构建基于 OceanBase 的分布式近实时分析处理数据库系统

许全清、杨传辉*、李瑞杰、谢东东、曹辉、肖毅、陈俊全、王彦佐、赵赛通、韩福生、刘斌、王国平、赵宇中、庄明强*OceanBase*，蚂蚁集团OceanBaseLabs@service.oceanbase.com

arXiv:2602.07584v1 [cs.DB] 7 Feb 2026

摘要——**现代数据基础设施亟需能够高效管理海量数据集，同时提供实时事务处理和先进分析能力的数据库系统。传统OLAP系统往往无法满足这双重需求，而新兴的实时分析处理系统仍面临持续挑战，例如数据冗余过多、跨系统同步复杂以及时间效率不佳。本文介绍了 OceanBase Mercury，这是一种面向拍字节级数据的创新型 OLAP 系统。该系统采用分布式多租户架构，确保持续可用性和弹性扩展性等企业级核心要求。我们的技术贡献包括三个关键组件：(1) 采用混合数据布局优化的自适应列式存储格式；(2) 用于物化视图的差分刷新机制，并提供时间一致性保证；(3) 支持三种不同数据格式的多态向量化引擎。在真实工作负载下的实证评估表明，OceanBase Mercury 的查询延迟比专用 OLAP引擎快 1.3× 到 3.1× 倍，同时保持亚秒级延迟，使其成为一种突破性的分析处理解决方案，能够在大数据环境中有效平衡分析深度与运营敏捷性。**

一、引言

在大数据快速发展的背景下，企业越来越需要能够无缝处理海量数据，同时提供近实时事务处理和强大分析能力的数据库系统。服务型工作负载要求系统对大量新摄入的数据执行复杂查询，并满足亚秒级延迟这一严格要求。这就要求系统能够高效处理数据并确保数据新鲜度、快速响应，同时在数据持续更新时保持高性能。传统联机分析处理（OLAP）系统往往难以满足这两方面的需求，因为它们通常针对相对静态的数据执行复杂的数据分析，且数据新鲜度的时间粒度较粗。一些现有的大数据解决方案采用多个不同系统的组合来支持混合服务与分析处理，但由于数据重复过多，以及系统间数据同步复杂，时间和存储效率都不尽如人意。

近期该领域的创新凸显了混合数据库的重要性。例如，HyPer [1] 使用硬件辅助复制来维护事务数据的一致性快照，从而支持并发执行 HTAP（混合事务/分析处理）操作，同时不会显著降低性能。PolarDB [2]集成计算存储，以优化云环境中的分析性能，展示了云原生数据库如何高效支持分析型工作负载。Greenplum [3]通过融入在线事务处理（OLTP）功能，扩展了传统的联机分析处理能力，并利用大规模并行处理（MPP）架构管理和分析分布在各个节点上的数 PB 数据。AnalyticDB-V [4],凭借对结构化和非结构化数据的查询融合能力，展现了混合分析引擎的一种先进实现方式。

此外，近期的市场解决方案，例如 YouTube 的 Procella [5],、阿里巴巴的 Hologres [6],以及字节跳动的 Krypton [7]，进一步探索了统一服务与分析型数据处理的系统。Procella 在单一框架内统一这两类工作负载，降低了系统复杂性，同时提升了性能。同样，Hologres 展示了在云原生服务中结合事务处理与分析处理能力的优势。Krypton 则体现了近实时服务与分析型工作负载之间的平衡，并借助集成的 SQL 引擎高效处理多样化的数据处理需求。尽管这些系统有效提升了数据新鲜度表现和数据分析效率，但采用新技术和新功能开发具有竞争力的 OLAP 系统，仍是一项充满挑战且令人振奋的探索。然而，现有 HTAP 系统仍面临取舍：有些系统需要独立的分析型基础设施（增加了复杂性），另一些系统则难以在保持事务处理能力的同时实现无缝的列存集成。OceanBase Mercury 的混合设计（列式基线、基于行的增量）独特地支持完整的 DML 操作，并具备分析性能；无需独立系统，即可实现可配置的数据新鲜度（从零到毫秒级）。

OceanBase 凭借经过验证的部署展现了互联网级的能力，能够在 1,500 节点集群 [8], [9]中处理万亿行数据集（≈20PB）。本文提出了 OceanBaseOLAP 数据库系统 Mercury[1], , 旨在支持 PB 级规模 [9], [10] ，并具备近实时分析处理能力，例如在线评论服务和

*杨传辉为通讯作者。

[1]Mercury在罗马宗教与神话中拥有惊人的速度，因此被用来命名我们的OLAP数据库系统。它是OceanBase 4.3.3的别名。

高德地图中的高精地图 [11]。OceanBase Mercury 基于这些创新，构建了分布式多租户架构，可确保高可用性、可扩展性和稳健的性能。其设计解决了在分布式事务 [12] 中维持 ACID（原子性、一致性、隔离性、持久性）特性的独特挑战，同时支持复杂的分析查询。通过在单一系统中集成 OLTP 和 OLAP 功能，它支持近实时商业智能和运营分析，这对现代企业应用至关重要。该系统采用多版本并发控制（MVCC）来确保一致性、基于分区的分布来实现可扩展性，以及复杂的复制机制来确保高可用性和容错性。

因此，本文的贡献如下：

- **新型混合存储架构**。我们在 LSM-tree 架构中提出了一种自适应列存。通过以行格式维护增量数据以保障事务效率，并以列式格式维护基线数据以提升分析速度，我们的设计使行存具备完整的 DML 能力，同时避免了传统OLAP系统典型的更新代价。

- **高效的物化视图**。我们优化了物化视图（物化视图）的实现，以确保数据一致性和查询效率。此外，我们为物化视图提出了高效的刷新机制，包括使用隐藏表的全量刷新和使用物化视图日志的增量刷新。

- **优化的向量化引擎**。我们设计了专为这种混合布局定制的向量化执行引擎。我们专门设计了三种数据格式，以适应各种计算场景。我们针对存储向量化重新设计并优化了若干算子和表达式，从而提升了向量化引擎的执行效率。

- **部署与评估**。我们已将 Mercury 部署于 OceanBase 的内部大数据技术栈以及公有云产品中，并在各种工作负载下开展了全面的性能研究。结果表明，OceanBase Mercury 即便与专用 OLAP引擎相比，也能取得更优的性能。

本文组织如下：第 II 节概述 OceanBase Mercury 的系统架构。第 III 节介绍 OceanBase Mercury 的列存。第 IV 节讨论物化视图，第 V 节介绍用于数据批处理的向量化执行引擎。第 VI 节通过实验验证 OceanBase Mercury 的高可用性和高性能。第 VII 节回顾相关工作，第 VIII 节讨论经验教训和未来创新。最后，第 IX 节给出结论。OceanBase 是一个基于木兰宽松许可证第 2 版开源的项目0 [13] ，源代码可在 Gitee [14] 和 GitHub [8] 上获取。

II. 系统架构

A. *OceanBase* 概述

OceanBase [9], [15] 采用无共享架构构建分布式集群。集群中的每个节点都配备独立的 SQL 引擎、事务引擎和存储引擎，从而确保出色的可扩展性、高可用性、高性能、低成本，以及与主流数据库的高度兼容性。集群中的多个节点汇聚为若干可用区。每个可用区包含多个称为 OBServer 的进程，用于处理数据；这些进程具有两个属性：互联网数据中心（IDC）以及 IDC 所在的区域或城市，为灾难恢复和高可用性设计提供地理信息。针对不同需求，OceanBase 提供多种高可用部署策略，包括同城三中心三副本、三地五中心五副本、同城和异地的主备部署模式，以及结合仲裁服务的多IDC部署解决方案，充分利用各可用区的地理分布特征，增强系统的韧性和灾难恢复能力。有关 OceanBase 架构设计的更多详情，请参阅 [15]。

B. 系统架构

如图1所示，OceanBase 通过用户定义的分区规则对单表数据进行水平分片，从而将表数据分布式存储在多个分区中。这种架构显著提升了数据管理的灵活性和处理效率。在此基础上，系统采用两级分区机制：一级分区作为数据组织的逻辑分组，二级分区则是其物理实现。每个物理分区都映射到一个平板——这是基本的存储单元，数据记录按顺序排列其中。这种层次化设计使逻辑结构与物理结构相匹配，从而优化存储效率，同时提升数据访问性能，并通过分布式平板管理实现水平扩展性。

如图1所示，OceanBase 通过即时记录重做日志的机制确保数据持久性。每当用户修改平板数据时，系统都会将这些更改写入与该平板专门关联的重做日志流。日志流 [16] 和各平板均采用多副本架构，副本经过策略性部署，分布在不同的可用区中，以降低单区域灾难风险。在此设计中，一个主副本（领导者）负责处理所有写操作，而次级副本（跟随者）则通过 Multi-Paxos 共识协议保持同步和一致性。主副本发生故障时，基于法定人数的选举过程可使幸存副本迅速选出新的领导者，并通过自动故障转移机制确保服务持续不中断。

每个集群节点运行一个 OceanBase 服务器进程（OB-Server），采用多线程架构实现。该

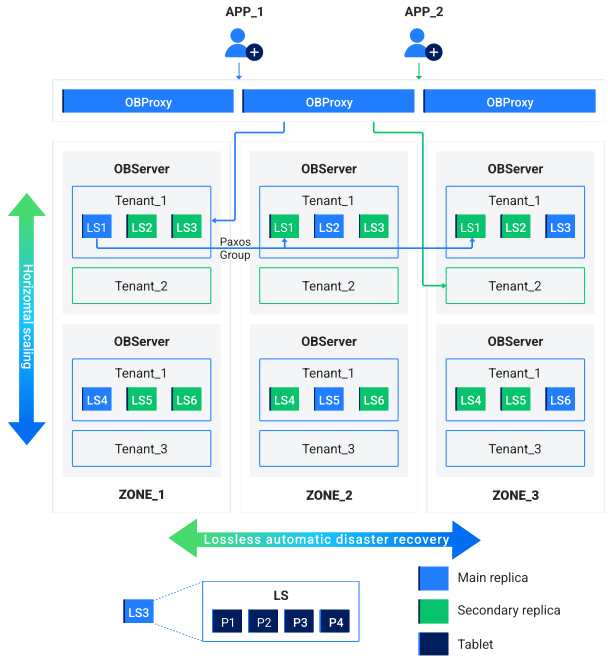


图 1. 系统架构

进程执行两项主要功能：1）访问存储在主机节点上的分区数据，进行本地数据管理；2）通过解析和处理路由到该机器的查询来执行 SQL。节点间通过 TCP/IP 协议进行通信，从而实现分布式组件之间可靠的数据交换。它还充当客户端接口端点，监听来自外部应用程序的连接请求，建立数据库会话，并通过标准化的 SQL 接口提供事务服务。

C. 分析处理引擎

OceanBase Mercury 的分析处理引擎集成到现有的 OBServer 架构中，与 OLTP 工作负载共享相同的存储和复制基础设施。这一设计选择带来了若干架构约束和设计决策：

与 OLTP 工作负载隔离： 分析引擎在列副本上运行，确保分析查询不会干扰正在进行的事务操作。分析查询从一致的列副本中读取数据，从而支持并发读写操作而不会相互阻塞。这种基于列副本的方法确保分析查询能够看到特定时间点的一致数据视图，同时事务写操作可以不间断地继续执行。然而，这一设计也带来了一些影响：(1) 列副本的创建和维护会产生开销，尤其是对于需要长时间持有列副本的长时运行分析查询；(2) 查询需要合并列式基线和基于行的增量存储中的数据，即使目标新鲜度为零，也会引入少量延迟（通常为毫秒级）。

