软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2024,35(6):2951−2973 [doi: 10.13328/j.cnki.jos.006901]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# HTAP 数据库系统数据共享模型和优化策略*

胡梓锐[1,2], 翁思扬[1,2], 王清帅[1,2], 俞 融[1,2], 徐金凯[1,2], 张 蓉[1,2], 周 烜[1,2]

[1](华东师范大学 上海市大数据管理系统工程研究中心, 上海 200062)

[2](华东师范大学 数据科学与工程学院, 上海 200062)

通信作者: 张蓉, E-mail: rzhang@dase.ecnu.edu.cn

**摘 要**: 混合事务与分析处理数据库系统 (HTAP) 因其在一套系统上可以同时处理混合负载而逐渐获得大众认可. 为了不影响在线事务处理 (OLTP) 业务的写入性能, HTAP 数据库系统往往会通过维护数据多版本或额外副本的方式来支持在线分析处理 (OLAP) 任务, 从而引入了 TP/AP 端版本的数据一致性问题. 同时, HTAP 数据库系统面临资源隔离下实现高效数据共享的核心挑战, 且数据共享模型的设计综合权衡了业务对性能和数据新鲜度之间的要求. 因此, 为了系统地阐释现有 HTAP 数据库系统数据共享模型及优化策略, 首先根据 TP 生成版本与 AP 查询版本的差异, 通过一致性模型定义数据共享模型, 将 HTAP 数据共享的一致性模型分为 3 类, 分别为线性一致性, 顺序一致性与会话一致性. 然后, 梳理数据共享模型的全流程, 即从数据版本标识号分配, 数据版本同步, 数据版本追踪 3 个核心问题出发, 给出不同一致性模型的实现方法. 进一步, 以典型的 HTAP 数据库系统为例对具体实现进行深入的阐释. 最后, 针对数据共享过程中涉及的版本同步、追踪、回收等模块的优化策略进行归纳和分析, 并展望数据共享模型的优化方向, 指出数据同步范围自适应, 数据同步周期自调优和顺序一致性的新鲜度阈值约束控制是提高 HTAP 数据库系统性能和新鲜度的可能手段.

**关键词**: HTAP 数据库系统; 一致性模型; 数据管理; 混合负载; 性能优化

**中图法分类号**: TP311

中文引用格式: 胡梓锐, 翁思扬, 王清帅, 俞融, 徐金凯, 张蓉, 周烜. HTAP数据库系统数据共享模型和优化策略. 软件学报, 2024, 35(6): 2951–2973. http://www.jos.org.cn/1000-9825/6901.htm

英文引用格式: Hu ZR, Weng SY, Wang QS, Yu R, Xu JK, Zhang R, Zhou X. Data Sharing Model and Optimization Strategies in HTAP Database Systems. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2951–2973 (in Chinese). http://www.jos.org.cn/1000-9825/6901.htm

## Data Sharing Model and Optimization Strategies in HTAP Database Systems

HU Zi-Rui[1,2], WENG Si-Yang[1,2], WANG Qing-Shuai[1,2], YU Rong[1,2], XU Jin-Kai[1,2], ZHANG Rong[1,2], ZHOU Xuan[1,2]

[1](Shanghai Engineering Research Center of Big Data Management, East China Normal University, Shanghai 200062, China)

[2](School of Data Science and Engineering, East China Normal University, Shanghai 200062, China)

**Abstract**: Hybrid transactional/analytical processing (HTAP) database systems have gained extensive acknowledgment of users due to their full processing support of the mixed workloads in one system, i.e., transactions and analytical queries. Most HTAP database systems tend to maintain multiple data versions or additional replicas to accomplish online analytical processing (OLAP) without downgrading the write performance of online transactional processing (OLTP). This leads to a consistency problem between the data of TP and AP versions. Meanwhile, HTAP database systems face the core challenge of achieving efficient data sharing under resource isolation, and the data-sharing model integrates the trade-off between business requirements for performance and data freshness. To systematically explain the data-sharing model and optimization strategies of existing HTAP database systems, this study first utilizes the consistency models to define the data-sharing model and classify the consistency models for HTAP data sharing into three categories, namely, linear consistency,

* 基金项目: 国家自然科学基金 (62072179); 2021 CCF-华为数据库创新研究计划
收稿时间: 2022-09-18; 修改时间: 2022-11-11; 采用时间: 2023-01-05; jos 在线出版时间: 2023-07-05
CNKI 网络首发时间: 2023-07-07

sequential consistency, and session consistency, according to the differences between TP generated versions and AP query versions. After that, it takes a deep dive into the whole process of data-sharing models from three core issues, i.e., data-version number distribution, data version synchronization, and data version tracking, and provides the implementation methods of different consistency models. Furthermore, this study takes a dozen of classic and popular HTAP database systems as examples for an in-depth interpretation of the implementation methods. Finally, it summarizes and analyzes the optimization strategies of version synchronization, tracking, and recycling modules involved in the data-sharing process and predicts the optimization directions of the data-sharing models. It is concluded that the self-adaptability of the data synchronization scope, self-tuning of the data synchronization cycle, and freshness-bound constraint control under sequential consistency are the possible means for better performance of HTAP database systems and higher freshness.

