值列表。

为衡量事务内合并的潜在收益，我们进行了微基准测试评估。测试创建了一个包含一百万行的表，用于查询/更新；另创建了一个包含一千五百万行的表，用于删除（由于删除操作会在实验过程中减少行数，因此删除操作需要更多行）。每行有两列：标识符和值。在原始版本中，每个事务执行的语句批大小为



(a) SELECT-MySQL (b) UPDATE-MySQL (c) INSERT-MySQL (d) DELETE-MySQL

图1：MySQL 上微基准测试的吞吐量。

条相同类型的语句（即查询、插入、更新或删除），每条语句使用主键（标识符）随机操作一行。各语句如下：1. SELECT VALUE FROM TABLE WHERE ID= ?; 2. INSERT INTO TABLE (ID, VALUE) VALUES (?, ?); 3. UPDATE TABLE SET VALUE = ? WHERE ID = ?; 4. DELETE FROM TABLE WHERE ID = ?。我们以每秒执行的语句数衡量原始版本的吞吐量。对于插入、更新或删除操作，我们还测量了使用 JDBC 批处理时的吞吐量。

然后，我们的方法使用上述方法，将每个事务中的所有语句合并为一条合并语句。为了与原始版本进行公平比较，我们将合并版本的吞吐量报告为每秒事务数乘以每个事务中待合并的语句数。这些实验的硬件和软件配置见第 V 节。

图1比较了 MySQL 上原始版本和合并版本的吞吐量。如图所示，MySQL 可以像我们的合并方法一样，将多个 INSERT 语句合并为一条，但对 UPDATE 和 DELETE 几乎没有进行优化。我们的合并版本在查询、更新和删除方面均优于 MySQL，性能最高提升 6.0 倍。我们还在 PostgreSQL 上进行了类似测量，发现我们的版本在查询和更新工作负载上均优于 PostgreSQL，速度分别最高提升 4.9 倍和 1.4 倍。

B. 跨事务合并

数据库客户端与数据库服务器之间的中间件有可能合并来自不同客户端的同类型事务。这种方法既带来了好处，也面临着挑战。

从优势来看，首先，跨事务合并带来了更多合并机会。单个事务中可供合并的语句数量有限，但如果考虑来自不同客户端的同类型事务，可合并的语句数量就会显著增加。

此外，当不同事务发生竞争时，也就是说它们访问相同的行，竞争语句可以比非竞争语句更有效地合并。例如，如果来自两位客户的两个事务都想购买同一件商品，也就是说它们都需要将商品数量减一，那么合并后的版本只需将商品数量减二。这样一来，两个更新就可以合并为一个——这甚至优于合并非竞争事务，因为后者仍需更新两行。通过这种方法，我们可以将竞争事务（这是数据库性能面临的典型挑战）转化为优化机会，其收益甚至超过非竞争事务。

最后，有些事务可能会执行相同的检查（例如，商品是否有货）或读取相同的值（例如，日期或税务信息）。合并后，合并版本只需执行一次这些操作。

这些场景的潜在性能提升可以从事务内合并的结果中推断出来（图1）。例如，假设每个原始事务都更新单行，那么其吞吐量将等于图中原始线或批处理线上的（批大小=1）点；假设这些事务之间不存在争用，且我们可以合并其中的 X 个，那么合并版本的吞吐量将等于图中合并线上的（批大小=X）点；假设这些事务之间存在争用，且我们可以合并其中的 X 个，那么吞吐量将提升 X 倍。

然而，实现跨事务合并比实现事务内合并更具挑战性，原因有几个。首先，事务是数据库系统中保证原子性和隔离性的基本单位。与事务内合并不同，跨事务合并打破了事务之间的边界，从而对隔离性造成影响。

为了解决这些问题，我们构建了一个静态分析工具，帮助开发者在不破坏隔离性的情况下找到合并机会（第 IV-A 节）。此外，我们的经验是，利用应用层知识和约束在应用层重写代码，可以带来更大的改进。例如，考虑这样一个事务：先将一个 ID 加一，然后使用新的 ID 创建一个订单，以确保不同订单拥有不同的 ID。“加一”语句是一个典型的竞争点，因为多个事务可能会尝试更新同一个值。然而，我们不能简单地将多个“加一”合并为一个“加 n”，因为这样多个事务

就会使用相同的 ID 创建订单。这个问题是由合并破坏了事务的隔离边界所导致的：在原始事务中，一个事务只能看到自身执行的“加一”，而看不到其他事务执行的操作；在合并后的版本中，事务则可以看到所有其他事务执行的加一操作。另一方面，在应用层重写代码相当直接。合并后的事务可以将旧 ID 值读入一个局部变量（old id），在数据库中将其加 n，然后使用 old id+1、old id+2 等创建每个订单。不过，要自动确定如何执行这一操作，可能需要复杂的分析。

第二个挑战是如何确定要合并哪些事务。我们的观察是，合并竞争事务更好，因为如上文所述，这样做不仅能带来更大的改进，还能让不同事务并行执行。从理论上讲，利用这一想法并不容易，因为不同事务可能以不同方式在多个位置发生竞争，而这些信息在事务执行之前可能并不可用。然而，我们在本文中使用的应用程序包含易于识别的合并点，例如 TPC-C 中的仓库区域。这可能是因为，在分布式数据库中划分数据时，数据库设计者一开始就会尽量减少跨分区事务。因此，我们可以自然地尝试合并属于同一分区的事务。不过，在必要时，也可以采用更复杂的分析来确定要合并哪些事务（参见第六节）。

III. 设计与实现

图2展示了 TransactionMerger 的设计。如图所示，TransactionMerger 是位于数据库客户端（例如 Web服务器）与数据库服务器之间的典型中间件。开发者需要将合并版本的代码提交给 TransactionMerger。数据库客户端会向中间件中的合并器发送 RPC（例如 PUR-CHASE(ITEM=BOOK1,ITEM COUNT=3)）。每个合并器会收集一定数量的 RPC，将其合并，并执行开发者提交的合并版本；这一过程包括向数据库服务器发送合并后的 SQL语句，并将结果分发回客户端。每个合并器内部都可以拥有多个线程，以最大限度地提升性能。

如第二节B所述，最好合并具有某些属性的事务。为了最大限度地提高合并的可能性，TransactionMerger 在不同位置引入分区器，以确保可能合并的事务能够进入同一线程。例如，在 TPC-C 中，将发往同一仓库区域的事务进行合并是一个自然的思路。随后，客户端分区器可以采用如下策略：发往仓库1至10的事务交由合并器1处理，发往仓库11至20的事务交由合并器2处理，依此类推。类似地，合并器1的分区器可以采用如下策略：发往仓库1至2的事务进入线程1，发往仓库3至4的事务进入线程2，依此类推。最后，每个线程的分区器根据事务所属的仓库区域对事务进行分区。

或者，客户端也可以直接向服务器发送 SQL 事务。这样，开发者就可以在 TransactionMerger 中逐步合并事务的一个子集。

TransactionMerger 的设计遵循两项原则，以实现容错、可扩展性和高性能。

保持合并器无状态。如果服务无需保留任何持久化数据便能正常运行，那么它就是无状态的。无状态服务在简化设计方面有诸多好处：如果某个服务实例发生故障，我们只需创建一个新的空白实例，无需承担复制等成本；客户端可以连接任意服务实例，从而简化负载均衡，因为我们可以在不同服务实例之间自由迁移客户端；如果服务流量增加，则可以添加空白实例来扩展服务。

为了保持合并器无状态，TransactionMerger 仍依赖数据库后端来实现原子性、隔离性、持久性等。因此，合并器无需维护重做日志、并发控制等机制所需的数据，这大大简化了其设计。如果合并器因任何原因发生故障，TransactionMerger 无需尝试恢复尚未完成的事务（这需要合并器复制其中间状态），而是可以始终中止这些未完成的事务，并要求客户端重试。当然，中止和重试会带来额外开销，但由于合并器故障应该很少发生，与复制中间状态的成本相比，这类开销可以忽略不计。

