arXiv:2603.02108v1 [cs.DB] 2 Mar 2026

Milliscale：低延迟对象存储上的快速提交

Jiatang Zhou 西蒙弗雷泽大学
jiatangz@sfu.ca

Kaisong Huang 卡尔加里大学
kaisong.huang@ucalgary.ca

Tianzheng Wang
西蒙弗雷泽大学
tzwang@sfu.ca

摘要

近期，以 Amazon S3 Express One Zone 为代表的低延迟对象存储服务具备毫秒级延迟并支持可变对象，因此成为 OLTP 引擎直接提交事务和持久化操作数据的理想选择，同时还能透明地提供强一致性、高持久性和高可用性。然而，由于 S3 Express One Zone 和现代去中心化日志记录的特性，简单采用这些服务仍可能导致较高的提交延迟。

本文介绍了 Milliscale，这是一种针对低延迟对象存储进行内存优化的 OLTP 引擎。Milliscale 采用新技术来降低提交时延并减少对象访问请求的数量，从而优化提交延迟。使用具有代表性的基准测试进行的评估表明，与基线相比，Milliscale 能显著降低提交延迟，同时保持高吞吐量。

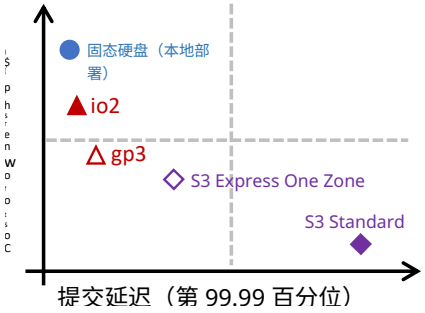


图 1：提交延迟与成本。传统对象存储（S3 Standard）和块存储（本地 SSD、io2 和 gp3）分别会导致较高的提交延迟和成本。

1 引言

对象存储服务（例如 Amazon S3、Google Cloud Storage 和 Azure Blob Storage）以低成本提供弹性存储、高可用性（超过 99.9% [5, 17]）和高持久性（11个9 [5]）。它们还支持通过 HTTP 请求进行非常灵活的访问。这些特性对数据库系统极具吸引力，分析领域广泛采用对象存储便是明证 [8, 12, 14, 15, 19, 28]。

尽管这些特性对联机事务处理也极具吸引力，但将对象存储用作联机事务处理引擎的主要存储后端，以对象的形式直接持久化预写式日志，尚未成为主流。主要原因是传统对象存储的延迟较高。如图 1 所示，S3 Standard 成本低，但延迟非常高。相比之下，块存储服务（例如 Amazon EBS gp3 和 io2 [2]）可实现显著更低的毫秒级提交延迟，但成本要高得多。有些服务提供的持久性保障也低于 S3（例如，EBS gp3 为两个 9，而 S3 为 11个 9）。传统对象存储服务也很少支持高效更新（如果支持的话）。因此，大多数现有的云端联机事务处理服务都使用专有日志服务 [7, 13], 高端块存储（例如 EBS io2），或带 SSD缓存 的对象存储 [8]。缺点是，这会增加系统实现的复杂性（例如，需要维护——引入单独的内部服务，或绕过 S3 Standard 的不可变性限制 [8]），并增加部署复杂度（例如，要求使用配备 NVMe 固态硬盘的特定计算实例）。

1.1 低延迟对象存储上的日志记录

以 Amazon S3 Express One Zone [3] 为代表的新型低延迟对象存储服务，可提供（1）个位数毫秒级延迟，以及（2）支持追加操作的可变对象。这些特性相比 S3 Standard 的百毫秒级延迟和不可变对象有了显著提升，也引出了一个有趣的研究问题：*OLTP 引擎能否通过直接向 S3 Express One Zone 这类对象存储服务记录日志，透明地实现高持久性和快速提交？*

为回答这个问题，我们着手改造 ERMIA [27], 这一内存优化型 OLTP 引擎，使其将日志记录刷新到 S3 Express One Zone。虽然 S3 Express One Zone 支持透明的高可用性、高持久性和灵活的日志访问，有助于简化热备等功能，但我们发现事务提交延迟仍然较高。如图 1 所示，使用 S3 Express One Zone 时，提交延迟可能会 ~2× 更高，而成本则接近 gp3。我们观察到以下两个主要原因。

过多的对象追加请求。现代数据库管理系统通常采用去中心化日志记录 [38, 41, 45]，以避免集中式日志记录瓶颈 [23]。常见做法是按线程进行日志记录，为每个事务工作线程分配一个专用日志缓冲区。日志缓冲区大小和日志缓冲区刷新的频率直接决定提交延迟。使用更多并行缓冲区有助于缓解集中式日志记录瓶颈，而使用更小的日志缓冲区、更频繁或更及时地刷新，则有助于降低提交延迟。这两种做法都会增加日志刷写（即对象追加）请求的数量，进而需要更多网络往返来刷新日志记录。一些现代内存优化型数据库管理系统也倾向于在日志中存储完整记录 [37, 39]（而非可能小得多的增量），以简化实现。这会进一步增加日志数据量，从而再次增大提交路径上的数据传输延迟。此外，云对象存储服务采用按使用量付费的 [4] 定价模式，收费依据包括数据量、存储时长和对象访问请求数量。因此，请求数量增加也会导致成本上升。

提交依赖。在去中心化日志记录中，要提交事务 T，OLTP 引擎必须确保 T 自身的以及依赖的日志记录（这些记录可能由其他线程在其私有日志缓冲区中生成）[34]均已持久化。与使用单个日志缓冲区相比，这自然会增加提交延迟，原因是

可能需要（1）刷新多个日志缓冲区，以及（2）跟踪依赖关系 [24, 33]。也就是说，事务 T 可能会读取或覆盖另一个事务 S 在另一份日志中记录的结果。提交 T 需要刷新两个线程的日志缓冲区，从而增加提交延迟。更糟的是，由于其序列化顺序较晚， T 可能不得不等待持久化它并不依赖的日志缓冲区。此类假阳性依赖可能导致提交延迟进一步增加。

1.2 Milliscale

本文介绍了 Milliscale，这是一款内存优化的联机事务处理引擎，可在云端低延迟的可变对象存储上实现快速提交。Milliscale 的核心是一种新型去中心化日志记录协议，并通过优化分别减少过多的日志刷写请求和不必要的依赖延迟。

在探索减少对象追加请求的方法时，我们得出了两点观察结果。(1) 日志刷写的频率直接受日志缓冲区填满速度的影响，而填满速度又取决于面向每个缓冲区的并发请求数量。(2) 在现代 CPU 上，同一套接字内少量线程（例如 2–4 个）对单个日志缓冲区的争用可以忽略不计。基于这些观察，我们摒弃常见的按线程日志记录方式，提出了受限去中心化日志记录：我们不再为每个线程分配一个日志缓冲区，而是为每组（例如两个）线程分配一个按比例增大（例如 2×）的日志缓冲区。正如第 4 节将详细说明的那样，通过根据对象存储的性能特征谨慎选择日志缓冲区大小，受限去中心化日志记录能够减少对象追加请求，同时不会影响提交延迟。

在受限去中心化日志记录的基础上，Milliscale 采用轻量级记录级依赖跟踪 方法，减少假阳性依赖造成的延迟。它不采用页面级跟踪 [20, 38] 或全局持久化提交时间戳 [1], ，而是让每个事务 T跟踪其最近的直接前驱的提交时间戳 P，该前驱生成了 T读取或更新的记录。一旦此类事务 P已提交， T便可安全提交。为此，系统维护一个事务本地变量，并在读取或更新记录时更新该变量。这一过程十分轻量，只需获取每个版本的提交时间戳，而底层并发控制协议通常已经维护了该时间戳。借助细粒度的记录级跟踪，Milliscale 避免了大量会造成不必要延迟的假阳性依赖。

请注意，我们的目标并非 实现尽可能低的提交延迟并与基于 NVMe SSD 的解决方案竞争，而是展示现代对象存储在提供令人满意的提交延迟和吞吐量方面的潜力。因此，我们不会采用 NVMe 固态硬盘等中间层，因为它们具有临时性，且仅适用于某些计算实例类型。总体而言，Milliscale 的提交延迟仍高于使用 NVMe 固态硬盘时的延迟。不过，我们认为，这是朝着云原生 OLTP 服务迈出的有意义的一步，这类服务具有更简单的系统架构，同时提供丰富的持久性和可用性功能。

我们基于前述 OLTP 引擎 ERMIA 构建了 Milliscale。ERMIA 的日志子系统是为块存储设计的。Milliscale 继承了 ERMIA 以内存为中心的优化（例如索引和并发控制），但替换了 ERMIA 的日志子系统，采用我们针对低延迟可变对象存储提出的新型去中心化日志记录方案。我们在一台拥有 32 个 vCPU 的 Amazon EC2 c6in.8xlarge 实例和 S3 Express One Zone 上，使用 YCSB 微基准测试和 TPC-C 进行评估。结果表明，与基线相比，Milliscale 的吞吐量具有竞争力，且尾延迟显著更低（最高降低 51.9%）。具体而言，对于写密集型工作负载，Milliscale 可将未经优化的 S3 Express One Zone 的 99.99 百分位提交延迟降至与 gp3 相近的水平，同时具备与 S3 Standard 相同的弹性和一致性保证。

Milliscale 中的技术也适用于其他对象存储服务。受限去中心化日志记录不仅可用于将数据缓冲区写入存储的日志记录场景（例如，在繁忙系统中减少 I/O 请求的数量），也可应用于其他场景。还可以通过添加基于 SSD 的缓存进一步优化读取路径 [8, 19]。我们将这些方向留作未来工作。

1.3 贡献与论文组织

我们做出了两项贡献：我们在对象存储之上运行联机事务处理的理念，并论证了直接将预写式日志持久化到低延迟对象存储的可行性。 我们提出了一种新的去中心化日志记录方法，可同时优化吞吐量和提交延迟。 基于所提出的技术，我们构建了 Milliscale，这是首个可直接向低延迟对象存储提交事务的内存优化型联机事务处理引擎。 我们对 Milliscale 和各项基线进行了全面评估，以验证我们的设计决策。Milliscale 已开源，地址为 https://github.com/sfu-dis/milliscale。

接下来，第 2 节介绍云存储的必要背景，并对 S3 Express One Zone 进行分析，以阐明我们开展这项工作的动机。第 3–5 节介绍 Milliscale 的设计与评估。我们在第 6 节讨论相关工作，并在第 7 节总结。