面向批处理： 该分析引擎旨在实现面向吞吐量的批处理，而非针对单条查询的延迟进行优化。向量化执行引擎（见第 V 节）以批次为单位处理数据，最大限度地提高 CPU 利用率和内存带宽。这种设计符合分析型工作负载的特征：查询通常会扫描大量数据，因此更受益于高吞吐量，而非降低单条查询的延迟。批处理范式意味着系统优先考虑整体查询吞吐量，而不是单条查询的响应时间。这适用于分析型工作负载，但可能不适合要求在一秒内响应的交互式查询。

资源管理： 为避免分析查询影响在线事务处理性能，系统采用了资源隔离机制。每个租户都可以配置资源限制（CPU 和内存），确保分析工作负载不会占用事务操作所需的资源。至关重要的是，内部调度器经过专门设计，优先调度在线事务处理线程，确保即使在分析负载并发运行时，事务延迟也能保持稳定。尽管采取了这些架构层面的防护措施，资源争用——尤其是 I/O 带宽方面的争用——在生产环境中仍然是一个挑战。因此，根据我们的部署经验，最好将繁重的分析查询路由到专用资源池，或安排在非高峰时段执行，以尽可能减少干扰。

存储集成： 分析引擎利用 OceanBase 现有的列存储能力（见第 III 节），以列式格式存储基线数据，同时以行格式维护增量数据。这种混合方案使系统既能受益于列式存储的分析性能，又能兼容现有的基于行的事务型工作负载。分析查询会根据查询模式和数据位置自动访问相应的存储格式。然而，这种设计也增加了复杂性：查询必须合并来自列式基线和基于行的增量存储的数据，因此需要谨慎进行查询规划和执行优化。

OLTP 性能保障： 我们通过严格的架构保障，确保事务性能不受影响。增量数据以行格式维护，从而确保 DML 能力和事务写入性能与标准行存表保持一致。此外，OceanBase 通过多层隔离保护 OLTP 工作负载：（1）采用操作系统级 cgroup 隔离，防止分析查询耗尽 CPU 和内存资源；（2）通过将用于分析查询的列存副本置于追随者副本上，实现物理读写分离；（3）自动路由查询，将繁重的分析型工作负载重定向至列存追随者，从而避免对主副本上的 OLTP 工作负载造成干扰。有关 OceanBase 事务处理能力的更多详情，请参阅 [9]。

部署见解： 我们在 OceanBase 内部大数据栈和公有云服务中的生产部署揭示了若干实际考量。首先，共享基础设施的设计意味着容量规划必须兼顾在线事务处理和分析型工作负载，因此需要仔细监控并进行资源分配。

其次，基于快照的读取模型意味着分析查询可能会因快照选择而读取到略微陈旧的数据。对于大多数分析型工作负载而言，这可以接受，但需要向用户清楚说明数据新鲜度的保障。第三，面向批处理的处理模型意味着，单个查询的延迟可能因数据量和查询复杂度而有显著差异；在大规模分析型工作负载中，有些查询可能需要数毫秒到数秒。这种吞吐量与延迟之间的权衡是一项根本性的设计决策，与我们在生产环境中观察到的分析型工作负载特征相符。

III. 列存

列存在提升分析查询性能方面发挥着关键作用。它是 OceanBase 有效支持联机分析处理工作负载的一项关键特性。本节将深入探讨 OceanBase 列存的独特实现，并重点介绍其设计与优化。

A. 基线与增量数据

OceanBase 实现了行存与列存的融合。自诞生以来，它一直采用 LSM-tree 架构，并在该架构方面积累了丰富的工程实践经验。此外，OceanBase 还针对列存进行了许多特殊设计，例如将所有用户数据划分为两类：基线数据和增量数据。在列存设计中，数据通常不会原地更新，而是在客户端或数据库端进行批处理，并通过追加或合并整理为列式格式。OceanBase 的存储引擎采用多层 LSM-Tree，天然避免了原地更新，因此非常适合列存设计。

我们采用列式存储方案，在 LSM-Tree 最底层的压实过程中，将数据组织为列式格式（基线数据），而上层的增量数据（MemTable 和小型 SSTable）、日志、事务模块及其他组件保持不变。这样可以确保行存保留完整的 DML （数据操纵语言）能力（例如单行更新和大事务 [17]）以及数据同步能力（下游 CDC、Binlog）。其创新之处在于与 LSM-tree 集成的混合设计：基线数据（大合并）转为列式存储，而增量写入仍采用基于行的方式。与牺牲 DML 能力的传统列存储不同，这种设计在实现分析性能的同时，保留了完整的事务操作能力（单行更新、大事务），无需独立系统即可实现真正的 HTAP。

- **基线数据：** OceanBase 利用其分布式多副本架构，实现一种名为“DailyCompaction”的专有机制。该过程通过两个不同阶段运行：首先，租户既可安排周期性压实周期，也可通过管理命令触发压实，以指定全局版本号。其次，该租户分布式存储中的所有数据副本执行协调一致的大合并，以与该版本同步，从而产生特定版本的基线数据。至关重要的是，该架构保证同一版本在所有副本上具有完全相同的基线数据，并通过确定性合并结果确保数据一致性与全系统完整性。为满足近实时 OLAP 的需求，查询引擎对该静态基线数据与动态的基于行的增量数据执行即时合并。这确保了尽管转换为列式格式的过程每日进行，数据可见性仍是即时的（实现数据新鲜度≈ 0，如 [18] 中的“SystemA2”所示），从而使分析查询能够访问最近已提交的事务，而无需等待压实周期。

- **增量数据：** 基线数据最新版本之后写入的所有数据都视为增量数据。具体而言，增量数据既可以是用户刚写入 MemTable 的内存数据，也可以是已经转储到 SSTable 的磁盘数据。MemTable 和 SSTable 是 OceanBase 存储引擎的组成部分：MemTable 位于内存中，用于存储动态数据，并提供读写操作；SSTable 位于磁盘上，用于存储静态数据，且为只读。

在大规模生产环境中，增量数据占每日写入量的 1%–10%。读取这些数据可确保近乎实时的可见性，但即时合并会带来开销：仅访问基线数据的查询通常比合并大量增量数据的查询快 5–10× 倍。每日压实周期至关重要，它会定期将基于行的增量数据转换为列式基线，并通过安排在非高峰时段执行，尽可能减少资源争用，同时恢复分析性能。

B. 自适应存储

OceanBase 利用基线数据和增量数据的特点，提出了一组对上层透明的列存实现方法：

- 基线数据以列存方式存储，增量数据则保存在行存中。所有用户 DML 操作均不受影响。列存表仍可像行存表一样执行所有事务操作。
- 在列存模式下，每列数据均存储为独立的 SSTable，所有列 SSTable 合并为一个虚拟 SSTable，作为用户的列存基线数据，如图 2 所示。
- 根据用户的表创建设置，基线数据可以采用三种存储模式：行存、列存，或行存与列存冗余存储。

C. TP/AP 一体化

如图 3 所示，我们不仅在存储引擎中实现了列存，还对优化器、执行器及其他存储相关模块进行了适配和优化。这使得现有的、具有联机分析处理需求的 OceanBase 客户能够在不影响运行中业务的情况下，从行存储迁移到列存。其他 OLAP 数据库的用户也可以通过 OceanBase 提供的迁移工具（即 OceanBase 迁移服务（OMS） [19]）迁移其数据

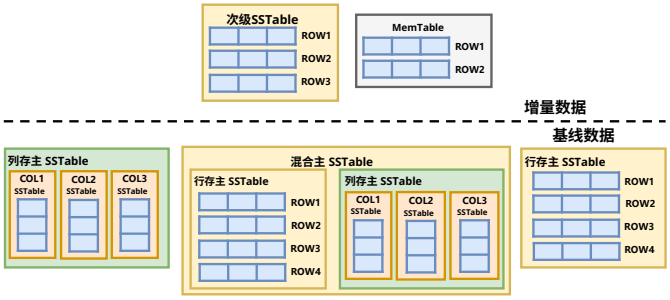


图 2. 混合存储架构

，并使用 OceanBase 的统一架构。用户可以像使用行存一样轻松地享受列存带来的性能优势。这种集成还使 OceanBase 能够通过统一的引擎和代码库实现真正的 TP/AP 协同，借助统一的引擎和代码库支持多样化的业务负载。有关 OceanBase TP 能力的更多详情，请参阅 [9]。

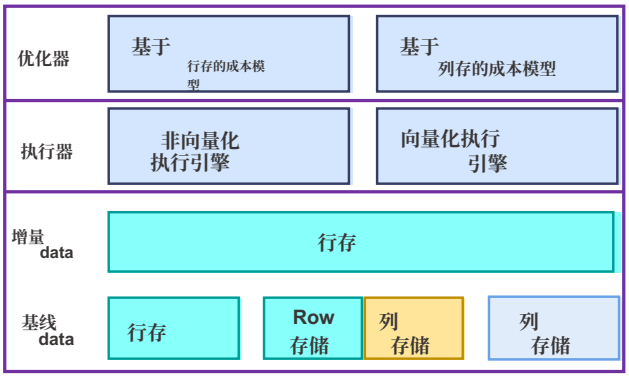


图 3. 行存与列存集成

- **TP/AP 一体化 SQL 引擎**。我们针对列式存储架构开发了创新的成本模型，并将面向列的统计指标纳入查询优化器。增强后的优化器通过基于成本的分析，综合考虑底层数据存储格式，动态生成执行计划。此外，我们还打造了全新的向量化处理引擎，并对核心关系运算符进行了架构升级。该自适应系统通过持续评估成本，智能调整向量化粒度和批处理维度，从而针对不同的查询模式实现最佳资源利用率。

- **TP/AP 一体化存储引擎**。用户数据根据表模式指定，可根据业务负载类型灵活设置为列存、行存或行列冗余模式。用户的查询、备份与恢复操作完全透明。列存表全面支持所有在线和离线 DDL 操作、所有数据类型以及二级索引创建，确保用户的使用方式与行存没有区别。

- **TP/AP 一体化事务引擎**。所有增量数据均以行存储。事务修改、日志内容和多版本控制均与行存采用相同的逻辑。

D. 自适应压实

引入新的列存模式后，数据合并行为与原有的行存数据相比发生了显著变化。由于所有增量数据都采用行存，因此需要与列存的基线数据合并，并为每一列拆分成独立的 SSTable。与行存相比，合并时间和资源使用量都会显著增加。

为了加快列存表的合并速度，压实过程得到了大幅优化。除了像行存表一样通过水平切分、并行处理和合并来加速外，列存表还实现了垂直切分加速。此外，列存表会将多个列的合并整合为一个合并任务。单个任务中的列数可根据可用系统资源自动确定，从而在整体合并速度和内存使用之间实现更均衡的权衡。

E. 列编码

OceanBase 通过两级压缩来存储数据。第一级是 OceanBase 自主研发的行列混合编码压缩，第二级是通用压缩。自主研发的行列混合编码是内置的数据库算法，支持无需解压缩即可直接查询，并可利用编码信息加速查询过滤，尤其适用于 AP 查询。例如，在增量编码中，当存储的列包含定长数值时，只需存储最小值以及每行与最小值之间的差值。它还支持字符串编码，例如对具有相同前缀的列进行前缀编码。此外，当一列是另一列的前缀时，会存储一列的完整内容和另一列的后缀，从而减少冗余，尤其适用于包含重复时间戳的列。更多详情请参阅 OceanBase[2] 的文档。

由于原有的行列混合编码算法往往面向行组织存储实现，OceanBase Mercury 专门为列式存储表设计了新的列编码算法。与原有算法相比，新方法支持完全向量化查询执行，可跨多种指令集进行 SIMD 优化，并显著提高数值数据的压缩率，从而在查询性能和压缩率两方面实现整体提升。

F. 数据跳过索引

数据跳过索引，有时也称为小型物化聚合 [20] 或区域映射 [21]，，是一种在查询处理中剔除无关数据的常用方法，对存储和计算的开销极低。与传统二级索引通过在专用数据结构中存储索引键列来加速范围扫描或点查询不同