最具挑战性的情况是，合并器成功完成合并后的事务，却在将结果分发给客户端之前发生故障。然而，我们发现，这种情况与以下情形并无不同：客户端向数据库服务器发送一个直接事务，服务器完成该事务，但客户端由于网络问题或自身故障而未能接收到结果。换言之，无论客户端应用程序是否使用 TransactionMerger，都应该已经具备用于处理这种情况的逻辑。在实践中，我们观察到一种常见的解决方案是，为每一行维护一个标志或时间戳，以标记该行是否已被处理，或标记处理过该行的事务。

支持在线重新配置。 TransactionMerger 可能出于多种原因需要重新配置。首先，如上文所述，如果我们出于容错或可扩展性方面的原因添加或移除合并器，则需要更新不同分区器的策略。其次，事务合并的性能取决于多个参数，例如线程数、要合并的事务数以及合并策略（即要合并哪些事务）。最佳设置可能取决于流量模式。由于流量模式可能随时间发生变化，因此，如果 TransactionMerger 能够在运行时更改这些设置，将会更理想。

然而，直接更改此类设置可能会引发微妙的同步问题。沿用前面的 TPC-C 示例，假设合并器1最初负责仓库1-10，之后由于负载均衡，我们需要将策略改为把仓库1-20的事务发送到合并器1。在

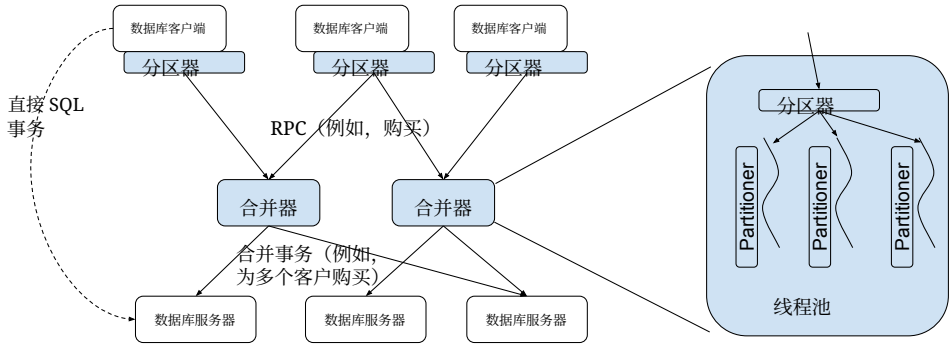


图 2：TransactionMerger 的设计

这种情况下，客户端分区器和合并器端的分区器都需要更新各自的策略。如果处理不慎，不同分区器之间的不一致可能会造成麻烦。一种解决方案是实现多个分区器的原子更新，但这很复杂，而且可能带来显著开销。

为支持在线重新配置，我们的设计原则是，系统的正确性不应取决于不同分区器之间的一致性。换言之，每个分区器都应能够处理任何事务，而不是假设由于上层分区器的策略，它只需处理其中一部分。以之前的 TPC-C 示例为例，遵循这一原则，线程端分区器应能够划分来自任何仓库的事务；合并器端分区器则应为它本不应处理的事务准备备用方案（例如，来自规模大于 10 的仓库的事务应交由线程 X 处理）。采用这种策略，系统的正确性便不再取决于分区策略。因此，不同的分区器可以异步更新；分区器之间暂时的不一致可能会损害性能，但不会影响正确性。

借助在线重新配置支持，我们实现了一种贝叶斯优化（BO）方法，用于在运行时调整线程数量以及要合并的事务数量。这种方法广泛用于系统配置调优 [14]–[17]。具体而言，我们采用高斯过程回归器（GPR）来建模配置与其对应吞吐量之间的关系。从一组初始随机配置开始，我们测量系统吞吐量并训练高斯过程回归模型。随后，贝叶斯优化会迭代选择新的最优配置，并利用新配置重新训练高斯过程回归模型。这个循环会持续进行，直到吞吐量提升趋于稳定或达到预定义的迭代次数上限，使 TransactionMerger 能够自主、高效地收敛到高性能配置。

实现。我们使用 gRPC 框架实现合并器 [18]，将每种事务类型都公开为一个 RPC 方法。合并器还包含若干用于合并事务的工作线程，每个线程都关联一个输入队列。当 gRPC 线程收到 RPC 请求时，会使用合并器端分区器确定要将该请求发送给哪个工作线程，然后将请求加入该工作线程对应的队列。每个工作线程都会反复从自己的队列中提取请求，使用线程端分区器对请求进行分区，合并同一分区中的事务，并将合并后的版本发送到数据库。工作线程的数量以及要合并的事务数量可在运行时通过另一个 RPC 方法进行配置，以便从客户端调整这些参数。当线程已合并足够数量的事务或发生超时时，就会停止合并。

我们利用 scikit-learn 的 GaussianProcessRegressor，在 Python 中实现贝叶斯优化方法，并采用径向基函数 +白噪声核核 [19]，，其中径向基函数用于捕捉非线性关系，白噪声核用于处理噪声。通过 scikit-optimize 的 gp minimize [20]，贝叶斯优化在 GPR 预测的引导下探索离散参数空间。我们还使用 gRPC 服务器，将重新配置部分与 TransactionMerger 集成。

IV. 案例研究

本节首先介绍寻找合并机会的方法，然后分享我们在两个应用程序 TPC-C 和 Spree 中实现事务内合并与跨事务合并的经验。TPC-C 是广泛用于评估在线事务处理（OLTP）系统的基准测试，而 Spree 是一款开源在线购物应用程序。我们先介绍这两个应用程序中的原始事务，然后说明如何重写这些事务以实现事务内合并与跨事务合并。

A. 通过静态分析发现合并机会

这项工作的核心思想是合并同一事务多个实例中结构相似的语句。然而，这可能会导致这些实例之间的隔离性遭到破坏。直观地说，给定同一事务的两个实例及其语句 $\{S_1^1, S_2^1, ..., S_n^1\}$ 和 $\{S_1^2, S_2^2, ..., S_n^2\}$，我们想知道它们的语句 $\{S_1^1, S_2^1, ...S_n^1, S_1^2, S_2^2, ...S_n^2\}$ 串行执行是否等价于重新排序后的执行 $\{S_1^1, S_1^2, S_2^1, S_2^2, ..., S_n^1, S_n^2\}$：如果等价，我们就可以采用重新排序后的执行，并创建合并执行 $\{S_1^1 + S_1^2, S_2^1 + S_2^2, ..., S_n^1 + S_n^2\}$（“+”表示合并）。

然而，答案并非总是肯定的。假设 S_2^1 更新了 S_1^2读取的某个值，那么上述重新排序会导致 S_1^2 获取到的值与串行执行中的不同，从而破坏隔离性。

为了系统地识别合并机会，同时不违反隔离性，我们的核心思路是将一个事务的语句划分为多个组，使冲突（即访问同一行，且至少有一次访问是写入）只存在于组内，而不存在于组间。这样，我们就可以像上文所述那样重新排序各组，而不违反隔离性。在上面的例子中，假设 S_1 和 S_2 之间存在冲突，而其他语句之间不存在任何冲突，我们会将事务划分为 $\{G_1, S_3, ..., S_n\}$，其中 G_1 包含 S_1和 S_2。然后，对于同一事务的两个实例，我们会将其执行重新排序为 $\{G_1^1, G_1^2, S_3^1, S_3^2, ..., S_n^1, S_n^2\}$。利用序列化图理论，可以轻松证明这种重新排序的正确性：在与重新排序后的执行相对应的依赖图中，冲突边只能从 T_1 指向 T_2，因为一个组只会与自身发生冲突，而且我们总是将 T_1的组放在同一位置的 T_2的组之前。