2 背景

本节介绍低延迟对象存储和日志记录的必要背景，以阐明我们开展这项工作的动机。

2.1 对象存储基础

对象存储将数据项视为“对象”，这些对象由唯一键标识，并通过 HTTP 请求访问，而非使用基于文件的 POSIX [21] API。对象是字节容器，多个对象可以组合存储在一个存储桶中。例如，Amazon S3 提供 GetObject/PutObject API，接受一个键作为输入（以及其他可能的参数），并执行对象读写操作。此外，它还提供用于删除和元数据查询等操作的其他 API。请求可以通过数据中心网络或互联网从任意计算客户端发出，但大多数高性能部署会采用前者以降低延迟；本工作也假设如此。

得益于现代数据中心网络 [15]，大多数云服务提供商都能提供具备强一致性和充足带宽的对象存储服务。尽管这些服务的内部设计各不相同，但它们通常采用已成为事实标准的 S3 API。因此，本文以 Amazon S3 为研究对象，并将探索其他服务留作未来工作。

传统对象存储服务（例如 S3 Standard）的特点是具备强持久性（例如 11个9）、强一致性、高可用性（超过 99.9%）和低成本。然而，其缺点是访问延迟较高（约为 100 毫秒），且对象不可变（即写入后不允许更新）。这些特点使传统对象存储的使用主要局限于分析等读密集型工作负载 [15, 19]。尽管某些系统 [8, 44] 广泛使用对象存储来保存操作数据，但它们需要采用复杂技术来规避 S3 Standard 的不可变性，并依赖积极的缓存策略，使用仅在高端计算实例上提供的本地 NVMe 固态硬盘。

2.2 S3 Express One Zone：20 年后的 S3

Amazon S3 Express One Zone 等低延迟对象存储服务通过个位数毫秒级延迟和支持通过追加操作实现可变对象，改进了传统服务。

对象操作。 S3 Express One Zone 还提供基于 HTTP 的 API，例如 GetObject/PutObject。通过指定写入偏移量（A WS S3 SDK 中的 writeOffsetByte），可使用 PutObject 发起追加操作，该偏移量即为对象末尾。AWS 当前的实现允许对象最多被追加 10,000 次 [6]。每次追加操作都会在对象中创建一个“部分”，每次追加（即一个部分）的最大大小为 5GB。

性能。 我们从 AWS EC2 实例发出 S3 请求，以评估 GetObject 和 PutObject 的延迟（详细设置见第 5 节）。完整对象写入的行为与追加操作类似，因此我们省略其结果，重点分析与我们的使用场景（预写日志）最相关的追加操作。

如图2所示，对于128–512KB的请求大小，S3 Express One Zone的追加延迟相近（~8ms）。超过512KB后，追加延迟呈线性增长。在2MB请求大小下，S3 Standard的延迟高得多，为 ~77ms，而 S3 Express One Zone的该数值低 ~71%（~22ms）。这意味着512KB是避免过高网络开销的最小请求大小。由于一个对象最多可接受10,000次追加，在512KB请求大小下，最大对象大小为4.88GB。数据库管理系统必须能够将日志切分为独立对象，我们将在后文说明。

使用GetObject检索数据时，对于S3 Standard和S3 Express One Zone而言，通常都比写入更快。后者表现出 ~5毫秒的延迟适用于所有不超过 256KB 的请求大小，如图2所示。将请求大小降至此阈值以下不会带来进一步收益；将其增至 1MB 也只会略微增加延迟。请求大小超过 2MB 后，读取延迟开始线性增长。

这些结果表明，S3 Express One Zone 的理想请求大小介于 256KB 到 1MB 之间，这可能使提交 OLTP事务的延迟上限达到 ~10 毫秒，仅为使用 S3 Standard 时延迟的十分之一。稍后我们将讨论 Milliscale 如何利用这一观察结果。

可用性与持久性。 与会在三个可用区之间复制数据的 S3 Standard 相比，S3 Express One Zone 将数据存储在单个可用区中，以降低延迟。其缺点是可用性低于 S3 Standard，因此需要构建在 S3 Express One Zone 之上的数据库管理系统来缓解这一问题。一个直接的解决方案是让 OLTP 引擎将对象复制到其他可用区或区域中的额外 S3存储桶。

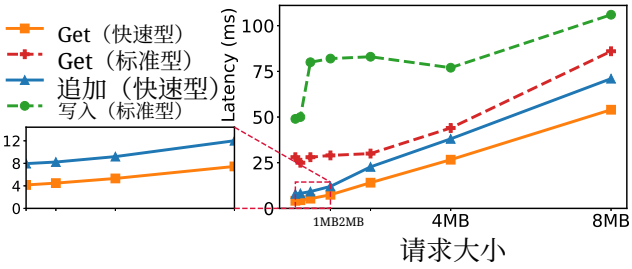


图2：不同请求大小下 S3 Standard 与 S3 Express One Zone 的对象获取、追加和写入延迟（毫秒）。S3 Express One Zone 在较小的请求大小（例如 512KB）下可提供个位数毫秒的延迟，因此非常适合联机事务处理（OLTP）。

定价。 S3 Express One Zone 采用按需付费定价模式，根据实际用量收费。费用包括：(1) 存储成本，按数据的存储量和存储时长计费；(2) 对象访问请求数量（例如，每次 PutObject 均需付费）；以及 (3) 客户端（例如 EC2实例）与 S3 之间的带宽用量。存储成本仅与数据大小和存储时长有关（与其他云存储的计费方式类似），而另外两项则与请求数量密切相关。截至 2026年初，AWS 每百万次请求收费 1.13 美元。此外，上传到 S3 Express One Zone 的每 GB 数据需支付 0.0032 美元。上传的数据量则由上传请求数量 × 请求大小共同决定。例如，通过发出 100 万次追加操作（即每次请求 2 MB 或 0.002 GB）来上传 2 GB 数据，费用为 $1.13 + 1100 万次请求 × 00.002GB × $0.0032。1.13 美元仅为发出这 100 万次请求的费用，其余部分则是上传 2 GB 数据的费用。因此，减少 S3 请求数量（这是我们工作的主要目标）也有助于降低成本。

2.3 为什么在 S3 Express One Zone 上运行联机事务处理?

乍看之下，S3 Express One Zone 的定价模式似乎不适合写入频率高的联机事务处理。然而，鉴于 S3 的诸多理想特性以及其他替代方案的缺点，我们认为，S3 Express One Zone 在以下几个方面是理想之选。

首先，像 S3 Express One Zone 这样的服务如今能够提供前面提到的强一致性、高可用性和强持久性，而且这一切都透明地实现。这可以大幅简化云原生数据库系统的构建。此类系统大多依赖由数据库管理系统供应商内部团队维护的定制日志服务，例如华为 Taurus 中的 PLOG [13] 和 Microsoft Socrates 中的 XLOG [7]。很多时候，如果没有这类专有服务，人们会使用 EBS 等块存储服务，但它们并不总能提供同等程度的保证。

其次，S3 Express One Zone 继承了其标准型对应产品通过 HTTP 请求灵活访问数据的特性。多个计算节点可直接访问对象，这简化了热备架构，备份服务器可以直接读取主节点生成的对象。相比之下，EBS gp3 等典型块存储服务不支持多个实例同时访问同一

时间的卷，或者存在限制（例如，io2 的“多重挂载”功能允许多台服务器挂载同一卷，但在 ext4 和 XFS 等常用文件系统下，不允许它们同时访问数据 [9]）。如果没有 S3 Express One Zone，数据库管理系统构建者就必须投入额外精力并采用额外硬件，才能实现上述功能。比较服务成本时，通常不会将这些额外的人力资本成本纳入考量。

最后，日志空间通常有上限，并通过定期检查点重复利用。这意味着第 2.2 节中提到的存储成本会趋于稳定，而不会随着数据量增长，因此每次往返的请求数量和传输的数据总量将成为唯一需要优化的指标。对这些指标进行优化，与现代预写日志的工作方式密切相关，接下来我们将对此进行介绍。

2.4 现代去中心化日志记录

传统的预写日志采用集中式架构：所有工作线程都争相写入同一个日志缓冲区，然后该缓冲区会定期刷新，以组提交事务。现代 OLTP 引擎大多已弃用这种模式，以避免集中式日志缓冲区上的争用 [24]。一种常见做法是采用去中心化的按线程日志记录，为每个工作线程分配一个专用日志缓冲区 [37, 38, 41, 45], ，从而消除日志缓冲区争用。每个日志缓冲区都会定期刷新到稳定存储（例如，在组提交时或超时后），方法是追加写入 S3 中的文件或对象。

流水线组提交。 在去中心化日志的基础上，一些系统采用流水线组提交 [23] 来提高吞吐量；我们假设 Milliscale 使用了这种优化，并据此进行设计。采用流水线组提交后，事务一旦“预提交”，即完成所有逻辑处理并将所有日志记录写入日志缓冲区，就会与工作线程分离。随后，事务会进入提交队列，等待最终提交。与此同时，原先负责处理该事务的线程 T 可以继续处理另一个事务。后台线程会监控当前持久化的日志序列号（LSN），并将事务出队；待其日志记录持久化后，再通知客户端。

提交依赖。流水线组提交使工作线程不再受日志刷新输入/输出的阻塞，但可能在事务之间形成依赖，因此需要在提交。考虑事务 T 更新了记录 x 并刚刚预提交，使 T 的结果立即可见。任何访问 x 的事务 S 都会形成对 T 的依赖：提交 S 要求由 T 生成的日志记录已被持久化。这在集中式日志记录中很简单，因为所有日志记录都插入唯一的系统范围日志缓冲区，刷新该缓冲区会隐式持久化所有依赖的日志记录 [23]。然而，在去中心化日志记录下，这通常要求刷新多个日志缓冲区。在该示例中，要提交 S，引擎需要刷新服务于两个线程的两个日志缓冲区。跟踪精确依赖（例如，通过维护集中式依赖图）还会进一步造成性能瓶颈 [24]。因此，先前已提出各种近似方法。它们通常涉及维护全局日志序列号（LSN），并且只允许提交LSN低于所有日志中最低持久LSN的事务