[2]https://en.oceanbase.com/docs/common-oceanbase-database-10000000001973723

（例如 B+ 树），其规模比原始数据小几个数量级。即使来自用户查询的优化器生成的过滤条件表现出低选择率（即过滤率），它也能在无需精细优化的表结构或在索引键上精确定义的查询范围的情况下实现稳健性能。此外，与传统索引方法相比，数据跳过索引在部署上更具灵活性，并显著降低了维护复杂度。

我们采用这一思路来计算并存储预聚合数据，例如每列数据的最小值、最大值、求和值和空值计数，并将其作为草图。这些草图持久化存储在每个列 SSTable 上基于块的树形块索引中，作为我们的数据跳过索引。索引树中的草图会递归聚合，因此索引树节点中的数据跳过索引可以表示其所指向的所有子节点的草图，使索引树不同层级上的数据跳过索引分别对不同粒度的数据执行预聚合。这种多粒度预聚合可以提供从数据块粒度到 SSTable 粒度的数据草图，并能很好地适应列上的灵活范围查询和随机数据修改；后者会使 DML 中覆盖受影响数据的预聚合数据失效。借助数据跳过索引，下推至存储引擎的谓词可以有效剪除无关数据，无需访问所有数据块，从而减少不必要的数据访问开销和 I/O消耗。跳过索引中的草图还可用于高效的聚合计算和数据统计信息收集。

与现有工业实现相比，我们的架构将数据跳过索引的适用范围扩展到了更多场景。不同于将数据跳过索引与数据分离，作为元数据服务或独立索引结构，并在执行前计算查询涉及的数据块的数据库（例如 Snowflake），OceanBase 将数据跳过索引直接集成到 SSTable 结构中。这种自包含设计简化了压实、备份、迁移和 DML 等数据操作，因为所有修改都封装在 SSTable 中。此外，在执行期间进行块评估，可针对涉及运行时参数或复杂表达式的复杂谓词进行动态剪枝。传统的区域映射或数据跳过索引实现通常依赖固定大小粒度块，因此必须在剪枝精度与存储和计算开销之间权衡；而 OceanBase 的数据跳过索引则天然具备粒度自适应能力。通过将预聚合草图与索引块树的层次节点对齐，我们实现了多分辨率数据跳过。最细粒度对应原子读取I/O单元（数据块），随着树向上延伸，自然形成更粗粒度的摘要。这种层次结构能够高效评估复杂组合布尔过滤条件，而且索引维护开销极低。除了过滤处理之外，数据跳过索引还用于聚合处理和优化器统计信息收集，例如在 OceanBase 中进行范围、不同值数量和有序性估计。

G. 查询下推

在 OceanBase Mercury 中，存储引擎已全面支持向量化，并提供了更多下推支持。在列存储引擎中，下推功能得到了进一步增强和扩展，包括：

- **下推所有查询过滤器**。所有查询的过滤器都可以下推到存储层。此外，根据过滤器的类型，还可以利用跳过索引和编码信息（参见第 III-E 节）进一步提升查询性能。

- **下推常见聚合函数**。在非分组场景中，计数、最大值、最小值、求和和平均值等聚合函数可以下推到存储引擎中执行计算。

- **下推分组操作**。对于不同值数量（NDV）较少的列，存储引擎会通过构建内部字典对数据进行编码，并为每行存储引用。分组下推利用这些字典信息，可显著提升性能。

IV. 物化视图

物化视图是一种特殊的视图，用于存储视图定义查询的结果。它通过以空间换时间来加速 AP 查询。物化视图保留某些耗时查询的查询结果，从而避免重复计算这些查询。

A. 概述

普通视图仅存储视图的定义，而不存储实际数据。因此，每次查询该视图时，都需要再次执行该视图的底层查询语句。物化视图包含一个专用的容器表，用于存储视图中所定义查询的结果。当用户查询物化视图时，数据库直接从该容器表中检索数据，从而无需重复计算。

基表同时包含基线（列式）数据和增量（基于行的）数据。物化视图由基表计算得出，并反映这两种数据类型。压实将增量数据转换为基线数据，但不会直接影响物化视图：全量刷新会查询基表（两种类型），而增量刷新则使用 mlog 跟踪基表的所有变更。物化视图是存储预计算结果的独立容器表。此外，由于物化视图的数据是根据其定义的查询结果预先计算的，当基表数据发生变化时，物化视图的内容可能会过时。为确保物化视图数据保持最新，OceanBase 提供全量刷新和增量刷新机制，以刷新容器表中存储的结果，从而反映基表的最新状态；这些机制将在第 IV-B 节和第 IV-C 节中讨论。

B. 全量刷新

执行全量刷新时，数据库会重新执行与物化视图关联的查询语句，重新计算并替换现有视图结果数据。

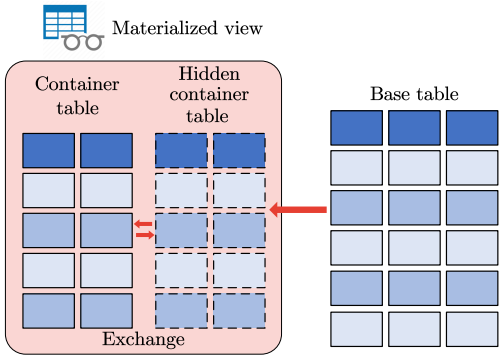


图 4. 全量刷新

OceanBase 使用异地刷新方法执行全量刷新（如图 4 所示），即创建隐藏表，在隐藏表上执行刷新语句，然后切换原表和隐藏表。因此，全量刷新操作需要额外的空间，并且会完整重建索引（如果有）。全量刷新可能耗时较长，尤其适用于自上次刷新以来基表发生重大变化，或基表中的数据量相对较小的场景。此外，为加快处理速度，全量直接加载方法会将完整数据集直接写入隐藏表的数据文件。该方法绕过 SQL 接口和事务处理层，直接分配空间，并将待插入数据写入 SSTable 结构（这是 OceanBase 存储中数据的内部表示形式），从而显著提高数据导入效率。全量直接加载方法非常适合 10 GB 以上级别的数据迁移，以及 CPU 和内存资源有限的环境。其精简的执行路径可最大限度地减少 CPU 消耗，是此类场景的高效解决方案。

C. 增量刷新

如果物化视图配置了增量刷新策略，数据库会根据基表的更新，定期或立即计算物化视图的变更，并将这些变更应用到物化视图。增量刷新的成本远低于全量刷新，适用于基表频繁更新的场景。OceanBase 利用物化视图日志（后文简称 mlog）记录更新的数据。因此，如果要创建采用增量刷新策略的物化视图，需要先基于基表创建物化视图日志。物化视图日志在内部实现为普通表，用于存储所有更新行的旧值和新值。

图 5 展示了增量更新的整体框架。在写入时，所有 DML 语句都会通过数据访问服务（DAS）同时更新基表和物化视图日志，而后台维护任务会定期将增量更新刷新到物化视图中。此外，维护任务还会定期删除

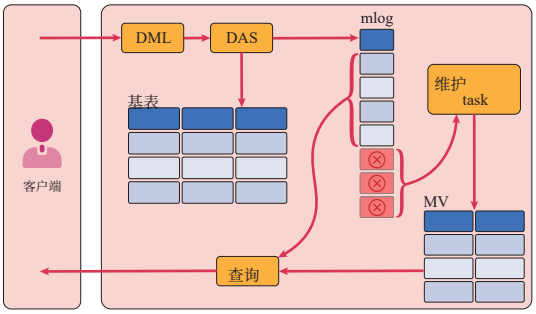


图 5. 增量更新的整体框架

已在物化视图中更新的记录。在查询时，查询模块会同时查询物化视图未更新部分和 mlog 中的数据，将它们合并后返回给用户，确保查询到的数据是最新的。

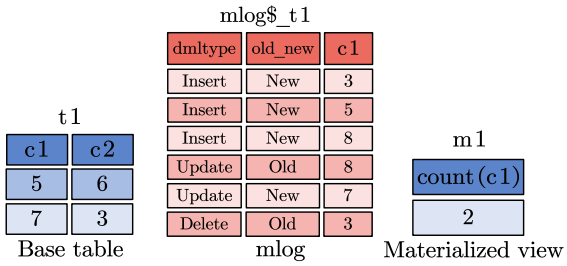


图 6. 增量刷新的示例

下面通过一个示例来说明增量刷新，如图 6 所示。原始表 *t1* 和物化视图 *m1* 的定义如下：

```
创建表 t1（c1 为 int 主键，c2 为 int）；创建物化视图 m1，内容为从 t1 中选择计数（c1）。
```

插入/删除/更新序列如下：

```
向 t1 插入值(3,4)；向 t1 插入值(5,6)；向 t1 插入值(8,3)；更新 t1，将 c1 = 7 设置为 c1 = 8；从 t1 中删除 c1 = 3；
```

对应的 *mlog* 是 *mlog$ t1*，其中 *dmltype* 指定数据操作语言操作的类型，而 *old new* 表示该行记录的是更新前还是更新后的数据。*m1* 的更新值可通过以下方式获取：

```
（select 计数（c1） from m1 where old_new = 'New'） - （select 计数（c1） from m1 where old_new = 'Old'）；
```

OceanBase 对增量刷新使用原位刷新方法，即直接在物化视图上执行刷新语句。如果增量数据过多，增量刷新可能比全量刷新更慢。为了加速

该过程，它还支持用于增量刷新的（增量）直接加载。该方法将增量数据直接构造成存储层格式（即 SSTable，由其元数据信息和一系列数据组成），并将更新后的数据直接写入存储层。增量旁路导入显著提升了刷新效率。

D. 刷新算法复杂度分析

增量刷新算法由两个阶段组成：(1) 刷新类型检查，用于确定刷新策略；(2)刷新 *DML* 生成，用于生成 SQL 更新语句。我们使用以下符号分析复杂度：**T** （表），**C**（列），**A** （聚合），**J** （连接），**S** （SELECT 项），**D** （增量行），**M** （物化视图行），**N** （UNION ALL 分支），以及**P** （分区）。

表 I 总结了不同视图类型的复杂度。简单 MAV 的复杂度最低，为 $O(S \times A)$；外连接视图的复杂度为二次，即 $O(T^2 \times (S + J + A))$，这是由表对处理导致的。主要刷新受益于分区级处理，其复杂度为 $O(P \times (S+J))$。对于具有适当索引的刷新操作，实际执行复杂度为 $O(D \times \log(M))$；或者，在并行度为 P 时，复杂度为 $O(D \times \log(M)/P)$。

表 I 刷新算法复杂度分析

视图类型	Time	空间
简单 MAV	$O(S \times A)$	$O(S + A)$
简单 MJV	$O(T \times (S + J))$	$O(T \times S)$
连接 MAV	$O(T \times (S + A + J))$	$O(T \times (S + A))$
外连接 MJV/MAV	$O(T^2 \times (S + J + A))$	$O(T^2 \times S)$
UNION ALL 物化视图	$O(N \times T \times (S + C))$	$O(N \times T \times S)$
MJV 完全刷新	$O(P \times (S + J))$	$O(P \times S)$
执行（已索引化）	$O(D \times \log(M))$	$O(D)$

V. 向量化执行引擎

向量化执行是一种高效的数据批处理技术。在分析查询中，向量化执行可以大幅提升执行性能。从 OceanBase 4.0 [15], 开始，向量化执行引擎默认启用。通过优化数据格式、算子实现和存储向量化，其执行性能得到了大幅提升。

A. 数据格式优化

如图7所示，OceanBase 的向量化引擎实现了一种新的列式数据格式。数据描述信息空值、长度和指针 信息按列分别连续存储，以避免数据存储信息冗余。针对不同的数据类型和使用场景，实现了三种数据格式：定长数据格式、变长离散格式和变长连续格式：

•**定长数据格式**只需要一个空值位图和连续数据信息，只需存储一个长度值。无需冗余地存储相同的长度值用于批次中的每条数据，也不再需要用于间接访问的指针信息。数据信息无需冗余存储，可以节省更多空间；数据可直接访问，访问时的数据局部性更好；数据可以连续存储，对 SIMD 指令更加友好。此外，在物化和序列化时，无需对数据执行指针置换操作，效率更高。