为识别这类组，我们的算法首先为每条 Statement 创建一个组，然后合并各组：如果两个组之间存在冲突或重叠，我们就合并这两个组以及它们之间的所有组。我们重复这一过程，直到无法再进行合并。

在重新排序后的执行中，如果一个组只包含一条 Statement，我们就可以使用本文讨论的技术合并对应的组。否则，我们要么需要顺序执行对应的组，要么必须重写应用程序逻辑，以便实现合并。在上面的示例中，这一步会生成执行序列 $\{G_1^1, G_1^2, S_3^1 + S_3^2, ..., S_n^1 + S_n^2\}$。

目前的实现依赖开发者提供 Statement 信息，包括该 Statement 读取或写入哪些列。然后，我们的算法根据以下标准检测冲突：两条 Statement 访问同一列，且至少有一条执行写入。这里有几个例外：首先，如果一条 Statement 插入新行，而另一条查询现有行，我们不会将它们视为冲突。其次，如果开发者确定两个事务在某些 Statement 中使用不同的主键作为条件，那么即使这些 Statement 访问同一列，也不会将它们归入同一组，因为它们不会在事务之间产生任何依赖关系。

B. TPC-C

TPC-C 对批发供应商环境进行建模，并定义了五种事务类型，每种类型都通过特定的 SQL 语句模拟现实世界中的业务操作。我们基于开源工具 Benchbase [21] 实现了合并事务，该工具实现了 TPC-C。以下事务属于 Benchbase 的 TPC-C 工作负载。

新建-订单事务模拟客户在系统中下达新订单的过程，其中包括检索客户数据，获取并更新与区域订单编号相关的字段，为新订单和订单表插入行；然后，对于订单中的每件商品，获取商品和库存详情、更新库存信息，并插入相应的订单项记录。

我们将发往同一仓库区域的新订单事务进行合并。如图 3 所示，原始版本以一个客户作为输入，而合并版本可以接受多个客户作为输入。我们的静态分析器将第 3–4 行归为一组，将第 8–10 行归为一组；我们的研究表明，通过改写应用程序逻辑，仍可在这两组中执行合并。接下来，我们逐行说明如何进行合并及其潜在收益：

- 原始版本的第 1 行会检索一个客户的折扣和信用信息。合并版本的第 1 行则使用第 II-A 节讨论的 SELECTIN 语义检索多个客户的此类信息。
- 原始版本第 2 行检索税务信息，以供后续处理。由于所有客户的税务信息都相同，合并版本只需执行一次。
- 第 3–4 行读取下一个订单 ID（next o id），然后将其加一。这两行显然存在冲突，因此我们的静态分析器会将它们归为一组。然而，我们发现，无需顺序执行这些分组，而是可以改写其逻辑，以创造更多合并机会。具体而言，合并版本的第 3 行只读取一次 next o id，但会创建一个数组 [next o id, next o id+1, ..., next o id+n]（在第 5–6 行中使用），以模拟顺序执行期间会收到的响应；合并版本的第 4 行则将 next o id 增加客户数量。这样的改写所产生的执行结果等同于串行执行，但效率更高。在原始版本中，第 4 行是一个竞争点，因为所有客户都需要更新这个值。然而，在合并版本中，第 4 行则成为一项优化机会，因为争用的更新可以在由数据库执行之前进行聚合，正如第二节B所述。这是一个例子，说明某些修改在应用程序端相当直接，却很难在数据库端自动完成。
- 原始版本的第 5-6 行插入一个未结订单（oorder）和该客户的一个新订单。合并版本的第 5-6 行使用合并 INSERT 语句插入多个订单，每个客户对应一个订单。请注意，这两行使用了第 3-4 行创建的模拟下一个订单标识符数组。
- 原始版本的第 7-10 行执行循环，为每件商品插入一个订单项，并相应地更新商品数量。合并版本的第 7-10 行使用合并语义替代循环：第 7-8 行使用 SELECT IN 语义获取多个商品的信息；第 9 行使用合并 INSERT 语句将多个商品插入订单项；第 10 行使用 UPDATE CASE WHEN 语句更新多个商品的信息。

```
输入：仓库编号 (w_id)、区域编号 (d_id)、客户编号 (c_id)、商品编
号数组 (item_ids)、供应商 w_id 数组 (supplier_w_ids)、数量数组
(quantities) -- 获取客户数据 1 SELECT c_discount,... FROM
customer WHERE c_w_id = ?AND c_d_id = ? AND c_id = ?; --
获取仓库数据 2 SELECT w_tax FROM warehouse WHERE w_id =?;
-- 获取区域的 d_next_o_id (@next_id)，并锁定以便更新 3 S
ELECT d_next_o_id AS next_id, d_tax FROM district WHERE d_
w_id = ?AND d_id = ? FOR UPDATE; -- 将 d_next_o_id 加一 4
UPDATE district SET d_next_o_id = d_next_o_id + 1
WHERE d_w_id = ?AND d_id = ?; -- 插入订单和 new_order 记
录 5 INSERT INTO oorder (o_id, o_w_id, o_d_id,...) VALUES
(@next_id,?,?,...); 6 INSERT INTO new_order (no_o_id, no_w_i
d, no_d_id) VALUES (@next_id,?,?); -- 对每个订单项执行 order_
lineitem：对每个订单项执行：-- 获取商品数据 7 SELECT i_price, i_
name, i_data FROM item WHERE i_id = ?; -- 获取库存数据 8
SELECT s_quantity,... FROM stock WHERE s_w_id = ? AND s_i_
id = ?FOR UPDATE; -- 插入订单项记录 9 INSERT INTO order li
ne_(ol_w_id, ol_d_id, ol_o_id,...) VALUES (?,?,?,...); -- 更新库存
10 UPDATE stock SET s_quantity = ?, s_ytd = s_ytd +?, s_
order_cnt = s_order_cnt + 1, s remote cnt = sremote cnt + ?
_ _ _ _WHERE s_w_id = ?AND s_i_id = ?;
```

（a）原始版本。

```
输入：w_id、d_id、客户编号数组（c_ids）、商品_ids、supplier_w_
ids、数量 -- 获取所有客户数据 1 SELECT c_id, c_discount,... FROM
customer WHERE c_w_id = ? AND c_d_id = ?AND c_id IN (?,...); --
获取所有客户的仓库数据 2 SELECT w_tax FROM warehouse WHERE
w_id = ?; -- 获取区域的 d_next_o_id（@next_id），并锁定以供更新
3 SELECT d_next_o_id AS next_id, d_tax FROM district WHERE d_
w_id = ? AND d_id = ? FOR UPDATE; -- 将 d_next_o_id 增加客户数
量（@n） 4 UPDATE district SET d_next_o_id = d_next_o_id +
@n WHERE d_w_id = ? AND d_id = ?; -- 插入所有订单和 new_
order 记录 5 INSERT INTO oorder (o_id, o_w_id, o_d_id,...) VALUES
(@next_id,?,?,...),(@next_id+1,?,?,...),...; 6 INSERT INTO new_order (
no_o_id, no_w_id, no_d_id) VALUES (@next_id,?,?),(@next_id+1,?,?),...;
-- 对所有订单明细项一次性执行 -- 获取所有商品数据 7 SELECT i_id, i_
price, i_name, i_data FROM item WHERE i_id IN (?,...); -- 获取所有库存
数据 8 SELECT s_w_id, s_i_id, s_quantity,... FROM stock WHERE (s_
w_id, s_i_id) IN ((?,?),...) FOR UPDATE; -- 插入所有订单明细记录 9
INSERT INTO order_line (ol_w_id, ol_d_id, ol_o_id,...) VALUES
(?,?,?,...),...; -- 更新所有库存 10 UPDATE stock SET s_quantity =CASE
WHEN s_w_id =? AND s_i_id = ? THEN ? ... ELSE s_quantity END, s_
ytd = CASE WHEN s_w_id = ? AND s_i_id = ? THEN s_ytd + ? ...
ELSE s_ytd END, s order cnt = CASE WHEN s w id = ? AND s i id = ?
_ _ _ _ _ _THEN s_order_
cnt + 1... ELSE s_order_cnt END, s_remote_cnt =CASE WHEN s_w_
id =? AND s_i_id = ? THEN s_remote_cnt +? ... ELSE s_remote_cnt
END WHERE (s_w_id, s_i_id) IN ((?,?),...);
```