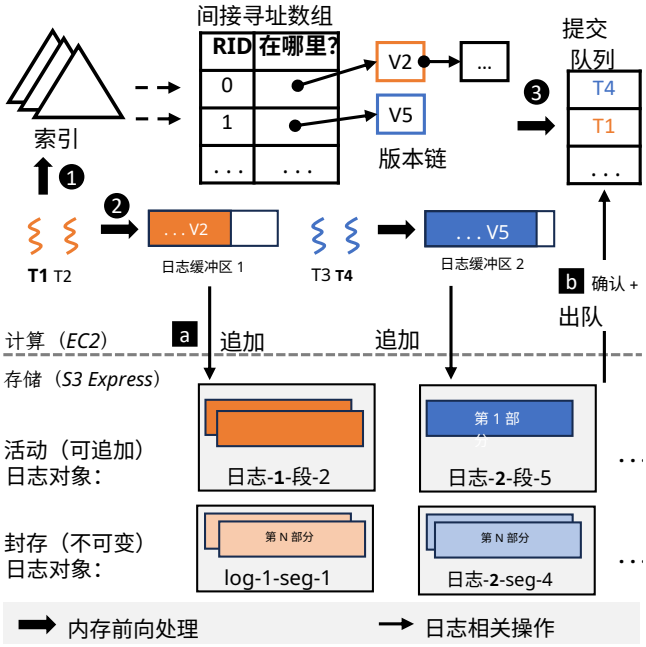


图 3：Milliscale 概览。工作线程通过索引和版本链访问记录。日志被分段为对象，一旦达到追加限制即被封存。为了在保持高吞吐量和低提交延迟的同时减少 S3 请求，线程被分组以共享日志缓冲区。预提交事务留在提交队列中，直到其自身及所依赖的日志记录被持久化。

以提交 [20, 26, 38]。稍后介绍 Milliscale 的解决方案时，我们将详细阐述这些方法存在的问题。

3 设计原则

S3 Express One Zone 的特性带来了以下 Milliscale 设计指南：

- **精简对象操作**。必须仔细校准并规划对象操作（尤其是用于日志记录的 PubObject）的请求大小和数量，以减少数据传输延迟。
- **缩短依赖导致的延迟**。S3 Express One Zone 的原始延迟可低至个位数毫秒。Milliscale 应通过削减不必要的软件导致的延迟，尽量使端到端提交延迟接近这一水平。
- **在保持高吞吐量的同时实现低延迟**。尽管我们的重点是降低（尾部）提交延迟，但仍希望保持先前工作所达到的高吞吐量。

4 MILLISCALE 设计

本节首先概述 Milliscale，然后详细描述其设计。

4.1 概述

Milliscale 是一种联机事务处理引擎，专注于降低基于低延迟可变对象存储的事务提交延迟。与大多数

云原生数据库管理系统一样，Milliscale 将计算与存储分离。事务请求会在计算节点（例如 Amazon EC2 实例）上被接受，该节点使用工作线程访问数据，并将日志记录写入对象存储。我们旨在优化提交路径，因此专注于日志子系统，并采用 ERMIA [27] 中具有代表性的面向内存的优化方案，以实现多版本并发控制、索引和数据组织。通过这种方式，Milliscale 在利用内存优化和对象存储的同时，保持 ACID 特性。

计算层（前向处理）。Milliscale 中的记录通过逻辑记录 ID（RID）标识。同一记录的各个版本按从新到旧的顺序链接成版本链 [40]。每张表由一个间接寻址数组表示，该数组将每个 RID 映射到记录的最新版本。每个版本都带有提交时间戳，即创建该版本的事务的提交序列号（CSN）。CSN 通过原子取值并加一指令从中央计数器中获取 [22]，事务进入预提交阶段时即执行此操作。此外，每个事务都关联一个开始时间戳，该时间戳在事务创建时或读取第一条记录时通过读取全局 CSN 计数器获得。如图 3（上）所示，事务遍历索引和版本链以访问数据，并且在修改数据库时将内容插入日志缓冲区。在去中心化日志记录的基础上，Milliscale 允许工作线程有限地共享日志缓冲区（但不会因此造成瓶颈），并且仅写入每条记录中被修改的字段。这些设计共同减少了 S3 追加请求。

存储层（提交路径）。遵循经典的流水线提交方式，[23]，事务 T 完成前向处理后，工作线程会将其放入提交队列进行预提交 T。随后，线程继续处理下一个请求，同时 T 等待其自身及依赖的日志记录在 S3 中持久化。

Milliscale 使用 S3 Express OneZone 作为数据的主要持久化存储，以利用 S3 的高持久性、可扩展性和低延迟追加操作。重要的是，Milliscale 不依赖高端计算节点上的 NVMe 固态硬盘等中间层，因此可以部署在低成本计算实例上。日志缓冲区会定期写入 S3 Express One Zone，使用

当 S3 追加操作达到容量上限或超时后，就会执行追加。如图 3（下）所示，日志被划分为多个对象，每个对象由日志缓冲区刷新操作所追加的分段组成。在 S3 Express One Zone 中，Milliscale 为每个日志维护一个活动对象，每次日志缓冲区刷新都会追加一个分段。例如，在图 3（下）中，log-1 由 $T1$ 和 $T2$ 共享，目前其活动对象中有两个分段。同时，$T1$ 已将其新版本 $V2$ 插入到 日志缓冲区 1 中，并进行了预提交。当 日志缓冲区 1 填满或发生超时后，就会作为第三个分段追加到当前活动对象（log-1-seg-1）中。活动对象达到追加上限（如第 2.2 节所述，为 10,000 次）后就会封存（变为不可变）。例如，图 3（下）中的 log-1 有一个已封存日志段（log-1-seg-1），而log-2（由 $T3$ 和 $T4$ 共享）有四个已封存日志段（log-2-seg-1 – log-2-seg-4）。每次追加成功后，我们都会检查提交队列，释放自身日志记录及其所依赖的日志记录均已持久化的事务。

总的来说，提交延迟取决于事务处于预提交状态的时长，而这又取决于（1）日志缓冲区刷新操作的大小和频率，以及（2）依赖跟踪延迟。

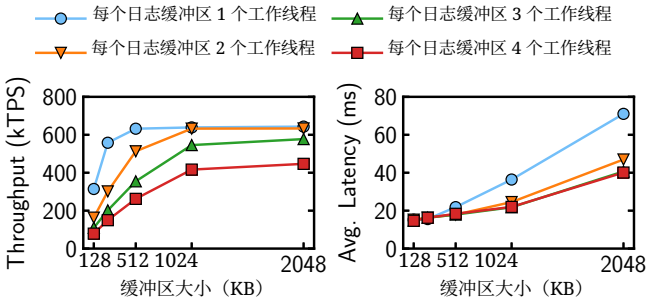


图 4：不同日志缓冲区大小和共享比率下 TPC-C 的吞吐量（左）和平均提交延迟（右）。共享比率为 4:1 时，S3 延迟占主导；而 2:1 时，吞吐量最高、延迟最低。

本节接下来将介绍 Milliscale 在保持高吞吐量的同时，减少这两类延迟的方法。

4.2 受限的去中心化日志记录

传统去中心化日志记录为每个工作线程分配一个日志缓冲区，以彻底消除日志缓冲区争用。然而，这会导致更多的日志缓冲区刷新操作。同一组并行线程完成的相同工作量（产生与使用一个日志缓冲区时相同数量的日志数据）如今被分散到多个日志缓冲区中，而每个缓冲区都需要在提交时持久化。近期的 NVMe 固态硬盘凭借高带宽和低延迟，进一步推动了此类设计的发展 [18]；通常，人们希望尽可能利用每线程日志实现并行化。然而，在对象存储上，情况并非总是如此：使用更多日志缓冲区会增加 S3 追加请求的数量，也会因网络状况而增加拖尾任务，从而降低性能。

按组日志记录。我们发现，关键原因在于现代多核系统中的日志缓冲区数量过多，减少缓冲区数量自然就能缓解这一问题。因此，Milliscale 没有采用传统的按线程日志记录，而是采取折中方案，提出了受限去中心化日志记录，使日志去中心化程度可以调节。受限去中心化日志记录采用按线程组日志记录的方式，将工作线程划分为多个组，并为每组分配一个独立的日志缓冲区。组内线程共享同一个日志缓冲区，并会争用该缓冲区（例如，图 3 中每个日志对应两个线程）。然而，正如第 5 节所示，只要谨慎控制并发度（即组大小），这种争用对性能的影响就微乎其微。换言之，日志分散到一组线程组中，而线程组的大小可根据给定硬件所能容忍的争用程度进行调整。这样一来，我们便能减少系统中的日志缓冲区总数，进而减少 S3 追加操作的次数，以及拖尾任务对性能的影响（即改善尾延迟）。

比例日志缓冲区与线程组扩展。按线程组共享日志缓冲区这一想法虽简单明了，但若要真正发挥效用，就应谨慎选择（1）组大小（即线程与日志共享比率）和（2）日志缓冲区大小，因为这两者共同决定了提交延迟以及向对象存储发送的请求数量。与按线程日志记录相比，日志

缓冲区大小应进行调整，以确保每个工作线程平均分配到的日志缓冲区空间至少与按线程日志记录时的日志缓冲区大小相同。这对于确保刷新操作得到显著减少是必要的，但也意味着需要使用更大的日志缓冲区，同时保持提交延迟与按线程日志记录时相当。然而，日志缓冲区越大（因而共享比率越高），其填充速率就越低（也就是说，填满每个日志缓冲区所需的时间更长）。这会导致流水线组提交下的事务提交延迟增加，而流水线组提交对于实现高吞吐量至关重要（第 2.4 节）。例如，假设事务 T 已预提交，并且是其线程组中第一个写入日志缓冲区的事务。T 随后会进入提交队列，直到日志缓冲区填满或发生超时。在持续繁忙的系统中，通常会出现前一种情况，而 T 会一直在提交队列中等待，直到日志缓冲区填满，并通过一次 S3 追加请求持久化。日志缓冲区越大，T 就越需要等待其他事务完成日志缓冲区的填充，之后 T 才能完全提交（即从提交队列中移除）。