• **变长离散格式** 是指在一批数据中，每个数据都可以不连续地存储在内存中，而每个数据在列中则使用数据地址指针、长度描述、长度信息和指针信息连续存储；如果存储层存储的是编码数据，采用这种格式时，在进行投影时无需深度复制数据，只需投影长度 和指针 信息。在短路计算场景中，一批数据可能只需计算其中几行，这种格式的描述也无需重新组织数据。

• **变长连续格式**是指数据在内存中连续存储。每个数据的长度信息和偏移地址使用偏移数组进行描述。与离散格式相比，这种描述格式要求在组织数据时确保数据连续，因此数据访问和批量复制数据的效率更高。不过，对于短路计算场景和列编码数据投影而言，这种格式并不友好，因为这些场景需要重新组织并深度复制数据。目前，这种格式主要用于基于列的物化场景。

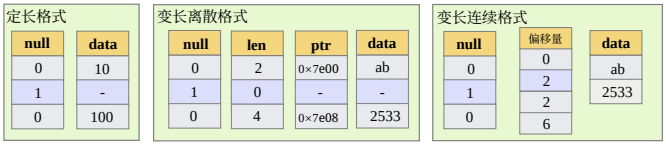


图 7. OceanBase 向量化引擎的数据格式

从磁盘读取微块数据后，存储层会对数据进行解压和解码，然后将其投影为与 SQL 层对应的内存格式。与之前的格式相比，所提出的定长格式性能更佳。值得注意的是，新定长格式简化了这一过程：投影为内存格式时无需设置 长度 和指针 ； 由于定长数据连续存储，还可以高效地执行批量 *memcpy* 操作。相比之下，采用之前的格式填充一批定长数据时，需要为每个 Datum 数据设置长度、空值和 指针 ，还需要将整数数据值填入由 指针指向的数据缓冲区，因此新格式的效率更高。

B. 运算符与表达式优化

向量化引擎对算子和表达式的实现进行了全面优化。主要优化思路是利用批数据属性信息、算法和数据结构优化，并基于新格式进行专门实现，从而减少

CPU 数据缓存未命中、CPU 分支预测错误和 CPU 指令开销，并提升整体执行性能。向量化引擎采用新格式重新设计并实现了排序、哈希连接、哈希分组、数据洗牌和聚合计算等算子和表达式，全面提升了整体计算性能。

1) 利用批数据属性信息：向量化引擎在执行过程中会维护批数据的特征信息，包括是否存在空值 以及批次中的所有行是否都无需过滤。这些信息能大幅加快表达式计算。例如，如果不存在空值，表达式计算时就无需考虑对空值 进行特殊处理。如果没有行被过滤，计算时就无需判断每一行是否已被过滤。此外，如果数据连续且未经筛选，则更适合进行单指令多数据计算。

2) 算法和数据结构优化：在算法和数据结构优化方面，实现了更紧凑的物化结构，支持按行或按列物化数据，从而节省空间并提高访问效率；排序 算子将排序键与非排序键分别物化，并结合排序键的保序编码（将多列数据编码为一列，可直接使用*memcmp*进行比较），使排序操作在比较过程中访问数据时的缓存未命中率更低、比较计算更快，整体排序效率更高。哈希分组 优化了哈希表结构，哈希桶中的数据存储更加紧凑。针对低基数分组键，采用数组进行优化，并将分组和聚合结果连续存储在内存中，等等。

3) 特化实现优化：该方法主要使用模板，为不同场景生成更高效的实现。例如，哈希连接特化将多个定长连接键编码为一个定长列，并将连接键数据放入桶中。它还优化了数据预取，并减少访问多个数据列时的数据缓存未命中。该方法支持聚合计算的特化实现，并分别实现不同的聚合计算，从而减少每次计算所需的聚合函数指令数和分支判断次数，大幅提升执行效率。

4）存储向量化优化：存储层全面支持新的向量化格式，并在投影、谓词下推、聚合下推和分组下推中更多地使用单指令多数据。投影定长和变长数据时，根据列类型、列长度以及是否包含空值 信息定制模板，并以批量浅拷贝的方式进行投影。计算下推谓词时，对于简单谓词计算，直接在列编码上执行；对于复杂谓词，则将其投影到新的向量化格式中，并在其上进行批量计算。

表达式。聚合下推充分利用中间层的预聚合信息，例如计数、求和、最大值、最小值等。分组下推充分利用编码数据信息，对于字典编码，具有非常明显的加速效果。

VI. 实验

本节开展了大量实验，在不同配置下评估 OceanBase Mercury（OceanBase 4.3.3 的别名），并将其与多种开源联机分析处理系统进行比较。

A. 实验设置

硬件设置。我们在单服务器以及由三台不同服务器组成的分布式集群上部署了 OceanBase Mercury。除非另有说明，实验均在配备 128 核 Intel(R) Xeon(R) Platinum 8369B CPU@2.90GHz、1TB 动态随机存取存储器和 44TB 机械硬盘的服务器上进行。

基线。我们对 OceanBase 4.3.3 与其他分布式数据库系统，即 StarRocks 3.3.3 和 ClickHouse 24.10.1，进行了全面比较。为确保评估公平，这些系统均部署在同一区域，且集群规模相同。

基准测试。在实验中，我们采用了三种基准测试：TPC-H [22], TPC-DS [23],和 ClickBench [24]。**指标**。我们通过测量基准测试查询的查询延迟来评估数据库的性能。每条 SQL 语句分别在热运行和冷运行条件下执行，以评估其查询延迟。冷运行时，我们会重启数据库系统并清空内存，然后执行查询并记录执行时间。相比之下，热运行时，我们会将每条 SQL 语句执行三次，并记录最短的执行时间。

B. 列编码的性能评估

OceanBase Mercury 实现了多种先进的列编码技术，包括多前缀编码、列间相等编码和列间子串编码，以提高压缩率并降低存储开销。如图 8 所示，我们在实验中使用了来自实际业务场景的十张表（即 $T1, \cdots, T10$）。加入多前缀编码、列间相等编码和列间子串编码后，约半数表的压缩率有所提升。 $T1$ 的空间节省率从 15% 提高到 17%（LZ4：从 10% 提高到 12% ），这主要得益于前缀编码和列间子串编码。 $T2$ 和 $T3$分别从 59% 提高到 67%、从 16% 提高到 25%，主要得益于前缀编码。 $T5$ 取得了显著提升，使用这三种技术后，空间节省率从 30% 提高到 48%（LZ4：从 20% 提高到 26%）。$T7$ 的提升幅度最大，从 66% 跃升至 87%（LZ4：从 32% 提高到 43%），这主要归功于前缀编码。 $T10$ 也有所提升，从 28% 提高到 33%（LZ4：从 10% 提高到 11%）。这些结果凸显了利用列级冗余实现更高压缩率的有效性。

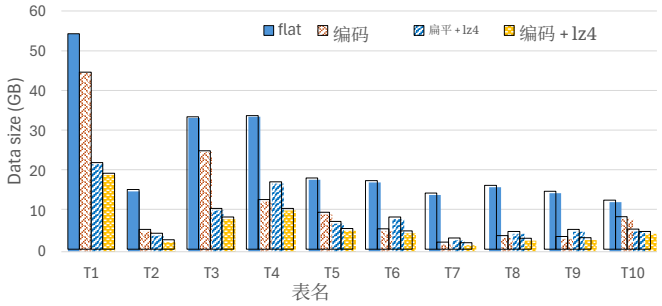


图8. 编码后的数据大小

C. 基于行的物化视图与基于列的物化视图的性能比较

在当前数据库环境中，物化视图作为一种优化方法，已被广泛用于加速报表生成和数据分析，尤其是在需要连接多个表以构建大型宽表进行深度分析的场景中。与此同时，自4.3.0版本起，OceanBase通过集成全新的列存储引擎，成功实现了行存储与列存的融合。这不仅显著提升了联机分析处理场景下的查询性能，也兼顾了在线事务处理业务的需求。

然而，在此版本之前，OceanBase 的物化视图采用行存储格式。尽管这种格式能够满足基本的数据存储需求，但面对大量数据和复杂的分析查询时，其性能提升空间有限。为弥补这一不足，尤其是在以分析查询为主的联机分析处理场景（如 OLAP 系统和数据仓库）中，引入列存物化视图来处理大数据集并执行复杂的分析任务，可以提高查询效率。我们在由三台服务器组成的分布式集群中进行了该实验。

表 II 全面比较了在不同基表存储引擎（行存储和列存储）及数据规模（10^8 行和 10^9 行）下，视图查询、基于行的物化视图（MV）和基于列的物化视图的性能。评估涵盖了七种常见的聚合操作：计数（*）、计数（c1）、计数（c2）、求和（c2）、平均值（c2）、最大值（c2）和最小值（c2）。

结果表明，与直接执行视图查询相比，物化视图可带来显著的性能提升。对于行存储基表，视图查询的延迟约为 32–33 ms，而基于行的物化视图可将其降低至 2–5 ms（提速约 6–11× 倍），基于列的物化视图则可达到 1.7–3.8 ms（提速约 8–19× 倍）。对于列存储基表，视图查询的延迟约为 13–14 ms，而基于行的物化视图可达到 3–6 ms（提速约 2–4× 倍），基于列的物化视图则可达到 1.7–3.7 ms（提速约 4–8× 倍）。在所有场景中，基于列的物化视图始终优于基于行的物化视图；对于行存储基表中包含 10^8 行的数据，总查询延迟从 227.1 ms（视图）降低至 17.2 ms（基于列的物化视图），性能提升了 13×。对于涉及特定列操作的分析查询，尤其是计数（c1）和计数（c2），基于列的物化视图的性能优势更加明显：其延迟可低于 1 毫秒（1.7–2.7 ms），而基于行的物化视图为 2.0–3.2 ms。值得注意的是，当数据规模从 10^8 行扩展到 10^9 行时，性能仍保持稳定，表明该物化视图实现具有出色的可扩展性。

基于行和基于列的实时物化视图均通过合并 mlog 与物化视图数据来查询最新的增量数据，实现了接近 0 的数据新鲜度。资源消耗因工作负载特征而异：列存物化视图在包含分析查询的联机分析处理场景中表现出色，而行存物化视图对于在线事务处理工作负载更为高效。对于 10^8 行（未进行编码和压缩时为 ≈ 4.3 GB），数据量约为 3.2 GB；而对于 10^9 行（未进行编码和压缩时为 ≈ 43 GB），在 OceanBase 中经过编码和压缩后，数据量约为 31 GB。

D. 向量化执行引擎的性能评估

OceanBase Mercury 实现了向量化引擎，针对各种列式数据格式设计了新的数据格式描述。向量化引擎消除了序列化开销，以及维护原有数据描述方式所产生的读写访问开销。基于新的数据格式描述，我们重新实现了各种算子和表达式，从而提升了其处理大规模数据的效率。