（b）合并版本。

图 3：TPC-C新订单事务。

我们的静态分析器报告了一个包含第 8 至 10 行的组，因为第 8 行和第 10 行都会访问库存数量。不过，我们发现重新排序是可行的：在原始版本中，每个事务都会读取当前库存数量并将其增加一定数量；在合并版本中，合并后的事务会读取当前库存数量，并将其增加汇总后的数量。这两个版本会产生等价的执行。请注意，这是原始版本中的另一个竞争点。

在合并版本中，第 1 至 6 行适用于事务间合并，而第 7 至 10 行则适用于事务间合并和事务内合并。

付款事务处理订单付款。它会确定处理付款的具体仓库和区域，以及付款客户的信息。随后，它会更新相关仓库和区域的销售统计，以计入这笔付款。此外，它还会更新与客户相关的字段，以反映这笔付款。最后，它会将该付款事务的历史记录插入历史表。

图 4 展示了我们如何将同一仓库区域的付款事务合并。我们的静态分析器报告了以下行

6-8 行被视为一组，因为它们涉及“if”语句，因此无法自动拆分；不过，我们再次找到方法重新编写这些语句，以创造更多合并机会。

- 原始版本的第 1 行和第 3 行会将仓库和区域的年初至今余额增加订单中的值。合并版本的第 1 行和第 3 行会将该余额增加合并版本处理的所有订单的值总额。由于针对同一仓库和区域的事务需要更新这些相同的值，因此它们在原始版本中是争用点，而在合并版本中再次被转化为优化点。

- 原始版本的第 2 行和第 4 行会检索仓库和区域信息。合并版本的第 2 行和第 4 行只执行一次相同的操作，因为不同订单将共享这些信息。

- 原始版本的第 5 行会检索客户信息。合并版本的第 5 行使用 SELECT IN 语句检索多个客户的信息。

- 原始版本的第 6-8 行以不同方式处理两类客户。为了实现合并，合并后的版本首先将客户分成两组，每组对应一类客户（第.....行之前的伪代码

```
输入：w_标识符，d_标识符，c_标识符，
    客户付款金额（付款_金额），
    客户仓库编号（c_w_标识符），
    客户地区编号（c_d_标识符）
-- 使用付款金额更新仓库 w_ytd
1 更新 仓库 SET w_ytd =w_ytd + ?
    <style 标识符='1'>WHERE w_标识符 = ?;
-- 获取仓库数据
2 查询 w_街道_1,... FROM 仓库 WHERE w_标识符 = ?;
-- 使用付款金额更新区域 d_ytd
3 更新 区域 SET d_ytd = d_ytd + ?
    WHERE d_w_标识符 = ?AND d_标识符 = ?;
-- 获取区域数据
4 查询 d _街道 _1,... FROM 区域
    条件 d_w_标识符 = ?且 d_标识符 = ?;
-- 为简洁起见，我们仅展示按标识符获取客户的操作
5 查询 c _信用 ,... 来自 客户
    条件 c_w_标识符 = ?且 c_d_标识符 = ? 且 c_标识符 = ?;


-- 对于信用不良客户，获取 c_数据并更新
如果c 信用 == "BC":_
  -- 获取客户 c_数据并更新
  6 查询 c _数据来自 客户
      其中 c_w_标识符 = ? 并且 c_d_标识符 = ? 并且 c_标识符 = ?;
  7 更新 客户 设置
      c_余额 = ? ,
      c_数据 = ?,...
      WHERE c_w_标识符 = ? AND c_d_标识符 = ? AND c_标识符 = ?;


-- 对信用良好的客户，更新客户余额
否则:
  8更新 客户 设置
      c_余额 = ? ,...
      其中 c_w_标识符 = ? 并且 c_d_标识符 = ?并且 c_标识符 = ?;

-- 插入历史记录
9插入 历史记录 (...) VALUES (...);
```

(a) 原始版本。

```
输入：w_id、d_id、c_ids、付款金额数组（payment_amounts）、c_w_id 数组（c_w_ids）、c_d_id 数组（c_d_ids） -- 使用汇总付款总额更新仓库 w_ytd 1 更新 warehouse，设置 w_ytd 为 w =ytd _@total，其中 w + id _?; -- 获取仓库数据 2 查询 w = street_ 、 ...，来自 warehouse，其中 w_1id_? ; -- 使用汇总付款总额更新区域 d = ytd 3 更新 district，设置 d_ytd 为 d_ytd = @total，其中 d_w + id _? 且 d_id = ?; -- 获取区域数据 4 查询 d_street = 、...，来自 district，其中 d_w_1id _? 且 d_id = ?; -- 获取所有客户数据 5 查询 c_w = id、c_d_id、c_id、c_credit、...，来自 customer，其中 (c_w_id、c_d_id、c_id) 位于 ((?,?,?),...) 中; -- 将客户分成两组 对于每个客户 c：如果 c.c_credit _"BC"：bc_customers.add(c)；否则：gc == customers.add(c)；如果 bc_customers 不为空：-- 获取信用不良客户数据并更新 6 查询 c_w_id 、 c_d_id、c_id、c_data，来自 customer，其中 (c_w_id、c_d_id、c_id) 位于 ((?,?,?),...) 中; 7 更新 customer，设置 c balance_CASE WHEN c w id_? AND c d id_? =AND c =id = ? THEN ? ... ELSE c
_ _ _ _
balance END、c data _CASE WHEN c w id =? AND c d id _? = AND c = id = ? THEN ? ... ELSE c_ _ _ _ _
data END、...，其中 (c_w = id、c_d_id、c_id) 位于 ((?,?,?),...) 中；如果 gc_customers 不为空：-- 更新信用良好客户 8 更新 customer，设置 c balance_CASE WHEN c w id_? AND c d id_? =AND c =id = ? THEN ? ... ELSE c
_ _ _ _ balance END、...，其中 (c_w = id、c_d_id、c_id) 位于 ((?,?,?),...) 中; -- 插入所有历史记录 9 插入 history (...) 值 (...),...;
```

(b) 合并版本。

图 4：TPC-C 付款事务。

6）。然后，在每个组内，合并版本可以通过 SELECT IN 语句和 UPDATE CASE WHEN 语句进行合并。这是另一个需要小幅调整应用程序逻辑的例子。

- 原始版本的第 9 行向历史表中插入一条记录。合并版本的第 9 行通过合并 INSERT 语句插入多条记录。

我们也对配送事务进行合并，但由于其语句的合并模式与新订单和付款事务的合并模式没有太大区别，因此这里不再详述。其他两类事务——订单状态和库存水平——通常不会成为性能瓶颈，因此我们不对它们进行合并。

C. Spree

Spree Commerce [22] 是基于 Ruby on Rails 构建的开源电子商务平台。该平台提供了一套全面的功能，可用于构建在线商店，包括产品、订单和客户管理、购物车功能等。这个平台在现实中很受欢迎，全球有数千家企业在使用，其中包括 GoDaddy、Bookshop.com、肯德基、Mitchells 等品牌 [13]。