因此，理想的做法是按组大小成比例地增加日志缓冲区大小。例如，当组大小为 2（即两个线程共享一个日志缓冲区）时，日志缓冲区大小应为 2×。

日志缓冲区大小与传输延迟。按组大小成比例地增加日志缓冲区大小，可以减轻日志缓冲区填充速率降低所带来的影响，但由于每个请求现在都更大，可能会加剧日志缓冲区刷新延迟。Milliscale 利用 S3 Express One Zone 的性能特性来解决这一问题。正如我们在第 2.2 节中所述，只有当请求大小超过 1MB 时，完成追加请求的延迟才会开始线性增长。因此，追加一个 512KB 的日志缓冲区和一个 1MB 的日志缓冲区所需的时间相近。因此，当共享比率为 2 时，将日志缓冲区大小设为 1MB，其提交延迟将与使用 512KB 日志缓冲区进行按线程日志记录相当。

图 4 展示了在 TPC-C 下，共享比率与日志缓冲区大小在吞吐量（左）和延迟（右）方面的权衡空间（详细设置见第 5 节）。在按线程日志记录下（即图中每个日志缓冲区对应 1 个工作线程），吞吐量在 512KB 日志缓冲区处趋于平稳。使用更大的日志缓冲区（例如 1MB/2MB）时，共享比率为 2 可保持与按线程日志记录相同的吞吐量，因为

每个线程平均拥有 512KB/1MB 的日志缓冲区空间。然而，共享比率过高（例如 4）会减少每个线程平均可用的日志缓冲区空间，增加争用和刷新频率，最终导致吞吐量下降。图 4（右）展示了相应的平均提交延迟。与图 4（左）不同，较高的共享比率会加快日志缓冲区填充速率，从而降低提交延迟。同时值得注意的是，在缓冲区大小为 1MB 时，共享比率为 2 或 4 的延迟表现与使用 512KB 日志缓冲区且不共享时相近。为兼顾高吞吐量（与按线程日志记录或图中每个日志缓冲区对应 1 个工作线程的情况相同）和低提交延迟，我们因此选择共享比率为 2、日志缓冲区大小为 1MB。正如第 5 节的评估所示，这种共享比率和日志缓冲区大小适用于各种 OLTP 工作负载模式，并且能够显著降低尾延迟。

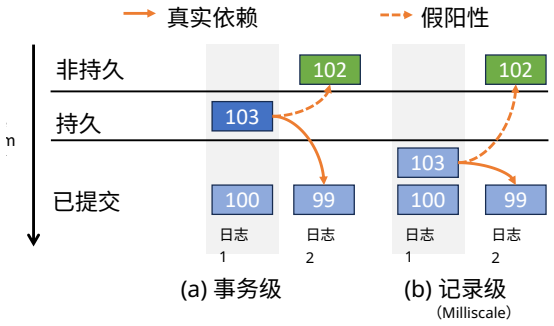


图 5：事务级（a）与记录级（b）依赖跟踪。记录级跟踪使提交序列号为 103 的事务无需等待提交序列号为 102 的事务先行提交，即可提交。

4.3 记录级依赖跟踪

受限去中心化日志记录减少了日志刷新操作和日志数据量，但与集中式日志记录相比，仍会带来额外的软件导致的延迟（第 2.4 节）。要提交一个事务T，必须确保 T 自身的日志记录以及 T 所依赖的所有日志记录都已持久化。这要求跟踪事务之间的依赖关系，并且只完全提交（即通知应用程序）满足这两项要求的事务。**假阳性依赖。**记录详细、准确的依赖信息（例如使用依赖图）需要额外的日志空间和 CPU 周期，因此大多数系统都会对事务间的依赖关系进行近似处理。早期工作 [20, 38] 使用日志序列号和 GSN 来确定事务间的偏序。不过，这些方法是基于页面的，对于访问同一页面上不同记录（因此彼此无依赖关系）的事务，会产生假阳性依赖。较新的方法采用更细粒度的事务级依赖跟踪 [1, 26]，以避免这类假阳性依赖。具体而言，只有当事务的开始时间戳低于所有日志中的最低持久时间戳时，该事务才能提交。然而，我们发现，这仍会导致显著影响提交延迟的假阳性。图 5(a) 展示了一个包含两个日志的示例，每个日志专用于一组线程（在按线程日志记录的情况下，则专用于单个线程）。事务以方框表示，框内标出其开始时间戳（或提交时间戳，具体取决于系统的并发控制协议）；例如，我们将时间戳为 103 的事务称为 T_{103}。此时，所有日志中的最低非持久时间戳为 102，因此任何时间戳低于该值的事务（即 T_{99} 和 T_{100}）都可以提交。同时，T_{103} 已预提交，且其日志记录已完全持久化。然而，由于它在 T102 之后才被序列化（例如，通过稍后执行原子 fetch-add 获取时间戳），尽管 T_{103} 不依赖于 T_{102}，它仍无法在 T_{102} 提交之前提交，从而产生了假阳性依赖。

记录级依赖跟踪。 Milliscale 更进一步，通过更细粒度的记录级依赖跟踪来缓解这一问题。在前向处理期间，每个事务都会维护其所依赖的最近一个直接前驱事务的提交时间戳（称为最大依赖序列

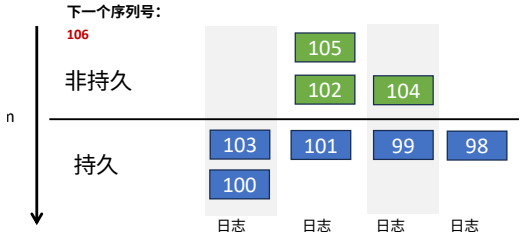


图 6：Milliscale 将最小的非持久 CSN 作为 gCSN。如果某个日志（日志 1 和日志 4）没有非持久事务，则将其非持久序列号视为下一个可能的序列号（106），即 gCSN = min(106, 102, 104, 106) = 102。

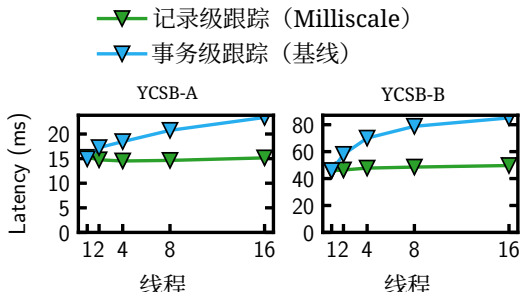
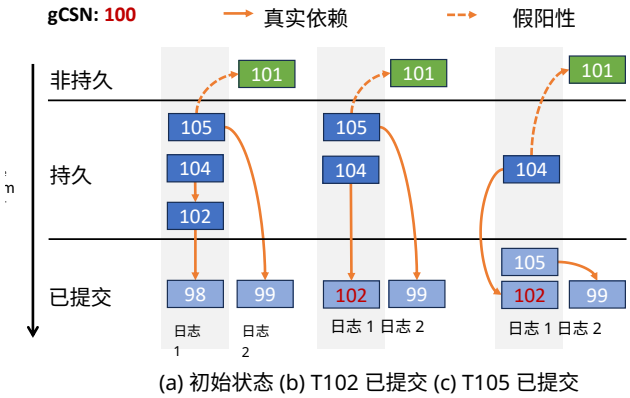


图 7：均匀分布下 YCSB-A（左）和 YCSB-B（右）的平均提交延迟。与事务级跟踪（基线）相比，Milliscale 通过在更细的记录粒度上进行跟踪，避免了更多假阳性依赖。

号，或 DSN）。例如，在图 5(b) 中，如果 T_{103} 读取或更新了由 T_{99} 生成的版本，那么当其当前 DSN 小于 99 时，就会将 DSN 更新为 99。随着日志缓冲区被刷新，存储引擎会根据每个日志的持久和非持久序列号计算系统范围的提交序列号。具体而言，Milliscale 会检查每个日志中的所有非持久序列号，并取其中的最小值作为提交序列号，如图 6 所示。随后，流水线组提交线程即可安全地提交 DSN 小于该提交序列号的任何事务，因为这表明该事务所依赖的所有事务都已提交。与图 5(b) 所示的传统事务级跟踪相比，记录级跟踪通过避免对 T_{103} 的假阳性依赖，使 T_{103} 能够提交。

为每个事务维护一个 DSN 的开销很小。由于它是事务本地的，因此不需要额外的并发控制。在 Milliscale 所基于的多版本并发控制（MVCC）中，每个版本都已携带其创建者的提交时间戳。对恢复的影响也很小，我们将在后文讨论。如图 7 所示，与事务级跟踪相比，在均匀分布下，YCSB-A（50%/50% 读取/更新）和 YCSB-B（95%/5% 读取/更新）事务的平均提交延迟最多可降低超过 35%。我们将在第 5 节进一步探讨记录级依赖跟踪对尾延迟的影响。



(a) 初始状态 (b) T102 已提交 (c) T105 已提交

图8：记录级依赖跟踪的病态情况。(a) 最初，T102、T104 和 T105 均为持久状态，全局提交序列号（gCSN）为 100。(b) T102 提交，因为其最大依赖序列号 [DSN,98] 小于 gCSN。(c) T104 无法提交，因为其 DSN[102] 大于 gCSN；但 T105 可以提交，因为其最新依赖事务 [99]已经提交。

分析与局限性。 记录级跟踪比以往方法能减少更多假阳性，但无法将其完全消除。图 8 说明了当系统从一个初始状态开始时，我们的方法的优点与局限性：T_{99} 和 T_{98} 已提交，提交序列号为 100。在图8(a)中，由于 T_{102} 仅依赖于 T_{98}，其 DSN 为 98，小于提交序列号（100）。这意味着 T_{102} 的所有依赖项都已提交；并且由于 T_{102} 自身的日志记录已持久化，T_{102} 可以安全提交。在图8(b)中，T_{104} 仅依赖于已提交的 T_{102}。但是，已提交序列号小于 102，这意味着提交序列号低于 102 的事务并非全部持久化（例如 T_{101}）。因此，T_{104} 无法提交。如何以较低开销彻底消除此类假阳性依赖，仍有待探索。不过，正如我们在第 5 节后文所示，在广泛的代表性工作负载中，此类病态情况的影响可以忽略不计。

4.4 检查点与恢复

随着日志对象不断累积，通过对内存数据执行检查点并将其写入对象存储来缩短恢复时间是有益的。Milliscale继承了ERMIA的检查点方法 [27]，因为前述受限去中心化日志记录与记录级依赖跟踪技术并不影响检查点的执行方式。为完整起见，我们在此说明其工作原理。对于Milliscale和ERMIA这类内存优化系统，主要目标是存储内存数据的一致性快照。检查点线程首先通过读取中央时间戳计数器获得事务开始时间戳 t，然后扫描每张表的间接寻址数组，保存在 t之前已提交的最新版本。得益于多版本并发控制，执行检查点不会阻塞前向处理，且检查点将表示数据库截至