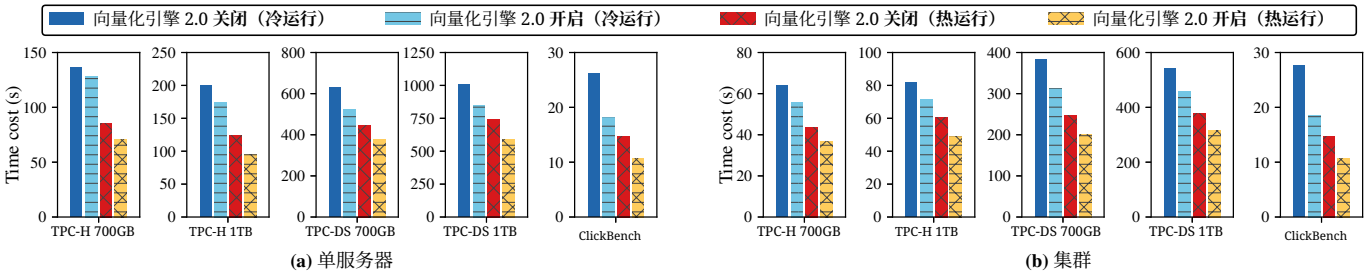
图9比较了在 TPC-H（700GB、1TB）、TPC-DS（700GB、1TB）和 ClickBench 基准测试中，禁用和启用向量化引擎时 OceanBase Mercury 的总查询延迟。启用向量化引擎后，OceanBase Mercury 在单服务器和集群设置下均取得了显著提升。向量化引擎在冷运行设置下将总查询延迟降低了 6% 至 33%，在热运行设置下则降低了 18% 至 28%。在 ClickBench 中提升最大，因为其查询针对单个大表。启用向量化引擎后，在图9（a）- ClickBench 的单服务器设置下，OceanBase Mercury 将查询延迟降低了 30%（冷运行）和 28%（热运行）；在图9（b）- ClickBench 的集群设置下，则将查询延迟降低了 33%（冷运行）和 27%（热运行）。

图10、图11和图 12分别展示了 TPC-H (1TB)、TPC-DS (1TB) 和 ClickBench 基准测试中各项查询的详细性能。启用向量化引擎后，OceanBase Mercury 几乎降低了每项查询的查询延迟。对于一些耗时最长的查询，性能提升尤为显著。例如，在图11 (b) 中，查询23的查询延迟从 138.99 秒降至 77.17 秒，提升约 44.5%。

为进一步评估向量化引擎的有效性，我们比较了启用向量化 2.0 后列存和行存配置的性能。

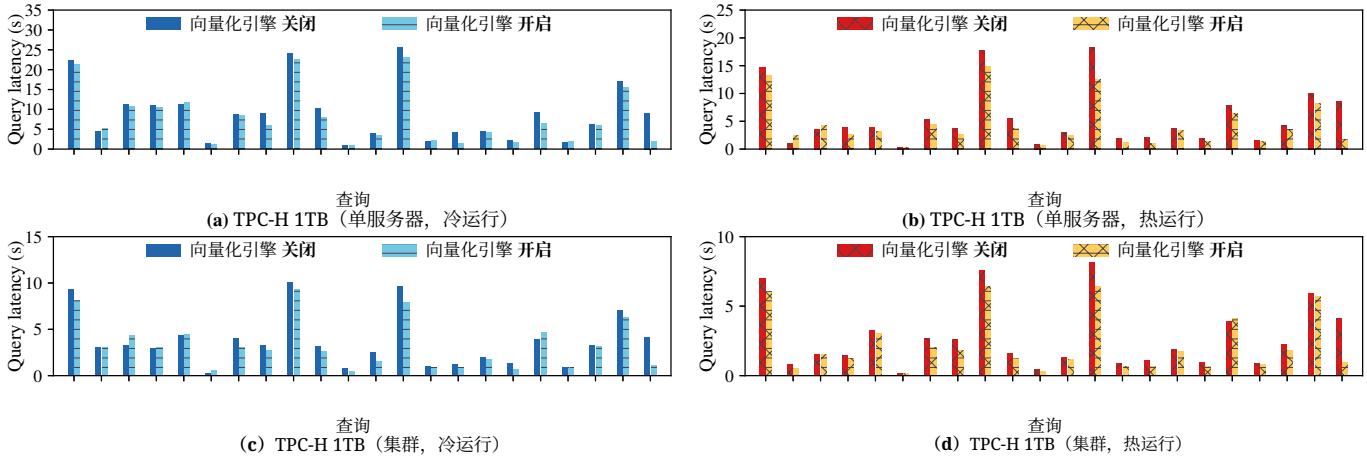
表 II 不同行数和基表存储引擎下，基于行的物化视图与列式物化视图上查询的延迟。

算子	含有 10^8 行的基表（毫秒）						含有 10^9 行的基表（毫秒）					
	行存基表			列存基表			行存基表			列存基表		
	视图	基于 R 的物化视图	基于 C 的物化视图	视图	基于 R 的物化视图	基于 C 的物化视图	视图	基于 R 的物化视图	基于 C 的物化视图	视图	基于 R 的物化视图	基于 C 的物化视图
计数（*）	32.3	4.7	3.8	13.9	6.3	3.7	31.9	5.1	5.0	13.9	6.2	3.7
计数（c1）	32.6	2.6	2.6	13.8	3.3	2.5	31.6	2.3	2.0	13.7	3.0	2.7
计数（c2）	32.2	2.6	2.1	13.8	3.0	2.5	31.8	2.0	2.0	13.9	3.0	2.3
求和（c2）	32.8	2.7	2.4	13.9	3.0	2.6	32.6	2.0	2.0	13.6	3.0	2.2
平均值（c2）	32.3	2.5	2.2	13.8	3.0	2.4	32.7	2.0	1.8	13.8	3.0	2.2
最大值（c2）	32.7	2.5	2.0	13.9	3.0	2.5	32.4	2.0	1.8	13.8	3.1	2.9
最小值（c2）	32.2	2.5	2.1	13.8	3.2	2.6	33.2	3.0	1.7	13.8	3.1	3.0
总计	227.1	20.1	17.2	96.8	24.8	18.8	226.2	18.6	16.3	96.7	24.4	19.0



(a) 单服务器　　**(b)** 集群

图 9. 关闭和开启向量化引擎时 OceanBase Mercury 的总查询延迟。



(a) TPC-H 1TB（单服务器，冷运行）　**(b)** TPC-H 1TB（单服务器，热运行）

(c) TPC-H 1TB（集群，冷运行）　**(d)** TPC-H 1TB（集群，热运行）

Fig. 10. 在 TPC-H（1TB）测试中，关闭和开启向量化引擎时 OB Mercury 的查询延迟 .

如表 III 所示，在所有基准测试中，列存的性能均优于行存；ClickBench、TPCH- 100G 和 TPCDS-100G 的执行时间分别为 14.54s、19.42s 和 108.19s，相较于行存实现了 1.7× 至 1.8× 倍的加速。

向量化引擎 2.0 在计算密集型查询方面表现出色。不过，在某些特定场景中会出现轻微退化，例如图 12 (a) 中的 ClickBench Q5（对“UserID”列进行去重计数）。这是由初始化开销导致的，例如批处理缓冲区分配；在 I/O 受限场景中，计算节省的开销无法完全抵消这些开销。尽管如此，该引擎仍能提供显著且稳定的为绝大多数分析型工作负载带来改进。

启用向量化 2.0 时的性能比较（热运行）

基准测试	列存（秒）	行存（秒）
ClickBench	14.54	24.32
TPCH-100G	19.42	35.27
TPCDS-100G	108.19	155.40

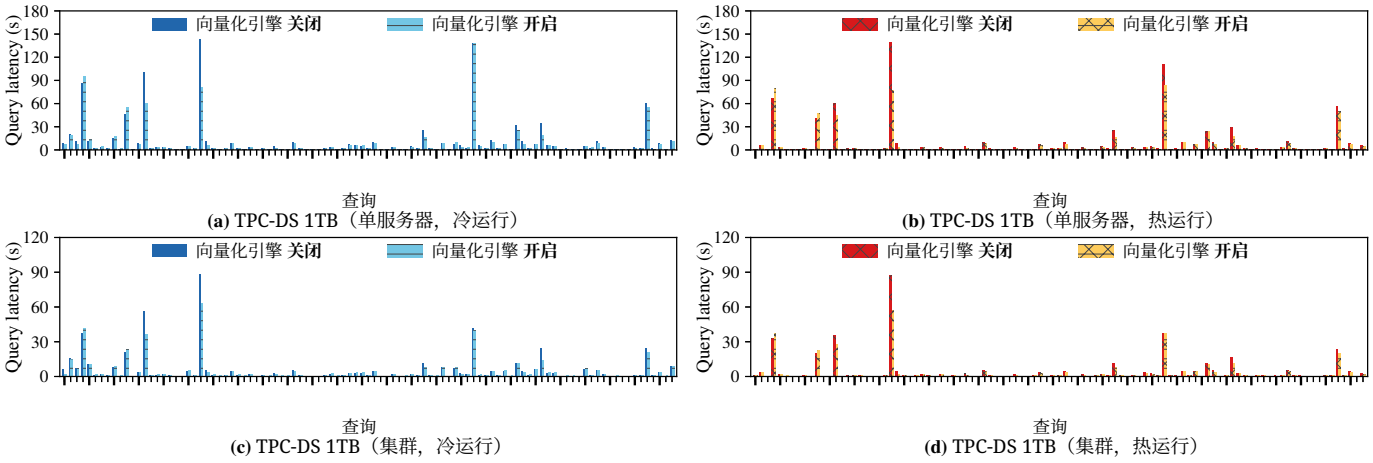


图11. 在 TPC-DS（1TB）基准测试中，关闭和开启向量化引擎时 OceanBase Mercury 的查询延迟。

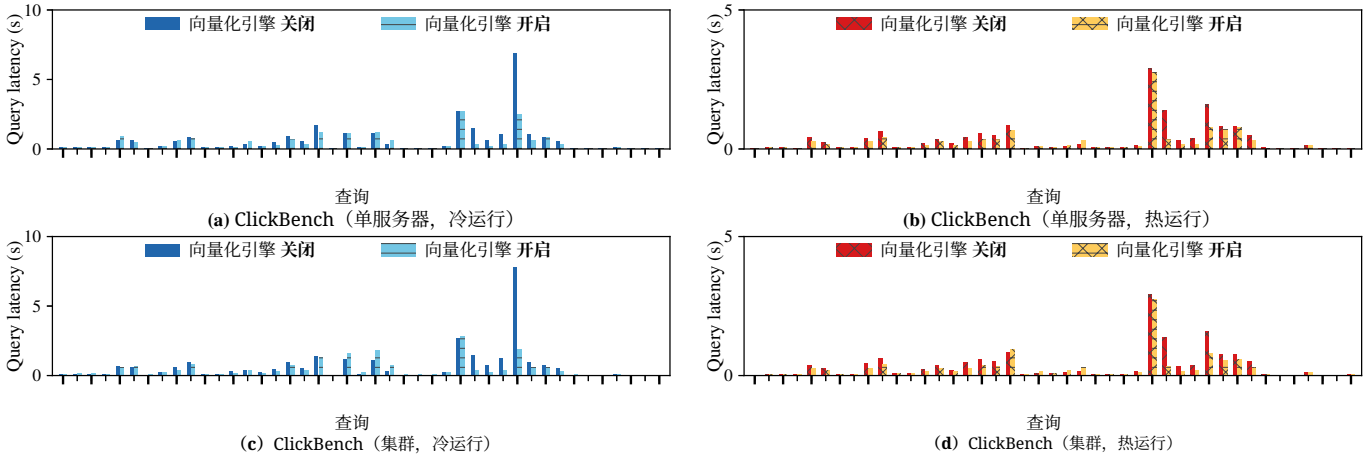


图12. 在 ClickBench 基准测试中，关闭和开启向量化引擎时 OceanBase Mercury 的查询延迟。

E. OceanBase (OB) 与 StarRocks (SR) 和 ClickHouse (CH) 的比较

ClickHouse [25], [26] 和 StarRocks [27]是两款知名的开源列式数据库，专为近实时分析处理而设计。公开基准测试 [28] 表明，ClickHouse 的性能优于 Databricks 和 Snowflake 等商业产品，因此，将 OceanBase 与这两款高性能开源系统进行比较便已足够。尽管它们以高性能著称，OceanBase 4.3.3 的分析处理性能仍优于它们的最新版本。图13和图14展示了在 TPC-H（700GB 和 1TB）及 ClickBench 基准测试中，OceanBase 4.3.3 与 StarRocks 3.3.3 和 ClickHouse 24.10.1 的总查询延迟。对于这两项基准测试，OceanBase 4.3.3 的耗时都少于 StarRocks 和 ClickHouse。