我们在自己的机器上部署了一个 Spree 网站，并模拟客户行为，以检查会触发哪些 Spree 事务。根据模拟结果，我们为两种事务实现了合并：新订单、添加商品和更新库存，因为这三种事务可能是使用频率最高的事务。由于新订单和更新库存相对简单，我们在此仅介绍添加商品。

添加-商品 事务会在 Spree 应用程序中向现有订单添加新商品。它会读取定价、商品详情、税务类别数据和可用库存；然后插入一条新的订单项记录；重新计算订单总额；最后更新订单的摘要字段。我们的静态分析器报告了一个涉及第 2 行和第 7 行的组，以及一个涉及第 10 至 11 行的组。我们发现，第一个组是由伪冲突导致的，因为第 2 行检查某一行是否存在，而第 7 行更新该行的某一列，因此它们实际上并不冲突。对于第二个组，我们发现对其中的语句重新排序没有问题。

- 原始版本的第 1-5 行会获取商品的价格、订单项详情（如存在）、税类详情、商品详情以及所有位置的商品库存。合并版本的第 1-5 行使用 SELECTIN Statement 为多个商品获取这些信息。请注意，对于第 5 行，由于 Statement 会计算总和，因此不容易拆分

输入：订单_标识符、变体_标识符、货币、税_类_标识符、数量 -- 获取商品 1 的价格 SELECT * FROM spree_prices WHERE deleted_at IS NULL AND currency = ? AND variant_id = ? LIMIT 1； -- 检查订单项是否存在 2 SELECT * FROM spree_line_items WHERE 订单_标识符 =? AND variant_id = ? ORDER BY created_at ASC LIMIT 1； -- 获取税类详情 3 SELECT * FROM spree_tax_categories WHERE deleted_at IS NULL AND id = ? LIMIT 1； -- 获取商品详情 4 SELECT * FROM spree_products WHERE id = ? LIMIT 1； -- 获取库存 5 SELECT SUM(count_on_hand) FROM spree_stock_items INNER JOIN spree_stock_locations ON spree_stock_locations.deleted_at IS NULL AND spree_stock_locations.id = spree_stock_it ems.stock_location_id WHERE spree_stock_ items.deleted_at IS NULL AND spree_stock_ locations.deleted_at IS NULL AND spree_stock_ locations.active = 1AND variant_id =?； -- 插入订单项记录 6 INSERT INTO spree_line_items (variant_i d, 订单_标识符, 数量, 价格, created_at, updated_at, 货币) VALUES (?,?,?,?,?,?,?,?)； -- 更新订单项记录 7 UPDATE spree_line_items SET pre_tax_amount = ? WHERE id = ?； -- 获取订单的总数量 8 SELECT SUM(数量) FROM spree_line_items WHERE 订单_标识符 =?； -- 获取订单的总价 9 SELECT SUM(价格 * 数量) FROM spree_line_items WHERE 订单_标识符 = ?； -- 更新订单记录 10 UPDATE spree_orders SET item_ total = ?, item_count = ?, 总额 = ?, updated_at = ? WHERE id = ? ； -- 确保 updated_at 时间戳已刷新 11 UPDATE spree_orders SET updated_at = ? WHERE id = ?； （a）原始版本。

输入：订单_标识符、变体_标识符、货币、税类_标识符、数量——获取所有商品的价格 1 查询 * FROM spree_prices WHERE deleted_at IS NULL AND currency = ? AND variant_idIN (?,...); ——检查订单项是否存在 2 查询 * FROM spree_line_ items WHERE (订单_标识符, variant_id) IN((?,?),...); ——获取税类详情 3 查询 * FROM spree_tax_categories WHERE deleted_at IS NULL AND idIN (?,...); ——获取商品详情 4 查询 * FROM spree_products WHERE id IN (?,...); —— 获取所有商品的库存 5 查询 variant_id, SUM(count_on_hand) FRO M spree_stock_items INNER JOIN spree_stock_locations ON spree_stock_locations.deleted_at IS NULL AND spree_ stock_locations.id = spree_stock_items.stock_location_id WHERE spree_stock_items.deleted_at IS NULL AND spree_ stock_locations.deleted_at IS NULL AND spree_stock_ locations.active = 1 AND variant_id IN (?,...) GROUP BY variant_id; ——插入所有订单项记录 6 INSERT INTO spree_ line_items (variant_id, 订单_id, 数量, 价格, created_at, updated_at, 货币) VALUES ((?,?,?,?,?,?,?,?),...); ——更新所有订单项记录 7 UPDATE spree_line_items SET pre_tax_a mount = CASE WHEN id =? THEN ? ... ELSE pre_tax_ amount END WHERE id IN (?,...); ——获取所有订单的总数量 8 查询 order_id, SUM(数量) FROM spree_line_items WHERE 订单_id IN (?,...) GROUP BY 订单_id; ——获取所有订单的总价 9 查询 订单_id, SUM(价格 * 数量) FROM spree_li ne_items WHERE 订单_id IN (?,...) GROUP BY 订单_id; —— 更新所有订单记录 10 UPDATE spree_orders SET 商品总额 = CASE WHEN id = ? THEN ? ..._ELSE 商品_总额 END, 商品数量 = CASE WHEN id = ? THEN ? ..._ELSE 商品_数量 END, 总额 = CASE WHEN id = ? THEN ?... ELSE 总额 END, updated_at = ? WHERE id IN (?,...); 11 UPDATE spree_ orders SET updated_at = ? WHERE id IN (?,...); （b）合并后的版本。

图 5：Spree 添加商品事务。

th[e sum] in th[e merge]d version. To address thi[s pro]bl em，合并版本应用 GROUP BY 子句，让数据库自动分离结果。

- 原始版本的第 6 行为一个商品插入一条记录。合并版本的第 6 行使用合并 INSERT 语句插入多条记录。
- 原始版本的第 7 行使用 UPDATE 语句设置商品的税前金额。合并版本的第 7 行则使用 UPDATE CASE WHEN 语句为多个商品设置该值。
- 原始版本的第 8 和第 9 行计算商品总数和订单总价。合并版本的第 8 和第 9 行则使用 SELECT IN 语句为多个订单计算这些值。同样，为了自动区分结果，它还应用了 GROUP BY 子句。
- 原始版本的第 10 行和第 11 行根据上述计算结果更新订单信息。第 10 行和

合并版本的第 11 行更新多个订单。请注意，第 10 行使用 UPDATE CASE WHEN，而第 11 行可以使用简单的 UPDATE，因为它将所有行都设为相同的值（即时间戳）。在静态分析中，这两行构成一组，因为它们都会更新“更新时间”列，但调整它们的顺序不会产生任何可见影响。

尽管 Spree 和 TPC-C 都针对在线购物场景，但两者在一些方面有所不同。首先，Spree 没有仓库或区域的概念，因此没有明显的方法来划分事务。我们的实现没有采用任何特定的分区器。其次，Spree 几乎没有事务内合并的机会，因为它的 ADD-ITEM 事务每次只添加一个商品。

D. 总结

如这些案例研究所示，在实践中，合并可以通过三种方式带来性能优势。首先，有时

多个事务会检索相同的信息，而合并后，合并事务只需检索一次此类信息。其次，有时多个事务会尝试更新同一个值，从而造成争用。通过在将这些更新应用到数据库之前先进行聚合，合并后的版本便可将竞争点转变为优化点。最后，也可以合并具有特定 SQL 语义的相似语句。

实现合并版本的工作量固然不小，但由于两个原因，仍然是可行的。首先，我们可以逐步实现合并，只针对部分事务，以及单个事务中的部分语句。这样，我们便能专注于性能关键事务和语句。我们在第 V 节测量这种渐进式合并对 TPC-C 的影响。其次，实现无需对应用程序进行高层次的重新设计，主要只需对各条语句进行局部调整，而这些局部调整往往遵循几种常见模式。因此，我们认为，应用侧事务合并是一种切实可行的性能提升方法，尤其是在数据库侧优化效果不足或无法使用时。