检查点开始时的状态。随后可通过删除覆盖已检查点数据的S3对象来截断日志。

该恢复协议与常见的仅重做日志设计也类似 [27, 30, 37–39]：多个线程可以并行扫描日志，并竞争安装最新版本。唯一需要进行的小幅调整，是确保在恢复期间保留 DSN 大于全局持久时间戳的事务。例如，在事务级依赖跟踪下，只需重放日志直到图8中的 $T100$。然而，在记录级依赖跟踪下，Milliscale 应确保事务 $T102$ 和 $T105$ 在崩溃后仍然保留。为此，我们在每条日志记录中嵌入事务的 DSN，并重放其依赖事务已提交的所有日志记录（即提交时间戳大于该日志记录的 DSN ）。

最后，大多数针对主存系统的既有仅重做日志设计，都会在每条日志记录中存储整个新版本，包括未更新的字段。这会增加日志数据量，进而导致更多 S3 追加请求。Milliscale 只在日志中存储增量（即已更新的字段）以减少日志数据量，但在内存中保留完整版本，以避免前向处理期间的记录重建开销。正如我们稍后在第 5 节所示，这种优化高度依赖工作负载；在 TPC-C 中，仅存储增量最多可节省三分之二的数据量。

4.5 讨论

受限去中心化日志记录在避免争用（按线程日志记录）与日志缓冲区刷新频率之间取得了平衡。另一种设计是仍为每个线程分配一个专用日志缓冲区，但将这些日志缓冲区分配在连续的内存中，以便通过一次输入/输出操作将其刷新。这样既能保留去中心化日志记录的全部优势，又需要在线程之间进行协调，例如处理各线程填满日志缓冲区的速度不同、从而导致刷新延迟的情况。因此，我们选择在一组线程内进行有限共享。如第 4.2 节所述，日志缓冲区大小至关重要，并根据 S3 Express One Zone 当前的延迟特征进行设置；在运行时自动调整这一大小是一个很有前景的未来方向。

与大多数替代方案相比，Milliscale 能提供更高的可用性。计算本地 SSD 是临时性的，因此必须将数据永久存储在能够经受计算重启的分离式存储服务中，例如 EBS。然而，EBS 的可用性仅限于单个可用区。尽管可以定期对 EBS 中的数据创建快照并存储到其他可用区，但存储层的快照边界可能与数据库管理系统的快照边界不一致，因而仍需由数据库管理系统负责大部分复制工作。此外，与 S3 Standard 相比，S3 Express One Zone 以牺牲高可用性保证为代价，提供了所需的低延迟。因此，Milliscale 之类的系统可能需要手动复制数据，才能达到与 S3 Standard 相同的高可用性水平。Milliscale 允许应用程序提供用于复制的 S3存储桶列表。执行组提交时，我们会向所有存储桶发出多个异步 S3 追加请求，并在全部请求或大多数请求完成后开始提交。

最后，本文重点探讨了如何调整不使用缓冲池的内存优化型 OLTP 引擎，使其能够使用 S3 Express One Zone。不过，这两种技术并不局限于此类引擎，基于缓冲池的系统也可以采用它们 [29, 32] 因为它们的日志子系统大体遵循相同的设计。例如，可以在 LeanStore [29] 的分布式日志记录设计之上采用受限去中心化日志记录。我们将这些探索留待未来工作。

5 评估

此前，我们已通过一组微基准测试展示了受限去中心化日志记录和记录级依赖跟踪的效果。现在，我们将使用更多基线和基准测试来扩展对 Milliscale 的评估，以展示以下几点：

- 在各种基准测试中，Milliscale 在延迟和吞吐量方面都能轻松胜过 S3 Standard。
- 与使用块存储（如 EBS）的基线相比，Milliscale 能够实现相当甚至更低的平均提交延迟和尾延迟，同时保持高吞吐量。
- 事实证明，Milliscale 中的技术对其他存储后端也同样有用，包括本地部署和云块存储。

5.1 实验设置

硬件。我们在一台专用的 Amazon EC2 c6in.8xlarge 实例上，针对多种存储后端进行了实验。该实例配备了主频为 2.9GHz（开启睿频后最高可达 3.5GHz）的英特尔至强铂金 8375C CPU，以及 64GB 内存。虽然 CPU 本身有 32 个核心（64 个超线程），但该实例通过 16 个物理核心提供 32 个虚拟 CPU（超线程）。

除 S3 Standard 和 S3 Express One Zone 外，我们还使用 EBS gp3 和 io2 作为基线。gp3 和 io2 均配置为 8,000 IOPS。gp3 卷的带宽最高为 2,000Mbps，而 io2 卷的带宽不受限制。该实例支持 50Gbps 的网络带宽和 25Gbps 的 EBS 带宽，并且不配备本地 NVMe SSD，以降低总体拥有成本，因此所有持久化数据都写入 EBS 或 S3。

软件。 服务器运行 Ubuntu 24.04.2 LTS，所有代码均使用 GCC 13 并开启全部优化进行编译。我们的实现基于 AWS SDK for C++ （版本 1.11.676） [6]。原版 AWS SDK 提供了 S3 追加操作的异步接口，但它通过应用程序无法感知的线程池（即 Milliscale）来实现，这会增加 CPU 调度开销。因此，我们自行实现了异步 S3 操作，在内部创建由自身管理的刷新线程，以调用同步的 AWS S3 追加接口。

Milliscale 中的每个工作线程组都分配有一个刷新线程。我们发现它们仅占用 3–5% 的 CPU 周期，因此在预算受限的场景下，可以将它们固定到同一个超线程上。为便于解读结果，我们在 16 个工作线程下运行实验，每个线程都固定到不同的核心上。每个刷新线程都固定到与对应工作线程组所用核心相同的核心上的不同超线程。对于使用块存储（gp3 或 io2）和每线程日志的基线方案，我们使用 io_uring 发起异步 I/O 操作，并为每个日志（工作线程）配置独立的刷新线程，以跟上填充速率并维持高吞吐量。

实验变体。我们在改进版 ERMIA [27] 之上实现了所有变体，将其原有的集中式日志记录替换为带流水线提交的每线程去中心化日志记录（第 2.4 节）、事务级依赖跟踪以及

基于增量的日志（第 4.4 节），以及（2）使用双缓冲进行日志刷写，使其与前向处理重叠，这在先前工作中很常见。随后，我们更改存储后端和提交协议，以构建以下实验变体：

• S3 Standard：将日志记录直接持久化到 S3 Standard 的基线，这支持记录级依赖跟踪。为获得最佳平均延迟，它仍采用每线程专用日志记录，因为在 S3 Standard 这样速度较慢的存储后端上，受限去中心化日志记录无法改善尾延迟。

• S3X：将日志记录直接持久化到 S3 Express One Zone 的基线，不采用 Milliscale 提出的优化（即直接使用 S3 Express One Zone）。

• gp3：与 S3X 相比，它使用 gp3 而非 S3 Express One Zone 作为存储后端。• io2：与 S3X 相比，它使用 io2 作为存储后端。

• Milliscale：与 S3X 相同，但同时采用受限去中心化日志记录和记录级依赖跟踪。

与先前的 OLTP 引擎工作相同， [27, 29, 32, 33, 37], 我们将 Milliscale 和各基线实现为共享库。应用程序（如各种基准测试）直接通过引擎的 C++ API 访问数据库记录，不经过 SQL 和优化器层。所有变体都采用 ERMIA 中相同的并发控制协议，并支持读已提交、快照隔离和可串行性。隔离级别的选择与我们的评估无关，因此所有实验均使用快照隔离。

根据第 2.2 节的观察结果，对于采用按线程日志记录的变体，我们使用 512KB 的日志缓冲区。对于 Milliscale，我们将日志组大小设为 2。根据第 4.2 节的分析，由于每个组由两个线程组成，我们将日志缓冲区设为 1MB（由两个线程共享），这样平均而言，每个线程获得的空间与按线程日志记录时相同。

基准测试。 我们采用基于 YCSB [11] 的微基准测试和端到端 TPC-C 基准测试 [36]。对于 YCSB，我们使用包含 3000 万条记录的数据库表。每条记录为 80 字节，包含 10 个 8 字节字段，其中一个是主键。每个事务按照均匀分布或偏斜分布（齐夫分布，theta=0.99）访问 10 条记录，通过基于表的主键列构建的 Masstree [31] 索引进行点读或更新（读改写）；更新操作会读取选定记录的全部内容，并修改其中一个随机字段。我们测试两种事务混合配置：YCSB-A 和 YCSB-B，分别执行 50%/50% 和 95%/5% 的读取/更新事务。对于 TPC-C，我们使用包含 100 个仓库的数据库，并让每个事务随机选择一个所属仓库进行访问。为专注于日志记录，我们将所有表数据保留在内存中，因此所有发往存储后端的流量都由刷新日志缓冲区产生。

我们报告所有工作负载的吞吐量、平均提交延迟和尾提交延迟。请注意，对于延迟，我们只测量前向处理后的提交延迟。这样可以公平地比较所有事务类型的提交延迟。每次实验运行 10秒，并重复三次；各次运行之间的方差极小，因此我们报告平均值。

5.2 微基准测试可扩展性

我们的第一组实验评估了在 YCSB-A 和 YCSB-B 微基准测试下的吞吐量和提交延迟。

吞吐量。 如图9（上）所示，在均匀分布下，除 S3 Standard 外的所有变体都随着线程数的增加而良好扩展，这得益于流水线提交和双缓冲能够隐藏存储延迟。在偏斜分布下，图 9（上）中也呈现出相同趋势。S3 Standard 的高延迟远超工作负载的日志缓冲区填充速率，使其无法扩展。与 YCSB-A 相比，YCSB-B 包含更多读操作，由于更新操作更少（因而日志流量更轻），导致日志缓冲区填充速率更低。因此，双缓冲能够隐藏大部分日志刷新延迟，并带来图10（上）所示的更高吞吐量。