对于 TPC-H基准测试，由于 ClickHouse 不支持 TPC-H 中的所有查询 [26], [29]，且处理其支持的查询需要一个多小时，因此不再对其进行讨论。与 StarRocks 相比，在 700GB TPC-H测试中，OceanBase 在单服务器上将查询总延迟降低了 23.7%，在集群中则降低了 33.8%。此外，在 1TB TPC-H测试中，其查询延迟降幅从单服务器上的 46% 到集群中的 19% 不等。查询效率的这些提升对于近实时分析处理系统至关重要。图15展示了 TPC-H 中各查询的详细延迟。对于 ClickHouse 不支持的查询 [26]，，我们将其横轴标签标为红色，并且不在图中绘制。OceanBase 4.3.3 在大多数查询中的延迟都更低，尤其是在复杂且耗时的查询中。例如，在图15（b）中的查询 18，OceanBase 耗时 6.07 秒，而 StarRocks 耗时 24.3 秒，ClickHouse 耗时 163.0 秒。查询 18 用于获取供应数量超过指定数值的供应商详细信息。该查询涉及三表连接操作，同时执行聚合、分组、排序等操作，并使用 IN 子查询。OceanBase 在该查询中的优势表明，它具备加速复杂查询的能力。

ClickBench 基准测试的性能测试在配备 16核处理器和 32GB DRAM 的服务器上进行。如图14所示，OceanBase 将查询

延迟降低了 68%（冷运行）和 35%（热运行），相较于 StarRocks；相较于 ClickHouse，则降低了 57%（冷运行）和 23%（热运行）。图16展示了 ClickBench 中所有查询的详细查询延迟，OceanBase 的性能表现更为出色。图16 (a) 中的查询24提升最为显著：OceanBase 耗时 4.4 秒，而 StarRocks 耗时 46.5 秒，ClickHouse 耗时 36.2 秒。

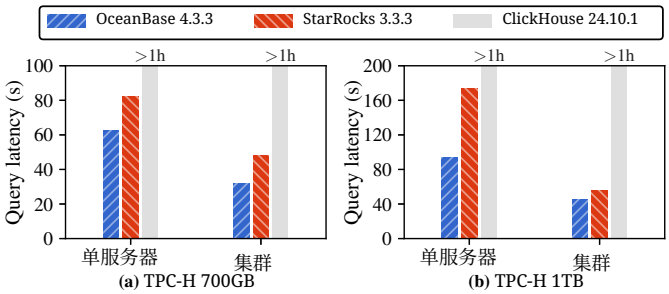


图13. TPC-H基准测试中OceanBase与StarRocks和ClickHouse的总查询延迟。

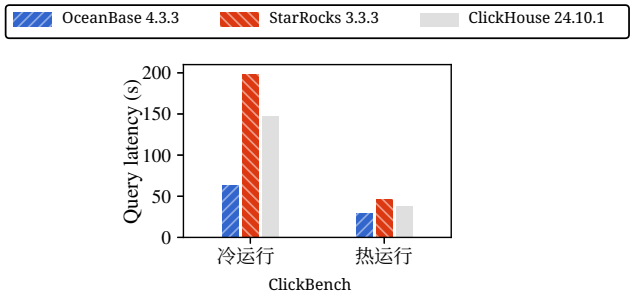


图14. ClickBench基准测试中OceanBase与StarRocks和ClickHouse的总查询延迟。

F. OceanBase 与 Doris 在更新密集型工作负载下的对比

我们基于实际的更新密集型工作负载，对比了 OceanBase Mercury 与 Doris。如图17所示，实验结果展示了 OceanBase 和 Doris 在不同写入密集型工作负载下的性能对比，测量了 18 个查询在写入比例从 0.0 到 0.2 不等时的平均响应时间。OceanBase 始终优于 Doris，平均响应时间约为 853ms，而 Doris 为 4327ms，性能优势约为 5× 。随着写入比例从 0.0 增加到 0.2，平均响应时间从 613ms 大幅上升至 5388ms，表明写入密集型操作会显著降低两个系统的查询性能。该图展示了不同写入类型（插入/更新）和表类型（列、MOW C编码跳过索引）组合下的这些趋势，并以对数刻度显示响应时间，以适应实验中观察到的广泛性能范围。

– – – –

VII. 相关工作

A. 近实时 OLAP

近实时 OLAP 在大数据应用中受到广泛关注，包括通过复杂分析查询，对新添加的数据执行监控和仪表板创建等服务工作负载。OceanBase 的社区用户和商业用户经常面临此类需求，传统上必须依赖多个系统才能满足要求。为缓解这些问题，OceanBase Mercury（即 OceanBase 4.3.3）增强了近实时 OLAP 能力，成为一个高性能、统一且灵活的 OLAP 数据库。OceanBase Mercury 旨在通过单一框架处理分析型工作负载，从而提升效率并减少资源浪费。待处理的数据既可以来自 OceanBase 的在线事务处理系统，也可以来自其他各种来源，包括其他数据库、Kafka、Flink、文件系统等。其列存模式、物化视图和向量化执行引擎方面的创新，显著提升了 OceanBase 的分析性能。借助 OceanBase 4.3.3，用户可以在单一系统中处理大规模分析查询和服务任务。这使开发者能够轻松构建大规模近实时 OLAP 系统，同时支持在线事务处理能力。

OceanBase Mercury 在内存处理、线性扩展和高数据压缩方面表现出色，但在内存依赖和混合工作负载调优方面仍面临挑战。与其他开源 OLAP 数据库相比，Presto [30]支持灵活的多源集成，但缺乏对用户定义函数（UDF）的支持，且难以满足内存需求。ClickHouse [26]利用列式存储和可扩展性实现高效分析，但集群扩展能力有限，维护成本较高。Druid [31] 专注于近实时摄取和多租户，但其 OLAP 性能不稳定，且系统复杂。StarRocks [27]结合向量化执行与 MPP 架构，可实现亚秒级查询，但受存算耦合以及对用户定义函数（UDF）或用户定义表函数（UDTF）支持有限的制约，体现了各系统在性能、可扩展性和生态系统成熟度之间的权衡。

此外，Google 的 Napa [32]通过积极使用基于 LSM 树结构维护的物化视图，支持可扩展数据仓库。Napa 引入了可查询时间戳 (QT)，用于管理数据新鲜度、查询性能和资源成本之间的权衡。为确保查询延迟稳定（方差较低），Napa 中新摄取的数据只有在 QT 前进后才会变得可见；而 QT 可能会延迟推进，直至后台压实或视图维护充分完成。相比之下，Mercury 在查询执行期间将列式基线与基于行的增量数据合并，实现了从零到数毫秒的数据新鲜度（ [18] 中的 A2 系统）。这种设计无需等待压实，即可立即访问已提交事务，在保持分析数据新鲜度的同时，优先保障在线事务处理稳定性。

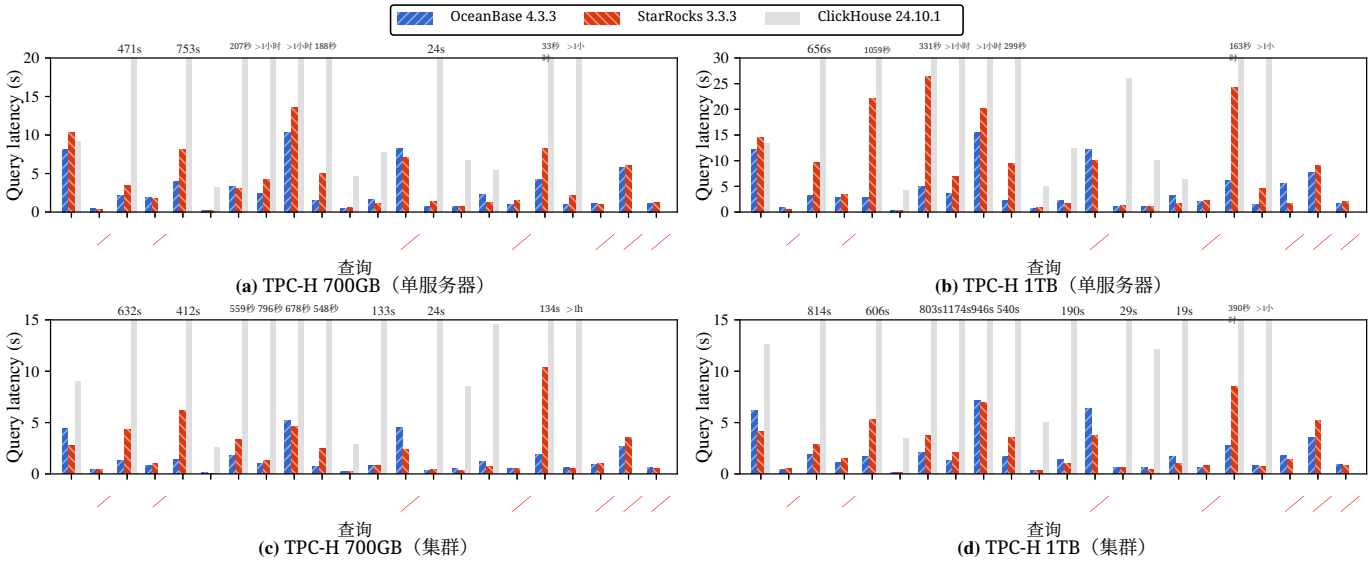


图 15. TPC-H 基准测试中 OceanBase 与 StarRocks 和 ClickHouse 的对比

ClickHouse 不支持以红色突出显示的查询，因此图中未绘制 ClickHouse 的这些查询

.