V. 评估

我们使用上述讨论的 TPC-C 和 Spree 工作负载评估 TransactionMerger。我们使用一台机器运行 MySQL 服务器，一台机器运行合并器，另有一台机器运行客户端。我们在 CloudLab [23],上使用 c220g5 机器进行实验。每台机器配备两颗 10 核英特尔至强银牌 4114 CPU（启用超线程后共 20 个虚拟核心）、192 GB 内存，以及一块 480 GB、6G SATA 固态硬盘 Intel DC S3500。各机器通过双端口 Intel X520-DA2 10Gb 网卡互连。我们运行 MySQL 服务器 8.0.38 版，并将其缓冲池大小配置为 8 GB，重做日志容量配置为 4 GB。

TPC-C 吞吐量。对于 TPC-C，我们使用 Benchbase [21] 提供的标准数据和工作负载生成器。我们调整工作线程数和要合并的事务数，并在此报告最大吞吐量。每项实验运行三次，并报告平均值。

图 6a 和图 6b 测量了四种设置：启用了 JDBC 批处理的原始版本（原始）；在客户端进行事务内合并的版本（无 RPC 的事务内合并）；使用合并器仅进行事务内合并的版本（RPC 事务内合并），以测量合并器的开销；以及在合并器上同时进行事务内和跨事务合并的完整 TransactionMerger 版本（RPC 全事务合并）。

如果我们只运行 NEW-ORDER 事务（图 6a），那么仅事务内合并就已经能够带来显著的改进。这是因为 NEW-ORDER 事务包含许多可以合并以提升性能的插入和更新语句（见图 3）。请注意，即使没有合并器，也可以实现这一改进。

通过跨事务合并获得的进一步改进取决于具体设置。在只有一个仓库的情况下，改进并不显著。原因在于，两个 NEW-ORDER 事务存在两个潜在的争用点：一个是更新每个区域的下一个订单 ID（d next o id），另一个是更新每个仓库的商品库存。当两个事务针对同一区域时，会发生第一个争用；当两个事务购买同一商品时，会发生第二个争用。由于 TPC-C 使用偏斜分布来确定要购买的商品，因此第二种情况很可能发生。

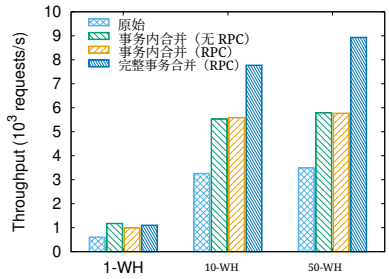
虽然我们根据仓库区域划分事务的方法可以将第一个竞争点转变为优化点，但无法解决第二个竞争点，因为不同区域的事务可能会购买同一件商品。因此，如果只有一个仓库，瓶颈就在第二个竞争点，改进幅度也就有限。不过，仓库数量增加后，针对不同仓库的事务便可并行运行。

这些实验展示了进一步提升性能的潜力。在这个具体示例中，我们可以根据（区域、商品）元组划分事务，从而同时解决两个争用点。一般而言，我们可能需要一种通用算法来分析争用并划分事务。

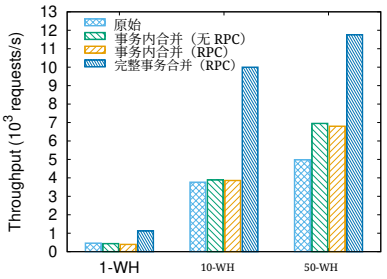
包含全部五种事务类型时，结果呈现出不同的模式（图 6b）。首先，除 50 个仓库的情况外，事务内合并并未带来显著提升。这是因为，在 1 个仓库和 10 个仓库的实验中，主要瓶颈是付款事务中更新每个仓库年初至今金额（w ytd）时产生的争用，而事务内合并无法解决争用问题。请注意，这种争用比新订单事务中的争用更严重，因为付款事务的竞争点是每个仓库的 w ytd 值，而新订单事务的竞争点是每个区域的 d next o id 值，并且每个仓库有 10 个区域。不过，在 50 个仓库的情况下，争用较轻，事务内合并便能带来提升。其次，跨事务合并在所有设置下都能持续提升吞吐量，原因相同。由于付款事务中的争用是主要瓶颈，且付款事务不像新订单事务那样有多个竞争点，因此总能通过跨事务合并解决争用。即使在争用较轻的 50 个仓库场景中，跨事务合并仍可减少数据库需要执行的更新数量。

总体而言，TransactionMerger 完整版可将 TPC-C 的吞吐量提升 1.82 至 2.65 倍。

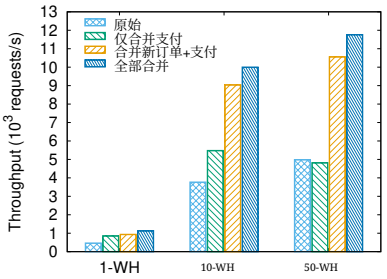
图 6c 衡量了合并部分事务的效果（完整版本会合并新订单、支付和配送）。如图所示，对于 1 个仓库的情况，仅合并支付就能获得大部分收益；对于 50 个仓库的情况，仅合并新订单就能获得大部分收益；对于 10 个仓库的情况，分别合并两者都会带来提升。这些实验表明，开发者可以逐步将合并应用于应用程序，首先



(a) TPC-C 新建订单事务的吞吐量



(b) TPC-C 五种事务的吞吐量。



(c) TPC-C 合并部分事务。

操作。

图6：不同设置下 TPC-C 的吞吐量。

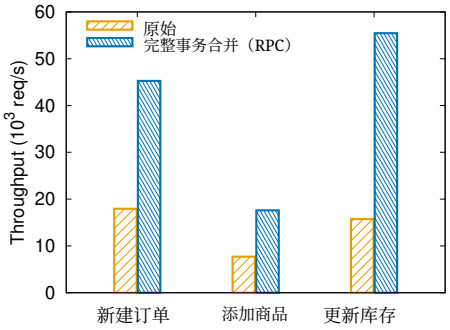


图7：Spree 的吞吐量。

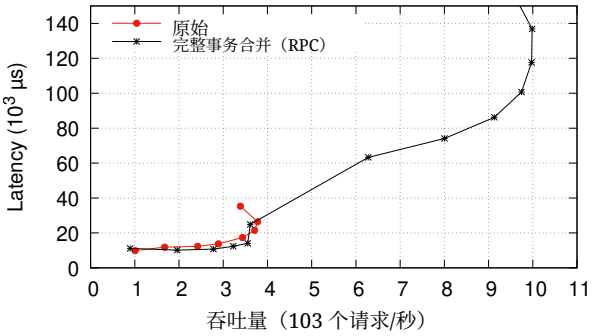


图 8：10 个仓库下 TPC-C 的延迟和吞吐量。

来自性能关键事务。

Spree 吞吐量。Spree 没有标准的数据和工作负载生成器，因此我们尽力创建了一个。具体而言，在实验开始之前，我们会生成 30000 个用户、每位用户一个订单、100000 个产品，以及包含所有商品的库存。每个订单包含库存中的 1 件商品。我们还使用范围内的随机值生成价格、税费和库存位置相关信息。实验期间，我们按照 IV-C 节中的讨论运行新订单和添加商品事务，并使用范围内的随机值生成它们的参数值。

图7展示了 Spree 的吞吐量。正如第 IV 节所述，Spree 没有事务内合并的机会。跨事务合并仍能带来显著提升，最高可达 3.52 倍。

对延迟的影响。由于中间件可能会增加事务处理的额外延迟，图8测量了处理 TPC-C 请求（全部五种类型）的延迟。如图所示，只有在吞吐量较低时（即每秒低于 1K 个请求），TransactionMerger 的延迟才会高于原始版本。随着吞吐量增加，TransactionMerger 的改进有助于减少排队延迟，从而抵消中间件引入的额外延迟。