平均延迟。 接下来，我们探讨平均提交延迟如何随线程数变化；稍后再讨论尾延迟。如图9（下）所示，在均匀分布下，Milliscale 在不同线程数下的平均延迟几乎保持不变（与单线程情况相近）。在16个线程下，Milliscale 的 YCSB-A 平均延迟比 S3X 低28.4%；图10（下）中 YCSB-B 的这一数值则低 42%。这些改进主要得益于 Milliscale 的记录级依赖跟踪，其粒度比以往方法更细。请注意，受限去中心化日志记录可能会略微增加延迟，因为多个线程会共享同一个日志缓冲区。与默认的2:1共享比率相比，采用1:1比率可使 YCSB-A 在16个线程下的延迟降低35%。不过，正如我们稍后将看到的，受限去中心化日志记录可以更显著地降低尾延迟。YCSB-B 从记录级依赖跟踪中获益更多：该工作负载本身写操作较少，而且在均匀分布下，写事务发生冲突的频率较低，因此真实依赖更少，Milliscale 能够有效处理这些依赖。

齐夫分布下的偏斜工作负载会导致事务之间出现更多依赖关系，尤其是真实依赖。不过，记录级跟踪最能有效避免事务因虚假依赖而阻塞提交。因此，在图9（右）中的 YCSB-A 工作负载下，Milliscale 在所有线程数下的平均延迟都与 S3X 接近。图10（右）中的 YCSB-B 也呈现出类似结果，但由于写操作仅占 5%，S3X 与 EBS变体之间的差距微乎其微。

5.3 尾延迟

在本实验中，我们将线程数固定为 16，并在均匀分布和齐夫分布下运行 YCSB-A 和 YCSB-B 工作负载。图 11 展示了 YCSB-A 在不同百分位数下的提交延迟。在所有百分位数下，S3 Standard 和 io2 的尾延迟分别最高和最低。它们代表了两个极端：价格低但速度慢，以及速度快但价格高。其他变体则试图在这两个极端之间取得平衡。在均匀分布下，Milliscale 和 gp3 的尾延迟特征相近，但在偏斜访问下，前者的延迟开始高于 gp3。我们将其归因于偏斜工作负载下存在更多真实依赖。尽管如此，在均匀分布和齐夫分布下，Milliscale 仍将 99.9 百分位延迟分别降低了 62.4% 和 51.8%，相比之下，S3X 的表现较差。99.99 百分位尾延迟的降幅分别为 56.2% 和 51.6%。如图 12 所示，运行 YCSB-B 的结果也呈现出类似趋势，因此这里不再赘述。

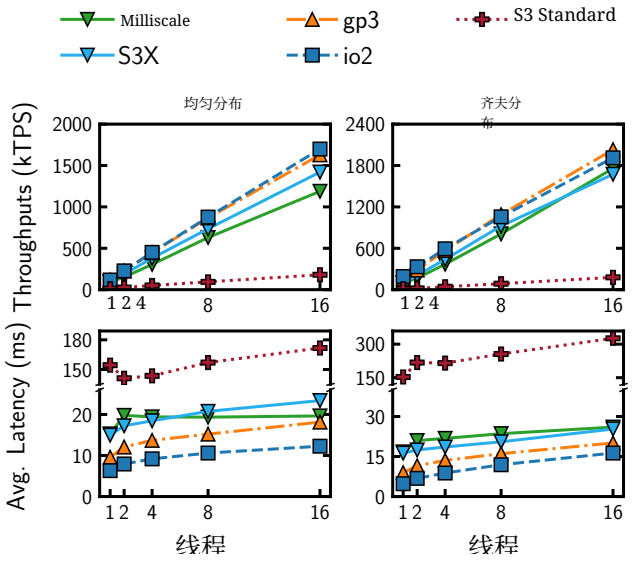


图9：均匀分布（左）和齐夫分布（右）下 YCSB-A（50% 读取、50% 写入）的吞吐量和平均延迟。除 S3Standardscale 外，Milliscale 的吞吐量与基线相当，并实现了与使用块存储后端相近的延迟。

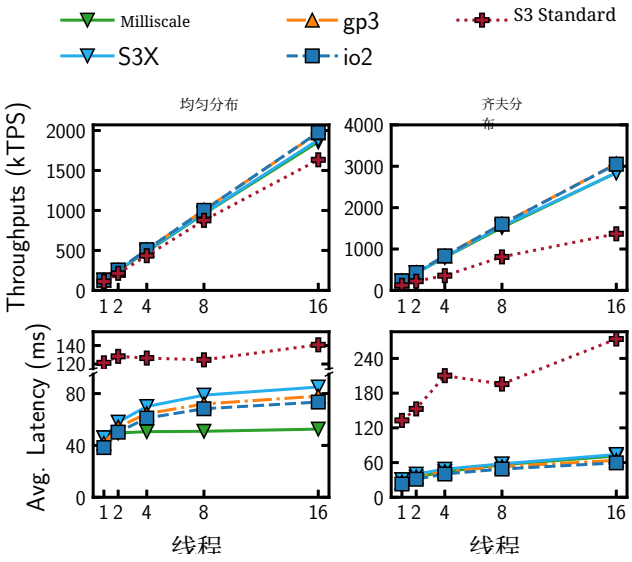


图10：均匀分布（左）和齐夫分布（右）下 YCSB-B（95% 读取、5% 更新）的吞吐量和平均延迟。由于（内存中的）读取操作更多，Milliscale 凭借记录级依赖跟踪，实现了比块存储更低的延迟。

5.4 消融研究

第 4 节分析了受限去中心化日志记录和记录级依赖跟踪对平均提交

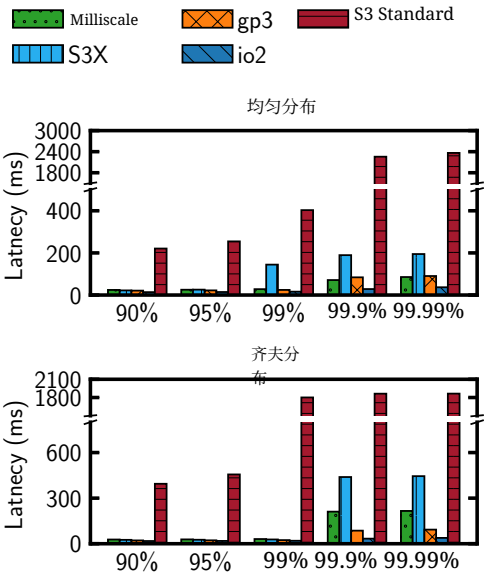


图 11：16 个线程下 YCSB-A 的尾延迟。在均匀分布下，Milliscale 的延迟与 gp3 的延迟相当。偏斜工作负载（底部）呈现出更多真实依赖，导致 Milliscale 的延迟增加；不过，在 99.9% 和 99.99% 分位点处，其延迟仍降低了超过 50%。

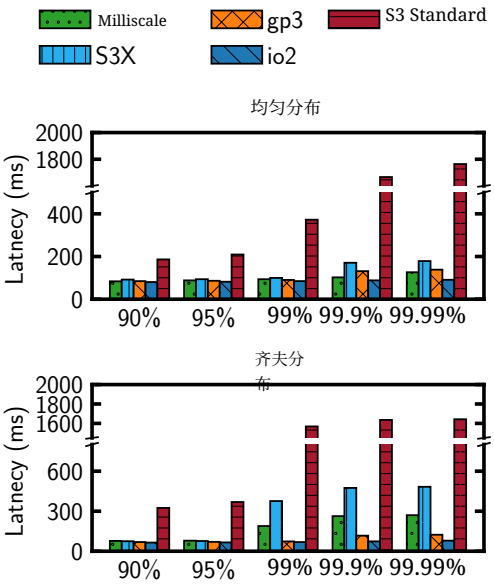


图 12：16 个线程下 YCSB-B 在均匀分布（顶部）和偏斜（底部）访问模式下的尾延迟。

延迟。现在，我们使用 YCSB-A 将评估扩展到尾延迟。我们先评估 S3X，然后分别在不同百分位数下评估每种技术。结果如图13所示。所有变体在 95 百分位数之前都表现出相当低的延迟。正如预期，受限去中心化日志记录（图中的 S3X + RDL）表现出

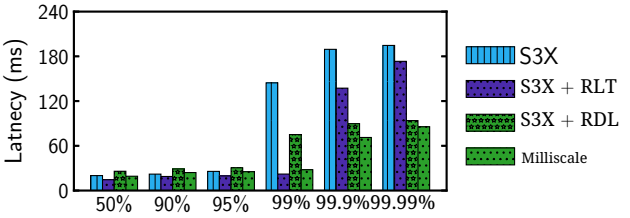


图 13：在 16 个线程和 YCSB-A 下，记录级依赖跟踪（S3X +RLT）、受限去中心化日志记录（S3X + RDL）及其组合效果（Milliscale）的影响。

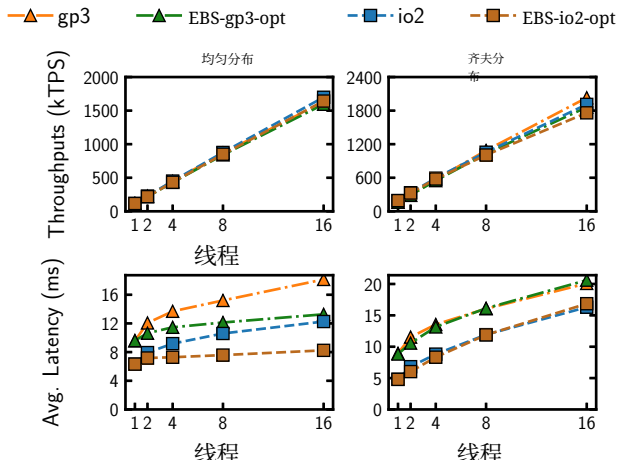


图 14：使用 EBS gp3/io2 时 YCSB-A 的吞吐量与平均延迟。Milliscale 优化（*-opt）主要对块存储上的均匀分布有用。

在 50 和 99 百分位数下，其尾延迟略高于记录级依赖跟踪（S3X + RLT），这是因为同一组中共享日志缓冲区的线程导致日志缓冲区争用略有增加。从 99 百分位数开始，S3X 的延迟显著增长，增幅超过 5×。记录级依赖跟踪将这一趋势推迟到了 99 百分位数，并且在 99.9 和 99.99 百分位数下仍然有效。在 99.9% 和 99.99% 的情况下，受限去中心化日志记录比记录级依赖跟踪更有效。我们发现，这是因为日志缓冲区较少时，工作负载本身产生的、必须在提交前解决的跨日志依赖也较少（例如，$T105$，如图8所示）。