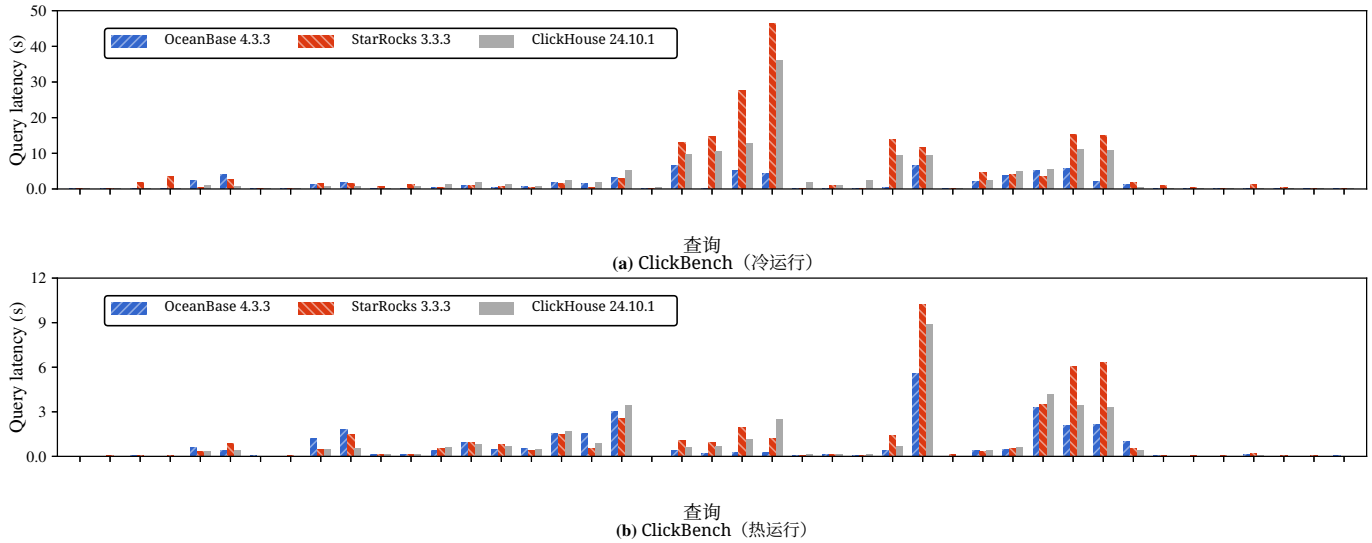


图 16. 单服务器下 OceanBase 与 StarRocks 和 ClickHouse 在 ClickBench 基准测试中的对比。

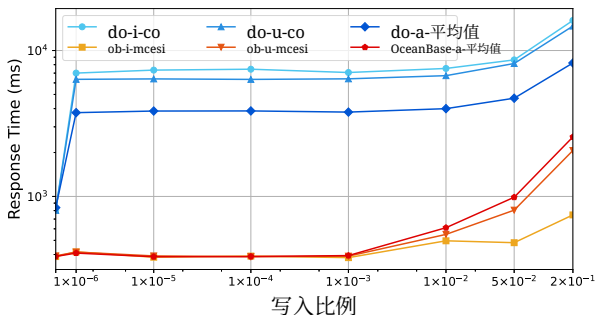


图 17. 更新密集型工作负载下 OceanBase 与 Doris 的对比。

B. 列存

列存凭借其独特的结构和访问模式，非常适合分析任务。C-store [33] 是读优化而非写优化的关系型数据库管理系统，并且

提供了针对不同列的投影方法。HBase [34]是一种基于 HDFS 构建的 NoSQL 分布式数据库系统；HDFS 包含图像等非文本数据，而 HBase 存储这类数据的位置。ClickHouse [25] 是一种高性能、采用 MPP 架构的列存储 OLAP 数据库，它充分利用所有可用的系统资源，尽可能快速地处理每个分析型查询。除了上述原生列存储数据库外，还有一些兼容列存储的数据库。例如，阿里巴巴的 Hologres [6] 实现了行列混合存储架构，其中采用列存来优化列扫描的性能。SQL Server 支持列存储索引，并在 [35] 中概述了列存储索引的设计与实现。在 [36]，中，作者得出结论：即使通过垂直划分模式或为每一列建立索引，使各列能够独立访问，行存储也无法获得列存储的性能优势，而我们

通过垂直划分模式或为每一列建立索引来优化行存储，从而使各列能够独立访问。 [37]通过深入探讨学术界和工业界列式数据库的架构细节，介绍了行存储衰落、列存储兴起的技术演进与历史。

C. 向量化执行引擎

向量化执行引擎已被证明是显著提升分析查询性能的强大机制，主要因为它能够用单条指令处理更大量的数据。向量化执行的概念最早在 [38],中提出，其中关系系统的类 Volcano 的逐元组执行架构被认定为现代 CPU 上 CPU 效率低的原因，而这源于解释开销。为解决这一问题，同一篇论文提出了 MonetDB/X100 查询引擎，它在 MonetDB 之上实现了向量化执行，标志着查询处理技术的关键进展。在这一突破之后，向量化执行因其显著的性能优势而在众多数据库系统中得到广泛采用。例如，CockroachDB [39]同时纳入了传统的逐行执行引擎和向量化执行引擎，并在查询执行期间根据输入数据的特征动态选择最优引擎。该领域的进一步进展体现在 [40], 中，其中为 Spark 上的 SQL 查询处理引入了全面的向量化执行引擎。该引擎优化 CPU 缓存利用率并最小化数据占用，从而进一步提高效率。此外， [41],对向量化执行与以数据为中心的编译式查询执行进行了详细比较，并从计算成本等多种指标评估了二者各自的优缺点。

D. 物化视图

与普通视图不同，物化视图会存储某些耗时查询的结果。在执行相关查询时，可以直接访问这些结果，而无需重复这些耗时操作。由于物化视图带来的便利，许多数据库系统都支持这一特性。Oracle 物化视图 [42] 专为数据仓库和复制而设计，其中利用物化视图进行的优化包括基于成本的选择方法的透明查询重写。IBM 的 DB2 在 2004 [43] 中提出了一种高级工具，可自动选择最优物化视图和索引，以降低维护成本。 [44] 中提出了一种算法，用于在 PostgreSQL 中随着基表中的数据发生变化而对物化视图进行增量更新。更具体地说，该算法设计了一个程序，可自动为触发器生成 PL/pgSQL 代码，从而能够对 PostgreSQL 中的物化视图执行同步增量更新。物化视图领域的三个关键问题在 [45], 和 这三个问题如下：当基表发生变化时如何高效地维护物化视图；如何通过物化视图高效地提升查询性能；如何选择合适的视图进行物化。

E. 更新模型

表 IV 总结了分析处理系统中采用的四种主流更新模型，每种模型都体现了写入性能与读取性能之间的不同权衡。**写时复制**通过维护单一且一致的数据版本实现最优读取性能，但由于每次更新都需要读取并重写整个行集，因而产生较高的写入成本。该模型在数据新鲜度至关重要的读密集型工作负载中表现优异，Delta Lake [46] 和 Snowflake [47]等系统即为例证。**读取时合并** 通过追加更新且不立即整合来最小化写入开销，将合并操作推迟到查询时。虽然这种方法能够实现高效写入，但当增量数据不断累积时，读取性能会下降，因为查询期间需要进行复杂的多版本合并和去重。OceanBase [15], 、TiFlash [48], 和 ClickHouse [26] 等系统采用该模型，优先保证写入吞吐量而非读取延迟。高效的分布式事务处理 [49] 以及死锁检测机制 [50],[51]对于维护此类系统的一致性至关重要。**Delta Store** 采用索引化增量结构来跟踪更新，使写入成本与读取成本均保持在中等水平。然而，当更新频繁时，谓词下推会变得复杂，查询优化效果也会下降。**写时合并（删除+插入）** 代表一种混合方案：它使用删除位图标记删除，同时将新数据追加到独立的行集中，通过高效过滤实现较低的读取成本，同时将写入成本保持在适中水平。该模型支持谓词下推并提供良好的查询优化，因而适用于既要求更新效率又要求查询性能的系统，Polar IMCI [2],、SQL Server CSI [35],和 Hologres [52]便是例证。

VIII. 经验与未来创新

我们总结了开发 OceanBase Mercury 过程中积累的经验，以期为未来的联机分析处理创新提供参考。

经验 1. 同时支持多种存储模型，能为用户提供更多选择，但也给优化器带来了更大的挑战。因此，需要更深入地理解和评估不同模型中的各类算子成本，才能选择更有效的执行计划。

经验 2. 增量数据过多会显著影响查询响应时间，而频繁压实则会大量消耗计算资源，并对用户操作造成不利影响。为通过压实优化查询性能，同时不干扰业务运营，需要加强监控和反馈机制。

第 3 课。刷新全量或增量物化视图通常需要处理大量数据。

表 IV AP 系统中的更新模型

模型	写入 Cost	写入过程	Read Cost	读取过程	预测推送。	查询 Opt.	系统
写时复制	High	读取基表，全部修改，写入新数据行集	None	None	Yes	Good	Hudi CoW [53], Delta Lake, Snowflake
读取时合并	None	None	High	读取多版本行，合并并去重	复杂	较差（增量较大）	Hudi MoR [53], StarRocks Unique, OceanBase、TiFlash、ClickHouse
Delta Store	Med.	通过索引查找行，向上-日期增量	Med.	合并基表与增量方式行号	复杂	较差（频繁更新）	Apache Kudu [54]
写时合并	Med.	查找行 通过索引，标记删除，写入新数据	Low	通过删除位图过滤已删除行位图	Yes	Good	SQL Server CSI、ADB、Polar IMCI, Hologres、StarRocks PK

因此，必须利用并行 DML 和快速批量数据加载技术来提升刷新速度。

经验 4. 每次刷新物化视图时，都会产生大量行删除。采用 TTL（生存时间）等高效方法来有效管理这些删除至关重要。

经验 5. 为了进一步加快计算，计算引擎应考虑基于编码数据（例如字典数据）执行表达式、聚合、连接、排序及其他操作。

经验 6. ARM 芯片具有更高的整体性价比，因此还需要进一步考虑如何在 ARM 环境中优化计算引擎。

IX. 结论

总之，快速演进的大数据领域亟需能够无缝处理海量数据，同时支持事务处理、近实时服务和强大分析能力的数据库系统。传统 OLAP 系统往往难以满足这些需求，因此近实时分析处理系统应运而生并得到广泛采用。本文介绍了 OceanBase Mercury 数据库系统，该系统旨在 PB 级规模下运行，并具备近实时 OLAP 能力。OceanBase Mercury 解决了在分布式事务中维护 ACID 属性、执行复杂分析查询以及提供高可用性、可扩展性和强大性能等方面的挑战。所提出的系统提供灵活的混合列存储模式、高效的物化视图刷新机制、适用于向量化引擎的三种数据格式，并在各种工作负载下展现出卓越性能，彰显了 OceanBase Mercury 在满足企业对高效数据处理和分析不断变化的需求方面的优势。

致谢

我们谨向匿名评审者致以谢意，感谢他们提出的真知灼见，也感谢同事们的贡献。源代码可在 https://github.com/oceanbase/oceanbase/tree/4.3.3 获取。

参考文献

[1] A. Kemper 和 T. Neumann，《再次一刀切！混合型 OLTP&OLAP 数据库管理系统 HyPer 的架构》，载于 实现实时商业智能——第四届国际研讨会 *BIRTE 2010*，举办于第*36*届超大型数据库国际会议 *VLDB 2010*，*2010*年*9*月*13*日，新加坡，修订版精选论文，丛书《商业信息处理讲义》，M. Castellanos、U. Dayal 和 V. Markl 编，第84卷。施普林格，2010年，第7–23页。[2] W. Cao、Y. Liu、Z. Cheng、N. Zheng、W. Li、W. Wu、L. Ouyang、P. Wang、Y. Wang、R. Kuan、Z. Liu、F. Zhu 和 T. Zhang，《POLARDB 遇上计算存储：在云原生关系数据库中高效支持分析型工作负载》，载于 第*18*届 *USENIX* 文件与存储技术会议，*FAST 2020*，*2020*年*2*月*24*日至*27*日，美国加利福尼亚州圣克拉拉，S. H. Noh 和 B. Welch 编。USENIX 协会，2020年，第29–41页。[3] Z. Lyu、H. H. Zhang、G. Xiong、G. Guo、H. Wang、J. Chen、A. Praveen、Y. Yang、X. Gao、A. Wang、W. Lin、A. Agrawal、J. Yang、H. Wu、X. Li、F. Guo、J. Wu、J. Zhang 和 V. Raghavan，《Greenplum：面向事务型和分析型工作负载的混合数据库》，载于*SIGMOD '21*：数据管理国际会议，线上会议，中国，*2021*年*6*月*20*日至*25*日，G. Li、Z. Li、S. Idreos 和 D. Srivastava 编。ACM，2021年，第2530–2542页。[4] C. Wei、B. Wu、S. Wang、R. Lou、C. Zhan、F. Li 和 Y. Cai，《AnalyticDB-v：面向查询融合的混合分析引擎，用于结构化和非结构化数据》，《*VLDB*学报》，第13卷，2020年，第3165、3152页，第12期。[5] B. Chattopadhyay、P. Dutta、W. Liu、O. Tinn、A. McCormick、A. Mokashi、P. Harvey、H. Gonzalez、D. Lomax、S. Mittal、R. Ebenstein、N. Mikhaylin、H. Lee、X. Zhao、T. Xu、L. Perez、F. Shahmohammadi、T. Bui、N. Mckay、S. Aya、V. Lychagina 和 B. Elliott，《Procella：统一 YouTube 上的服务与分析数据》，《*VLDB*学报》，第12卷，第12期，第2022–2034页，2019年。[6] X. Jiang、Y. Hu、Y. Xiang、G. Jiang、X. Jin、C. Xia、W. Jiang、J. Yu、H. Wang、Y. Jiang、J. Ma、L. Su 和 K. Zeng，《阿里云 Hologres：用于混合服务与分析处理的云原生服务》，《*VLDB*学报》，第13卷，第12期，第3272–3284页，2020年。[7] J. Chen、R. Shi、H. Chen、L. Zhang、R. Li、W. Ding、L. Fan、H. Wang、M. Xiong、Y. Chen、B. Dong、K. Guo、Y. Lin、X. Liu、H. Shi、P. Wang、Z. Wang、Y. Yang、J. Zhao、D. Zhou、Z. Zuo 和 Y. Liang，《Krypton：字节跳动的实时服务与分析型 SQL 引擎》，《*VLDB*学报》，第16卷，第12期，第3528–3542页，2023年。[8]《OceanBase》，https://github.com/oceanbase，2024年。[9] Z. Yang、C. Yang、F. Han、M. Zhuang、B. Yang、Z. Yang、X. Cheng、Y. Zhao、W. Shi、H. Xi、H. Yu、B. Liu、Y. Pan、B. Yin、J. Chen 和 Q. Xu，《OceanBase：一个每分钟处理7.07亿事务的分布式关系数据库系统》，《*VLDB*学报》，第15卷，第12期，第3385–3397页，2022年。[10]《TPCC-OceanBase》，https://tpc.org/results/fdr/tpcc/ant financial~tpcc~ alibaba cloud elastic compute service cluster~fdr~2020-05- 17~v01.pdf。[11] Q. Xu、W. Sun、C. Yang、J. Liu、Z. Wei、F. Han、L. Wang 和 X. Zhai，《OceanBase 单元化：构建下一代在线地图应用》，载于 *2025*年 *IEEE* 第*41*届数据工程国际会议（*ICDE*）。IEEE，2025年，第4183–4196页。[12] Q. Zhang、J. Li、H. Zhao、Q. Xu、W. Lu、J. Xiao、F. Han、C. Yang 和 X. Du，《异构网络中的高效分布式事务处理》，《*VLDB*学报》，第16卷，第6期，第1372–1385页，2023年。