由于 TransactionMerger 旨在提高吞吐量，因此应在关注吞吐量时使用。当吞吐量较低时，客户端始终可以绕过 Transaction-

表I：TPC-C 中的资源消耗与吞吐量提升。吞吐量↑ 定义为使用合并器时的最大吞吐量与仅使用客户端事务内合并时的最大吞吐量之比。CPU↑ 定义为使用 TransactionMerger 时（合并器 CPU 利用率 +数据库 CPU 利用率）与使用客户端事务内合并时的数据库 CPU 利用率之比。

设置	吞吐量↑	CPU↑
1-WH	$\frac{1126}{438} = 2.57$	$\frac{200\%+400\%}{300\%} = 2.00$
10-WH	$\frac{9996}{3893} = 2.57$	$\frac{1400\%+2300\%}{1500\%} = 2.47$
50-WH	$\frac{11759}{6952} = 1.69$	$\frac{1600\%+3000\%}{2800\%} = 1.64$

合并器并生成直接发送至数据库的 SQL语句（图2）。此外，借助 TransactionMerger 对在线重新配置的支持，此类设置可以在运行时动态调整。

合并器的资源消耗。在 TransactionMerger 中，合并器会消耗额外的资源（在我们的实验中，CPU 是主要资源），这就引出了一个问题：TransactionMerger 的方法是否值得采用。换句话说，我们能否将这些额外资源用于数据库以获得更高的吞吐量，而不是将其用于合并器?

表I展示了一些具体数据点。如表中所示，对于 TransactionMerger，吞吐量的提升值得额外的 CPU 利用率。另一方面，由于原始版本无法充分利用可用的 CPU，

增加数据库端的资源可能不会显著提升吞吐量。在争用严重时，这种现象尤为明显（例如，1-WH 情况下 CPU 利用率为 300%，而最大 CPU 利用率为 4000%）。总体而言，与增加数据库资源相比，TransactionMerger 提供了一种灵活的解决方案：TransactionMerger 在低争用和高争用场景下都能带来性能提升；由于 TransactionMerger 的无状态特性，我们可以随时增加或移除资源；对在线重新配置的支持使 TransactionMerger 能够适应工作负载。然而，扩展数据库端通常涉及重新分区或迁移数据，这更加复杂且成本高昂，并且在高争用场景下效果不佳。在实践中，我们认为这两种方法可以互为补充。

适应动态工作负载。为评估 TransactionMerger 在线重新配置的有效性，我们测试贝叶斯优化能否引导系统为 TPC-C 找到最优配置。我们将要合并的事务数量随机设置为 1 到 10，并将线程数设置为 1 到总区域数。我们从 20 个随机配置样本开始，在每个样本下运行 TransactionMerger 30 秒，以测量吞吐量。我们使用这 20 个样本训练高斯过程回归模型，然后利用贝叶斯优化推荐最优配置并重新训练模型。我们重复这一优化过程，直到收敛；这里的收敛定义为配置不再变化且预期吞吐量趋于稳定。对于 1-WH、10-WH 和 50-WH，在训练高斯过程回归模型后，分别经过 1、4 和 2 次优化迭代即可收敛。当然，开发者可以利用其专业知识缩小搜索空间，以加快这一过程。

VI. 相关工作

应用侧优化。 开发者可以通过多种方式优化数据库应用的性能。例如，OLTPShare [24] 会将单语句只读事务批量执行，以提高高负载场景下的吞吐量。其后续工作扩展到包含更新的多语句事务，但仍只合并读操作，并按常规方式执行写操作 [7]。专注于只读语句的好处是，这些技术可以在中间层自动应用，无需修改应用程序代码。Marroqu´ın 等人进一步将这一思路应用于基于云的查询即服务（QaaS）系统中的查询 [25]。该方法调整关系运算符以支持批量执行，并通过 SQL 重写来表达共享执行计划，使用了数组操作和 SQL 结构中的展开运算符等技术。对于简单情况，它可以自动完成重写，但复杂查询仍需开发者付出努力。我们的工作进一步探索了合并写操作的好处，但需要修改应用程序代码。

对于争用型工作负载，一种常见的优化方式是将发生争用的商品拆分到多个分区。例如，如果某件热点商品正在促销，查询和更新其商品数量可能会成为争用瓶颈。我们可以将该商品拆分到多个分区，每个分区存放部分商品数量，并让客户随机选择一个分区。不过，由于某个分区可能已经售罄，而其他分区仍有库存，因此负载均衡可能会成为问题。我们的方法通过仍然维护单一库存数量，并在争用的语句到达数据库之前将其合并，从而避免负载均衡问题。

其他优化技术包括但不限于：在可能的情况下，重写事务，将发生争用的语句移至靠后的位置；将一个事务拆分为多个事务等 [5], [6]。这些技术与我们的工作在很大程度上是正交的。

数据库端优化。有大量工作致力于优化数据库的不同方面，例如并发控制 [1], [2], [26]–[29]、提交协议 [30]–[31]、复制 [1], [4], [32], 、查询规划 [3], [33], [34],等（由于相关工作太多，此处列举并不完整）。据我们所知，最相关的工作是 SharedDB：它将多个查询合并为一个包含共享算子的全局计划，并对这些查询进行协同处理，从而减少冗余工作 [35]。我们的工作为在应用程序端执行此类优化提供了另一种方式。如前所述，应用程序端优化具有一些独特的机会，而这些机会几乎无法在数据库端实现自动化。

用于划分事务的算法。为提升数据库性能，分区方法会将事务分配到不同分区以便并发执行，尽量减少分区间的冲突，从而降低分布式事务的开销。Schism 使用基于图的最小割分区算法来划分事务并平衡负载，尽量减少跨分区边的数量，即分布式事务的数量 [36]。Strife 在运行时将事务分组成批次，通过数据访问图分析冲突，并将其划分为无冲突集群和剩余项 [37]。其分区算法分三步进行：识别 k 个热点数据项（每个集群一个），将事务划分到集群或剩余项中，再合并集群以减少剩余项。Horticulture 根据访问图按频繁访问的列拆分表，并使用属性参数将事务分配到分区 [38]。

我们的工作提出在可行时合并竞争事务。根据我们在 TPC-C 中的经验，简单的基于启发式的分区在大多数情况下已足够。对于更复杂的争用，我们预计这些算法可以帮助 TransactionMerger 以通用的方式确定合并策略。

VII. 结论

本文表明，通过在事务内部及事务之间合并相似语句，我们可以显著提升数据库应用程序的性能，而无需更改数据库后端。所需工作并非易事，但可以逐步开展。因此，事务合并为提升数据库性能提供了一种替代方案，尤其是在数据库侧优化效率低下或无法使用时。