5.5 对块存储的影响

如前所述，Milliscale 中的实用技术并不局限于对象存储，也可应用于 EBS，甚至本地部署的固态硬盘。我们在 gp3 和 io2 上应用了受限去中心化日志记录（共享比率同为 2:1）和记录级依赖跟踪。图 14 和图 15 分别展示了 YCSB-A 下的吞吐量/平均延迟和尾延迟表现。借助 Milliscale 的优化，在 16 个线程下，gp3 和 io2 的平均延迟分别降低了 26.8% 和 32.8%。99.99 百分位延迟的降幅分别为 42.2% 和 41.6%。与之前的结果类似，在偏斜齐夫分布下

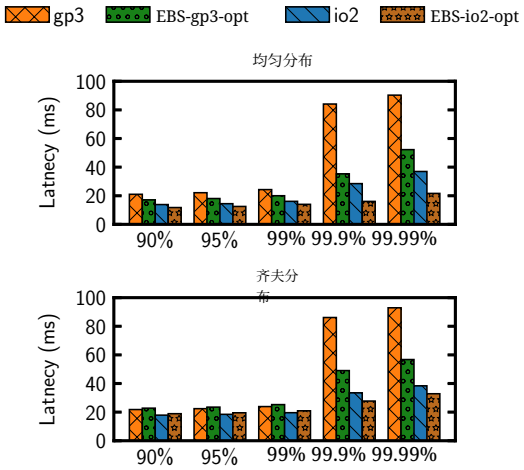


图 15：块存储后端上 YCSB-A（16 个线程）在有无 Milliscale 优化时的尾延迟。

（此时事务之间的依赖更多），平均延迟和吞吐量没有明显下降；在 16 个线程下，gp3 的尾延迟降低了 39.0%，io2 则降低了 14.4%。

总体而言，这些结果表明，Milliscale 技术对更快的存储后端有用，但对于延迟更高的同类存储（如 S3 Express One Zone）更为必要且更有效。

5.6 端到端 TPC-C 结果

我们的最后一项实验使用端到端 TPC-C 测试 Milliscale 的性能。如前所述，我们使用包含 100 个仓库的数据库，并让每个工作线程随机选择其所属仓库。这样可以通过增加跨日志依赖来对系统施加更大压力，并加剧事务之间的争用。在 2:1 的共享比率下，图 16 显示，在不同规模下，Milliscale 中的受限去中心化日志记录将 S3 追加请求的数量减少了 ~50%。如图 17 所示，在吞吐量和延迟方面，Milliscale 的表现都显著优于基线 S3 Standard。在 16 个线程下，Milliscale 的平均延迟比 S3X 高 3.8%，比 gp3 高 23%。在第 99 百分位数处，S3X 和 Milliscale 的延迟都远高于 gp3 和 io2；我们认为这是由各存储后端的性能特性差异造成的：S3 Express One Zone 的尾延迟从第 99 百分位数开始激增，而 gp3 和 io2 则从第 99.9 百分位数开始激增。尽管如此，如图 18 所示，与此前的微基准测试结果类似，Milliscale 能有效降低相较于 S3X 的延迟，并在第 99.9 和第 99.99 百分位数处达到与 gp3 相近的延迟。

6 相关工作

我们的工作与先前利用对象存储支持数据库管理系统并优化预写日志的研究最为相关。

对象存储上的数据库管理系统。 据我们所知，最早尝试将 S3 用于数据库管理系统的是 Brantner 等人 [10]。该系统基于 S3 Standard，必须应对 S3 的诸多缺陷，尤其是弱一致性、缺乏追加支持以及极高的延迟（100 毫秒级）。这些导致

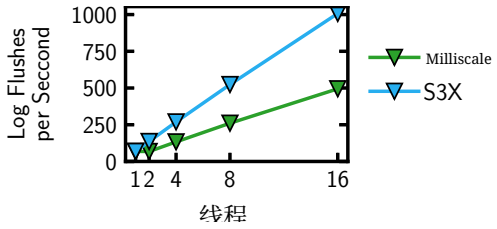


图 16：在 TPC-C 工作负载下，采用受限去中心化日志记录且共享比率为 2:1（两个线程共享 1MB 日志缓冲区）时，每秒 S3 追加请求数与按线程日志记录（每个线程使用 512KB 缓冲区）的对比。

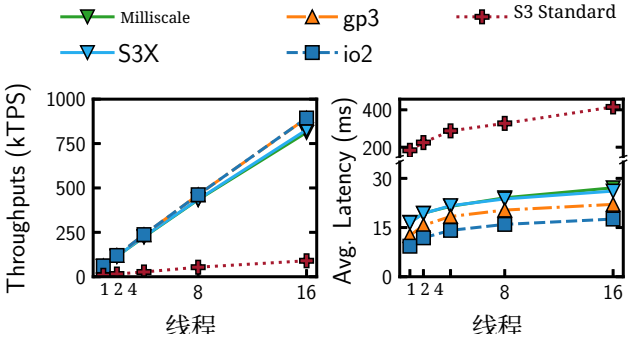


图 17：TPC-C 吞吐量和平均延迟。

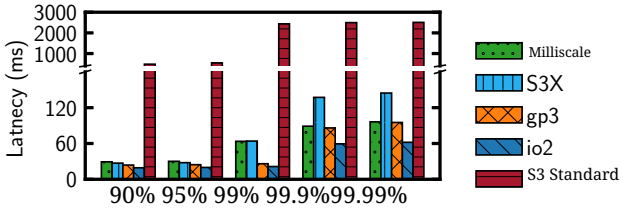


图 18：16 个线程下 TPC-C 的尾延迟。

并非完全符合 ACID 的解决方案，并且日志记录被卸载到单独的消息队列实现中。较高的端到端事务延迟（秒级）使这些方法受到冷落。如第 2 节所述，S3 Express One Zone 并未在很大程度上解决这些问题（强一致性和追加支持）或缓解这些问题（延迟），而这使我们得以构建 Milliscale。Delta Lake [8] 面向数据湖应用，在 S3 Standard 上管理事务。为弥合内存与 S3 Standard 之间的速度差距，它依靠计算侧 SSD 进行大量缓存。该领域的其他方法大多聚焦于 S3 Standard 上的 OLAP 或混合工作负载。Colibri [35] 提出一种面向联机事务处理和 OLAP 工作负载的混合架构，以利用云存储并分离热页和冷页。AnyBlob [15] 提出一种基于 io_uring 的下载管理器，可异步处理并发的 S3 GET 请求，使数据库管理系统能够在读密集型 OLAP 工作负载中充分利用高速数据中心网络，同时将 CPU 开销降至最低。LiquidCache [19] 在计算节点与对象存储之间部署缓存服务器。虽然数据存储在对象以 Parquet 文件的形式存储数据时，缓存服务器采用专为快速过滤条件求值优化的自定义内存格式。PushdownDB [43] 使用（现已停止服务的）S3Select [16] 功能，将过滤、投影和聚合操作直接下推至 S3 存储节点，从而减少 OLAP 工作负载下存储与计算之间的网络流量。FlexPushdownDB [42] 在 PushdownDB 的基础上扩展出一种混合架构，以细粒度方式结合缓存与计算下推。与先前工作相比，Milliscale 面向不同的存储后端（S3 Express One Zone），并专注于优化预写式日志，以降低提交延迟。

日志优化。 Milliscale 建立在若干重要的可扩展日志记录先前研究成果之上。Aether [23] 评估并展示了提前释放锁和日志刷写流水线的潜力，使后续系统能够在不牺牲正确性的前提下，将事务吞吐量提升至与异步提交相同的水平。然而，集中式日志记录固有的瓶颈仍然要求采用去中心化日志记录 [24, 41, 45], ，后者后来成为内存优化型 OLTP 引擎中的标准设计选择 [29, 30, 32, 37]。传统去中心化日志记录的一大缺点是以牺牲事务提交延迟为代价换取吞吐量。跨日志依赖是造成这一问题的主要原因，先前已有多种方法被提出，以减轻其影响。例如，如果事务不依赖其他日志，LeanStore 会采用远程刷新规避 [20]，跳过不必要的日志缓冲区刷新。自主提交 [33] 会在工作线程之间并行执行小型日志刷新，并主动插入虚拟事务以推进 GSN，从而减轻拖尾任务的影响。Border-Collie [26] 提出了无等待算法，可高效找到事务安全提交所需的逻辑时钟位置上界。ELEDA [25]会填补各日志中 LSN 的间隙，使全局持久化 LSN 标记能够及时推进。与这些方法相比，Milliscale 以更细粒度的记录粒度跟踪并解决依赖，同时保持较低开销。

7 总结

我们探讨了利用以 Amazon S3 Express One Zone 为代表的近期低延迟可变对象存储服务进行 OLTP 预写式日志记录的可行性。虽然传统去中心化日志记录方法能够提供高吞吐量，但在 S3 Express One Zone 上它们会带来较高的事务提交延迟（尤其是尾延迟）。基于 S3 Express One Zone 的性能特征，我们提出了 Milliscale，一种内存优化OLTP引擎，它使用 S3 Express One Zone 作为预写日志存储，以优化提交延迟。Milliscale 包含两项关键技术——受限去中心化日志记录和记录级依赖跟踪——它们分别在不增加日志数据传输时间的情况下降低日志刷新频率，并避免先前工作中常见的假阳性依赖。这些技术也可应用于其他系统或存储后端。使用微基准测试和端到端 TPC-C 的评估表明，Milliscale 能够有效降低平均延迟和尾延迟，同时保持高吞吐量。

参考文献

[1] Adnan Alhomssi、Michael Haubenschild 和 Viktor Leis。2023。《LeanStore 的演进》。载于*BTW 2023*。Gesellschaft für Informatik eV，259–281。