[13] 《木兰公共许可证 2.0》，https://license.coscl.org.cn/MulanPubL-2.0，2021年。[14] 《OceanBase》，https://gitee.com/oceanbase，2024年。[15] Z. Yang、Q. Xu、S. Gao、C. Yang、G. Wang、Y. Zhao、F. Kong、H. Liu、W. Wang 和 J. Xiao，《Oceanbase Paetica：支持单机和分布式集群的混合共享无任何内容/共享全部内容数据库》，《VLDB学报》，第16卷，第3740、3728页，2023年，第12期。[16] F. Han、H. Liu、B. Chen、D. Jia、J. Zhou、X. Teng、C. Yang、H. Xi、W. Tian、S. Tao、S. Wang、Q. Xu 和 Z. Yang，《PALF：分布式数据库的复制预写日志》，《VLDB学报》，第17卷，第12期，第3745–3758页，2024年。[17] C. Fang、C. Qian、Q. Yang、Z. Wang、Z. Yang、F. Kong、Q. Xu、H. Cao、F. Han 和 C. Yang，《Malt：OceanBase 大事务管理框架》，载于《2025年数据管理国际会议伴随论文集》，2025年，第378–390页。[18] C. Zhang、G. Li 和 T. Lv，《Hybench：一种面向 HTAP数据库的新型基准测试》，《VLDB学报》，第17卷，第5期，第939–951页，2024年。[19] 《OceanBase 迁移服务（OMS）》，https://en.oceanbase.com/docs/oms-en，2026年。[20] G. Moerkotte，《小型物化聚合：一种用于数据仓库的轻量级索引结构》，载于 VLDB'98，《第24届超大型数据库国际会议论文集》，1998年8月24日至27日，美国纽约州纽约市，A. Gupta、O. Shmueli 和 J. Widom 编。Morgan Kaufmann，1998年，第476–487页。[21] G. Graefe，《快速加载与快速查询》，载于《数据仓库与知识发现》，第11届国际会议 DaWaK 2009，奥地利林茨，2009年8月31日至9月2日，论文集，丛书《计算机科学讲义》，T. B. Pedersen、M. K. Mohania 和 A. M. Tjoa 编，第5691卷。施普林格，2009年，第111–124页。[22] 《TPC-H》，https://www.tpc.org/tpch/，2024年。[23] 《TPC-DS》，https://www.tpc.org/tpcds/，2024年。[24] 《ClickBench》，https://benchmark.clickhouse.com/，2024年。[25] （2016）ClickHouse。[在线]。可访问：https://clickhouse.com/ [26] R. Schulze、T. Schreiber、I. Yatsishin、R. Dahimene 和 A. Milovidov，《ClickHouse-Lightning：人人可用的快速分析》，《VLDB学报》，第17卷，第12期，第3731–3744页，2024年。[27] 《 StarRocks 》，https://www.starrocks.io/，2024 年。[28] 《 ClickHouse 与 Databricks、Snowflake 的比较》，https://benchmark.clickhouse.com/，2025年。[29] 《TPC-H（1999）—— ClickHouse 文档》，https://clickhouse.com/docs/en/getting-started/example-datasets/tpch，2024年。[30] R. Sethi、M. Traverso、D. Sundstrom、D. Phillips、W. Xie、Y. Sun、N. Yegitbasi、H. Jin、E. Hwang、N. Shingte 和 C. Berner，《Presto：一切之上的 SQL》，载于 2019 IEEE 第35届数据工程国际会议（ICDE），2019年，第1802–1813页。[31] F. Yang、E. Tschetter、X. L´eaut´e、N. Ray、G. Merlino 和 D. Ganguli，《Druid：实时分析型数据存储》，载于《2014年 ACM SIGMOD 数据管理国际会议论文集》，丛书 SIGMOD '14，美国纽约州纽约市：计算机协会，2014年，第157–168页。[在线]。可访问：https://doi.org/10 1145 2588555 2595631 。/。A. Agiwal、[32] K. Lai、G. N. B. Manoharan、I. Roy、J. Sankaranarayanan、H. Zhang、T. Zou、M. Chen、Z. Chen、M. Dai 等，《Napa：借助 Google 的稳健查询性能实现可扩展数据仓库》，《VLDB学报》，第14卷，第2997、2986页，2021年，第12期。[33] M. Stonebraker、D. J. Abadi、A. Batkin、X. Chen、M. Cherniack、M. Ferreira、E. Lau、A. Lin、S. Madden、E. O'Neil 等，《C-store：面向列的数据库管理系统》，载于《让数据库运行起来：Michael Stonebraker 的实践智慧》，2018年，第491–518页。[34] M. N. Vora，《面向大规模数据的 Hadoop-HBase》，载于《2011年计算机科学与网络技术国际会议论文集》，第1卷。IEEE，2011年，第601–605页。[35] P.-°A. Larson、C. Clinciu、E. N. Hanson、A. Oks、S. L. Price、S. Ran- garajan、A. Surna 和 Q. Zhou，《SQL Server 列存储索引》，载于《2011年 ACM SIGMOD 数据管理国际会议论文集》，2011年，第1177–1184页。[36] D. J. Abadi、S. R. Madden 和 N. Hachem，《列存储与行存储：它们究竟有多大差异？》，载于《2008年 ACM SIGMOD 数据管理国际会议论文集》，2008年，第967–980页。[37] K. Sridhar，《面向大数据处理的现代列存储》，载于《大数据分析：第五届国际会议》，BDA 2017，印度海得拉巴，2017年12月12日至15日，论文集第5卷。施普林格，2017年，第113–125页。[38] P. A. Boncz、M. Zukowski 和 N. Nes，《Monetdb/x100：超流水线查询执行》，载于 Cidr，第5卷，2005年，第225–237页。[39] R. Taft、I. Sharif、A. Matei、N. VanBenschoten、J. Lewis、T. Grieger、K. Niemi、A. Woods、A. Birzin、R. Poss 等，《CockroachDB：具有弹性的地理分布式 SQL 数据库》，载于《2020年 ACM SIGMOD 数据管理国际会议论文集》，2020年，第1493–1509页。[40]Y. Shen、J. Xiong 和 D. Jiang，《使用向量化执行提升 Spark 上的 SQL 查询性能》，载于《第50届国际并行处理会议论文集》，2021年，第1–11页。[41] T. Kersten、V. Leis、A. Kemper、T. Neumann、A. Pavlo 和 P. Boncz，《关于编译型和向量化查询，你一直想知道却不敢问的一切》，《VLDB学报》，第11卷，第13期，第2209–2222页，2018年。[42] R. G. Bello、K. Dias、A. Downing、J. Feenan、J. Finnerty、W. D. Norcott、H. Sun、A. Witkowski 和 M. Ziauddin，《Oracle 中的物化视图》，载于 VLDB，第98卷，1998年，第24–27页。[43] D. C. Zilio、C. Zuzarte、S. Lightstone、W. Ma、G. M. Lohman、R. J. Cochrane、H. Pirahesh、L. Colby、J. Gryz、E. Alton 等，《使用 IBM DB2 设计顾问推荐物化视图和索引》，载于《国际自主计算会议论文集》。IEEE，2004年，第180–187页。[44] N. T. Quoc Vinh，《PostgreSQL 物化视图的同步增量更新》，《程序设计与计算机软件》，第42卷，第5期，第307–315页，2016年。[45] R. Chirkova、J. Yang 等，《物化视图》，《数据库基础与趋势》，第4卷，第4期，第295–405页，2012年。[46] M. Armbrust、T. Das、S. Paranjpye、R. Xin、S. Zhu、A. Ghodsi、B. Yavuz、M. Murthy、J. Torres、L. Sun、P. Boncz、M. Mokhtar、H. V. Hovell、A. Ionescu、A. Luszczak、M. Switakowski、T. Ueshin、X. Li、M. Szafran-ski、P. Senster 和 M. Zaharia，《Delta Lake：基于云对象存储的高性能 ACID 表存储》，《VLDB学报》，第13卷，第12期，第3411–3424页，2020年。[47] B. Dageville、T. Cruanes、M. Zukowski、V. Antonov、A. Avanes、J. Bock、J. Claybaugh、D. Engovatov、M. Hentschel、J. Huang 等，《Snowflake 弹性数据仓库》，载于《2016年 ACM SIGMOD 数据管理国际会议论文集》。ACM，2016年，第215–226页。[48] D. Huang、Q. Liu、Q. Cui、Z. Fang、X. Ma、F. Xu、L. Shen、L. Tang、Y. Zhou、M. Huang 等，《TiDB：基于 Raft 的 HTAP 数据库》，《VLDB学报》，第13卷，第12期，第3072–3084页，2020年。[49] H. Zhao、W. Lu、Z. Zhao、Y. Hong、Q. Xu、J. Xiao、F. Han、C. Yang 和 X. Du，《一种面向异构网络的高效两轮分布式事务处理方法：H. Zhao 等》，《VLDB期刊》，第34卷，第4期，第51页，2025年。[50] Z. Yang、C. Qian、X. Teng、F. Kong、F. Han 和 Q. Xu，《LCL：一种基于锁链长度的分布式死锁检测与解决算法》，载于《2023年 IEEE 第39届国际数据工程会议（ICDE）》。IEEE，2023年，第151–163页。[51] Z. Yang、C. Qian、X. Teng、F. Kong、F. Han、Q. Xu 和 D. Hu，《LCL+：为 OceanBase 构建的基于锁链长度的分布式死锁检测与解决服务》，《ACM计算机系统汇刊》，第43卷，第4期，第1–33页，2025年。[52] X. Jiang、Y. Hu、Y. Xiang、G. Jiang、X. Jin、C. Xia、W. Jiang、J. Yu、H. Wang、Y. Jiang 等，《阿里云 Hologres：用于混合服务与分析处理的云原生服务》，《VLDB学报》，第13卷，第12期，第3272–3284页，2020年。[53] 《Apache Hudi》，https://hudi.apache.org/，2026年。[54] T. Lipcon、D. Alves、D. Burkert、J.-D. Cryans、A. Dembo、M. Percy、S. Rus、D. Wang、M. Bertozzi、C. P. McCabe 等，《Kudu：面向高速数据的快速分析存储》，Cloudera 公司，第28卷，第36–77页，2015年。