参考文献

[1] A. Thomson、T. Diamond、S.-C. Weng、K. Ren、P. Shao 和 D. J. Abadi，“Calvin：面向分区数据库系统的快速分布式事务”，载于《2012 年 ACM SIGMOD 数据管理国际会议论文集》，系列：SIGMOD ’12。美国纽约州纽约市：ACM，2012 年，第 1–12 页。 [在线]。获取地址：http://doi.acm.org/10 1145 2213836 2213838 。/。S. Tu、W. Zheng、E. [2] Kohler、B. Liskov 和 S. Madden，“多核内存数据库中的快速事务”，载于《第二十四届 ACM操作系统原理研讨会论文集》，系列：SOSP ’13。美国纽约州纽约市：ACM，2013 年，第 18–32 页。 [在线]。获取地址：http://doi.acm.org/10 1145 2517349 2522713 。/。V. Leis、 [3] A. Gubichev、A. Mirchev、P. Boncz、A. Kemper 和 T. Neumann，“查询优化器究竟有多出色？”，《VLDB 基金会会刊》，第 9 卷，第 3 期，第 204–215 页，2015 年。[4] I. Zhang、N. K. Sharma、A. Szekeres、A. Krishnamurthy 和 D. R. K. Ports，“通过不一致复制构建一致性事务”，系列：SOSP ’15。美国纽约州纽约市：计算机协会，2015 年，第 263–278 页。[5] D. Shasha、F. Llirbat、E. Simon 和 P. Valduriez，“事务切分：算法与性能研究”，ToDS，1995 年。[6] C. Xie、C. Su、C. Littley、L. Alvisi、M. Kapritsos 和 Y. Wang，“通过模块化并发控制实现高性能 ACID”，载于《第 25 届操作系统原理研讨会论文集》，系列：SOSP ’15。美国纽约州纽约市：ACM，2015 年，第 279–294 页。 [ ] 10 1145 2815400 2815430 [7] 在线。获取地址：http://doi.acm.org/。/。R. Rehrmann、C. Binnig、A. Bohm、K. Kim 和 W. Lehner，“不同隔离级别下 OLTP工作负载的共享机会”，《VLDB基金会会刊》，第 13 卷，第 10 期，第 1696–1708 页，2020 年 6 月。[8] C. Cheng、M. Han、N. Xu、S. Blanas、M. D. Bond 和 Y. Wang，“这是开发者的责任还是数据库的责任？重新思考数据库中的并发控制”，载于 CIDR，2023 年。[9] C. Tang、Z. Wang、X. Zhang、Q. Yu、B. Zang、H. Guan 和 H. Chen，“Web应用程序中的临时事务：好的一面、坏的一面和糟糕的一面”，载于 SIGMOD，2022 年。[10] “Statement（Java平台SE 8）- EXECUTEBATCH”，https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html。[11] 事务处理性能委员会，“TPC-C 主页”，http://www.tpc.org/tpcc/。 [在线]。获取地址： \网址{ http://www.tpc.org/ tpcc/[12] R. Rehrmann，“OLTP工作负载中的查询合并”，2023 年。[13] “Spree Commerce文档”，https://spreecommerce.org/docs/ user/what-is-spree-commerce，2024 年。[14] O. Alipourfard、H. H. Liu、J. Chen、S. Venkataraman、M. Yu 和 M. Zhang，“CherryPick：自适应发掘大数据分析的最佳云配置”，载于《第 14 届 USENIX网络系统设计与实现研讨会（NSDI 17）》，美国马萨诸塞州波士顿，2017 年 3 月，第 469–482 页。[15] B. Behzad、S. Byna、Prabhat 和 M. Snir，“通过自动调优优化高性能计算应用程序的 I/O性能”，《ACM 并行计算汇刊》，第 5 卷，第 4 期，2019 年 3 月。 [在线]。获取地址：https://doi.org/10 1145 3309205 。/ M. Agarwal、D. Singhvi、P. Malakar [16] 和 S. Byna，“基于主动学习的并行 I/O性能自动调优与预测”，载于《2019 年 IEEE/ACM第四届国际并行数据系统研讨会（PDSW）》，2019 年 11 月，第 20–29 页。[17] N. Rajesh、K. Bateman、J. L. Bez、S. Byna、A. Kougkas 和 X.-H. Sun，“TunIO：用于优化高性能计算 I/O的人工智能驱动框架”，载于《2024 年 IEEE国际并行与分布式处理研讨会（IPDPS）》。IEEE，第 494–505 页。[18] “gRPC”，https://grpc.io/。[19] “使用 scikit-learn 的 GaussianProcessRegressor”，https://scikit-learn.org/stable/modules/generated/sklearn/gaussian\ process.GaussianProcessRegressor.html。[20] “使用 skopt 进行贝叶斯优化”，https://scikit-optimize.github.io/ stable/auto examples/bayesian-optimization.html。[21] “Benchbase”，https://github.com/cmu-db/benchbase。[22] “Spree”，https://github.com/spree/spree。[23] “CloudLab”，https://www.cloudlab.us。[24] R. Rehrmann、C. Binnig、A. B¨ohm、K. Kim、W. Lehner 和 A. Rizk，“OLTPShare：OLTP工作负载共享的理由”，《VLDB基金会会刊》，第 11 卷，第 12 期，第 1769–1780 页，2018 年 8 月。[25] R. Marroqu´ın、I. M¨uller、D. Makreshanski 和 G. Alonso，“付出一份，免费获得数百份：通过共享查询执行降低云成本”，载于*ACM 云计算研讨会论文集*，丛书 SoCC ’18。美国纽约州纽约市：计算机协会，2018 年，第 439–450 页。[26] X. Yu、A. Pavlo、D. Sanchez 和 S. Devadas，“TicToc：穿越时间的乐观并发控制”，载于*2016* 年数据管理国际会议论文集，丛书 SIGMOD ’16。美国纽约州纽约市：计算机协会，2016 年，第 1629–1642 页。[27] S. Mu、L. Nelson、W. Lloyd 和 J. Li，“整合并发控制与共识，以处理冲突下的提交”，载于第 *12* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 16*）。佐治亚州萨凡纳：USENIX 协会，2016 年，第 517–532 页。 [在线]。可访问：https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu [28] B. Ding、L. Kot 和 J. Gehrke，“通过事务批处理和操作重排序改进乐观并发控制”，*VLDB* 基金会会刊，第 12 卷，第 2 期，第 169–182 页，2018 年。[29] Y. Lu、X. Yu、L. Cao 和 S. Madden，“Aria：一种快速实用的确定性 OLTP 数据库”，*VLDB* 基金会会刊，第 13 卷，第 12 期，第 2047–2060 页，2020 年。[30] S. Maiyya、F. Nawab、D. Agrawal 和 A. E. Abbadi，“统一共识与原子提交，实现有效的云数据管理”，*VLDB* 基金会会刊，第 12 卷，第 5 期，第 611–623 页，2019 年。[31] Y. Lu、X. Yu、L. Cao 和 S. Madden，“分布式 OLTP 数据库中的基于纪元的提交与复制”，2021 年。[32] M. Abebe、B. Glasbergen 和 K. Daudjee，“Dynamast：复制系统的自适应动态主控”，载于*2020* 年第 *36* 届 *IEEE* 国际数据工程会议（*ICDE*）。IEEE，2020 年，第 1381–1392 页。[33] R. Marcus、P. Negi、H. Mao、N. Tatbul、M. Alizadeh 和 T. Kraska，“Bao：让学习型查询优化走向实用”，载于*2021* 年数据管理国际会议论文集，2021 年，第 1275–1288 页。[34] Y. Zhao、G. Cong、J. Shi 和 C. Miao，“Queryformer：用于查询计划表示的树形 Transformer 模型”，*VLDB* 基金会会刊，第 15 卷，第 8 期，第 1658–1670 页，2022 年。[35] G. Giannikis、G. Alonso 和 D. Kossmann，“SharedDB：一石消灭千条查询”，*VLDB* 基金会会刊，第 5 卷，第 6 期，第 526–537 页，2012 年 2 月。[36] C. Curino、E. Jones、Y. Zhang 和 S. Madden，“Schism：一种由工作负载驱动的数据库复制与分区方法”，*VLDB* 基金会会刊，第 3 卷，第 1–2 期，第 48–57 页，2010 年 9 月。[37] G. Prasaad、A. Cheung 和 D. Suciu，“使用快速动态分区处理高竞争 OLTP 工作负载”，载于*2020* 年 *ACM SIGMOD* 数据管理国际会议论文集，2020 年，第 527–542 页。[38] A. Pavlo、C. Curino 和 S. Zdonik，“共享无关架构并行 OLTP 系统中的感知偏斜的自动数据库分区”，载于*2012* 年 *ACM SIGMOD* 数据管理国际会议论文集，2012 年，第 61–72 页。