[2] 亚马逊云科技。2025。Amazon EBS 卷类型。https://aws.amazon.com/ebs/volume-types/ [3] 亚马逊云科技。2025。Amazon S3 Express One Zone 存储类别。https: //aws.amazon.com/s3/storage-classes/express-one-zone/ [4] 亚马逊云科技。2025。Amazon S3 定价。https://aws.amazon.com/s3/ pricing/ 访问日期：2025-10-24。[5] 亚马逊云科技。2025。Amazon S3 存储类别。https://aws.amazon.com/s3/storage-classes/ [6] 亚马逊云科技。2025。AWS SDK for C++ 版本 1.11.676。https: //github.com/aws/aws-sdk-cpp/releases/tag/ 1 11 676 。[8] Panagiotis Antonopoulos、[7] Alex Budovski、Cristian Diaconu、Alejandro Hernandez Saenz、Jack Hu、Hanuma Kodavalla、Donald Kossmann、Sandeep Lingam、Umar Farooq Minhas、Naveen Prakash 等。2019。Socrates：云中的全新 SQL Server。载于《2019 年国际数据管理会议论文集》。ACM，1743–1756。[8] Michael Armbrust、Tathagata Das、Liwen Sun、Burak Yavuz、Shixiong Zhu、Mukul Murthy、Joseph Torres、Herman van Hovell、Adrian Ionescu、Alicja Łuszczak、Michał undefinedwitakowski、Michał Szafrański、Xiao Li、Takuya Ueshin、Mostafa Mokhtar、Peter Boncz、Ali Ghodsi、Sameer Paranjpye、Pieter Senster、Reynold Xin 和 Matei Zaharia。2020。Delta Lake：云对象存储上的高性能 ACID 表存储。《VLDB 会议录》13，12（2020 年 8 月），3411–3424。[9] AWS。2026。使用 Multi-Attach 将 EBS 卷挂载到多个 EC2 实例。https://docs.aws.amazon.com/ebs/latest/userguide/ebs-volumes-multi.html[10] Matthias Brantner、Daniela Florescu、David Graf、Donald Kossmann 和 Tim Kraska。2008。在 S3 上构建数据库。载于《2008 年 ACM SIGMOD 国际数据管理会议论文集》（SIGMOD '08）。251–264。[11] Brian F Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。2010。使用 YCSB 对云服务系统进行基准测试。载于《第 1 届 ACM 云计算研讨会论文集》。143–154。[12] Benoit Dageville、Thierry Cruanes、Marcin Zukowski、Vadim Antonov、Artin Avanes、Jon Bock、Jonathan Claybaugh、Daniel Engovatov、Martin Hentschel、Jiansheng Huang、Allison W. Lee、Ashish Motivala、Abdul Q. Munir、Steven Pelley、Peter Povinec、Greg Rahn、Spyridon Triantafyllis 和 Philipp Unterbrunner。2016。Snowflake 弹性数据仓库。载于《2016 年国际数据管理会议论文集》（SIGMOD '16）。215–226。[13] Alex Depoutovitch、Chong Chen、Jin Chen、Paul Larson、Shu Lin、Jack Ng、Wenlin Cui、Qiang Liu、Wei Huang、Yong Xiao 和 Yongjun He。2020。Taurus 数据库：如何在云中实现快速、高可用且节省成本。（2020），1463–1478。[14] DuckDB 文档。2025。S3 API 支持。https://duckdb.org/docs/stable/ core_extensions/httpfs/s3api [15] Dominik Durner、Viktor Leis 和 Thomas Neumann。2023。利用云对象存储实现高性能分析。《VLDB 会议录》16，11（2023 年 7 月），2769–2782。[16] Arushi Garg 和 Preethi Raajaratnam。2024。如何优化对 Amazon S3 中数据的查询。https://aws.amazon.com/blogs/storage/how-to-optimize- 3 querying-your-data-in-amazon-s[17] 2025 Google。。Google Cloud 文档——存储类别。https://docs.cloud.google.com/storage/docs/storage-classes?_gl=1*18hxa54*_ up*MQ..&gclid=Cj0KCQiAo4TKBhDRARIsAGW29beKlmBi5jG3PCEP _ 0 8 3 0 5 5 9 0 qzQMj FiGT vSd LO lRy -IyTrRk HiRgqx aAt_cEALw_wcB&gclsrc=aw. ds#standard[18] Gabriel Haas 和 Viktor Leis。2023。现代 NVMe 存储能做什么，以及如何充分利用它：面向高性能存储引擎的高性能输入/输出。《VLDB 会议录》16，9（2023 年 5 月），2090–2102。[19] Xiangpeng Hao、Andrew Lamb、Yibo Wu、Andrea Arpaci-Dusseau 和 Remzi Arpaci-Dusseau。2025。LiquidCache：面向云原生数据分析的高效下推缓存。《VLDB 会议录》16（2025）。[20] Michael Haubenschild、Caetano Sauer、Thomas Neumann 和 Viktor Leis。2020。重新思考高性能存储引擎的日志记录、检查点与恢复。载于《2020 年 ACM SIGMOD 国际数据管理会议论文集》（SIGMOD '20）。877–892。[21] IEEE 和 The Open Group。2016。《The Open Group 基本规范第 7 版，IEEE 1003.1 标准》。（2016）。[22] 英特尔公司。2016。Intel 64 和 IA-32 架构软件开发人员手册。（2016 年 10 月）。[23] Ryan Johnson、Ippokratis Pandis、Radu Stoica、Manos Athanassoulis 和 Anastasia Ailamaki。2010。Aether：一种可扩展的日志记录方法。《VLDB 会议录》3，1（2010 年 9 月），681–692。[24] Ryan Johnson、Ippokratis Pandis、Radu Stoica、Manos Athanassoulis 和 Anastasia Ailamaki。2012。多核与多插槽硬件上的预写日志可扩展性。《VLDB 杂志》21，2（2012 年 4 月），239–263。[25] Hyungsoo Jung、Hyuck Han和Sooyong Kang。2017年。面向多核的可扩展数据库日志记录。Proc. VLDB Endow. 11, 2（2017年10月），135–148。[26] Jongbin Kim、Hyeongwon Jang、Seohui Son、Hyuck Han、Sooyong Kang和Hyungsoo Jung。2019年。Border-Collie：面向多核硬件的无等待、读取最优数据库日志记录算法。见2019年国际数据管理会议论文集（SIGMOD '19）。723–740。[27] Kangnyeon Kim、Tianzheng Wang、Ryan Johnson和Ippokratis Pandis。2016年。ERMIA：面向异构工作负载的快速内存优化数据库系统。见2016年国际数据管理会议论文集。1675–1687。[28] Andrew Lamb、Yijie Shen、Daniël Heres、Jayjeet Chakraborty、Mehmet Ozan Kabak、Liang-Chi Hsieh和Chao Sun。2024年。Apache Arrow DataFusion：快速、可嵌入的模块化分析查询引擎。见2024年国际数据管理会议配套论文集（圣地亚哥AA，智利）（SIGMOD ' 24 5 17 10 1145 3626246 3653368 ）。– 。[29] https://doi.org/ . / 。Viktor Leis、Michael Haubenschild、Alfons Kemper和Thomas Neumann。2018年。LeanStore：超越主存的内存数据管理。见2018年IEEE第34届国际数据工程会议（ICDE）（IEEE ICDE）。185–196。[30] Hyeontaek Lim、Michael Kaminsky和David G. Andersen。2017年。Cicada：可靠快速的多核内存事务。见2017年ACM国际数据管理会议论文集（SIGMOD '17）。21–35。[31] Yandong Mao、Eddie Kohler和Robert Tappan Morris。2012年。面向快速多核键值存储的缓存优化。见第7届ACM欧洲计算机系统会议论文集。183–196。[32] Thomas Neumann和Michael J Freitag。2020年。Umbra：具备内存性能的基于磁盘的系统。见第10届创新数据系统研究会议，CIDR 2020，荷兰阿姆斯特丹，2020年1月12日至15日，在线论文集。[33] Lam-Duy Nguyen、Adnan Alhomssi、Tobias Ziegler和Viktor Leis。2025年。告别组提交：自主提交在NVMe 固态硬盘上实现高吞吐量和低延迟。Proc. ACM Manag. Data 3, 3，文章编号191（2025年6月），24页。[34] Raghu Ramakrishnan和Johannes Gehrke。2003年。数据库管理系统（第3版）。[35] Tobias Schmidt、Dominik Durner、Viktor Leis和Thomas Neumann。2024年。一石二鸟：为混合事务分析处理设计混合云存储引擎。Proc. VLDB Endow. 17, 11（2024年7月），3290–3303。https://doi.org/10.14778/3681954. 3682001[36] TPC。2010年。TPC-C基准测试（联机事务处理）标准规范，修订版5.11。http://www.tpc.org/tpcc [37] Stephen Tu、Wenting Zheng、Eddie Kohler、Barbara Liskov和Samuel Madden。2013年。多核内存数据库中的快速事务。见第24届ACM操作系统原理研讨会论文集（SOSP '13）。18–32。[38] Tianzheng Wang和Ryan Johnson。2014年。借助新兴非易失性内存实现可扩展日志记录。Proc. VLDB Endow. 7, 10（2014年6月），865–876。[39] Tianzheng Wang、Ryan Johnson和Ippokratis Pandis。2017年。Query Fresh：强化版日志传输。Proc. VLDB Endow. 11, 4（2017年12月），406–419。[40] Yingjun Wu、Joy Arulraj、Jiexi Lin、Ran Xian和Andrew Pavlo。2017年。内存多版本并发控制的实证评估。Proc. VLDB Endow. 10, 7（2017年3月），781–792。[41] Yu Xia、Xiangyao Yu、Andrew Pavlo和Srinivas Devadas。2020年。Taurus：面向内存数据库管理系统的轻量级并行日志记录。Proc. VLDB Endow. 14, 2（2020年10月），189–201。[42] Yifei Yang、Matt Youill、Matthew Woicik、Yizhou Liu、Xiangyao Yu、Marco Serafini、Ashraf Aboulnaga和Michael Stonebraker。2021年。FlexPushdownDB：云数据库管理系统中的混合下推与缓存。Proc. VLDB Endow. 14, 11（2021年7月），2101–2113。[43] Xiangyao Yu、Matt Youill、Matthew Woicik、Abdurrahman Ghanem、Marco Serafini、Ashraf Aboulnaga和Michael Stonebraker。2020年。PushdownDB：利用S3计算加速数据库管理系统。见2020年IEEE第36届国际数据工程会议（ICDE）。1802–1805。[44] Matei Zaharia。2025年。借助Lakebase融合事务处理与分析世界。Proc. VLDB Endow. 18, 12（2025年8月），5539。[45] Wenting Zheng、Stephen Tu、Eddie Kohler和Barbara Liskov。2014年。借助多核并行实现快速持久性与恢复的快速数据库。见第11届USENIX操作系统设计与实现会议论文集（科罗拉多州布鲁姆菲尔德）（OSDI'14）。465–477。