**Key words**: HTAP database system; consistency model; data management; hybrid workload; performance optimization

## 1 引 言

### 1.1 HTAP 数据库系统

Gartner 于 2014 年给出了 HTAP 数据库系统的定义[1,2], 即支持在同一套数据库系统中同时进行事务处理 OLTP 和实时分析混合负载 (OLAP) 的数据库系统称为 HTAP 数据库系统. HTAP 数据库系统旨在实时分析事务最新更新的数据. 在面对业务需要同时支持混合负载处理的场景下, 与原先的事务处理 (TP) 和分析处理 (AP) 系统独立的解决方案相比, HTAP 数据库系统具有以下的优势.

- 兼备事务处理和分析查询的处理支持能力.
- 数据同步速度快, 支持实时分析.
- 部署和维护成本低.

在 HTAP 这一概念提出前 Hyrise[3]、SAP HANA[4]和 HyPer[5,6]等单机数据库在 2010 年左右便已经开始探索基于多种存储结构来同时支持事务处理和分析查询. 到了 2015 年, HTAP 概念的提出之际, Oracle[7]和 SQL Server[8]等传统数据库厂商便通过在原有的架构设计上进行扩展以支持 HTAP 业务. 同时随着分布式系统理论的不断延拓, 又涌现出诸如 Greenplum[9]、SingleStore[10]、FoundationDB[11]等一批在分布式环境下支持高效的混合负载处理能力的分布式数据库系统. 近年来, HTAP 概念的再度强化也使得越来越多的数据库厂商, 如 PingCAP 的 TiDB[12], Google 的 F1 Lightning[13], 阿里巴巴的 PolarDB[14]和 Amazon 的 Aurora[15]等都开始以支持 HTAP 业务作为其产品的重要标签, 在学术界也涌现出了诸多面向 HTAP 数据库系统的架构设计以及优化方案[16–18].

针对不同的应用场景, HTAP 数据库系统具有不同的应用目标, 相应地出现了不同架构的 HTAP 数据库系统. 比如, 面向银行等金融场景的实时监控业务, AP 任务对数据的实时性要求高, 希望 AP 端尽可能获取 TP 端写入的最新数据, 进而保证分析的有效性; 在电商等互联网业务的秒杀和促销等场景上, AP 端则可以容忍一定的实时性缺陷换取 TP 端更好的吞吐和业务扩展性. 因此, 与传统数据库相比, HTAP 数据库系统更关注处理性能与数据新鲜度之间的权衡问题[19]. 但无论 HTAP 数据库系统如何权衡两者, 需要保证基本的事务处理性能要求, 如若缺乏对 TP 端吞吐的基本保证 (数据生成端), 用户的业务 SLA (service level agreement) 就无法保证[20,21]. 也就是说, 在保证基本 TP 能力的情况下, 可以通过损失一方的高性能, 提升另一方的性能. 所以 HTAP 数据库系统的目标通常可以归纳为在保证基本的 TP 业务性能要求情况下, 实现 AP 业务对新鲜数据的访问以及高吞吐分析处理能力, 即支持实时分析[22,23].

### 1.2 资源隔离与数据共享

HTAP 数据库系统与传统的 OLTP/OLAP 数据库系统的最大区别就在于其在同一套系统里实现高效的混合负载处理能力, 即在提供稳定、高效 TP 处理能力的同时, 实现快速、及时的 AP 分析处理能力. 对于追求一体化、整体化的 HTAP 数据库系统来说, 其架构设计上存在两个核心难点问题.

- 资源隔离: 如何调度或者隔离 TP 负载和 AP 负载对资源的抢占, 如 CPU、缓存、磁盘 IO 等.
- 数据共享[24]: AP 负载如何访问 TP 负载生产出的新鲜数据版本, 如基于同一套单副本存储、多副本间数据

拷贝/传输等. 这一概念抽象地构建了 TP 数据生产至 AP 数据消费的全流程.

1.2.1 资源隔离

TP 负载通常具有在短时间内频繁读写小规模数据的特征, 对缓存的命中率和磁盘/网络 IO 的时延要求较高; AP 负载往往需要执行大规模数据连接、聚合等计算密集型操作, 对 CPU/内存/磁盘等资源消耗较大. 因此由于 TP/AP 负载特性的不同, 并发执行两种负载时, 往往会对系统资源产生争用, 从而对性能造成较大的影响[23,25]. 为了避免两者的资源竞争, 实现稳定的服务保障, HTAP 数据库系统需要对 TP/AP 业务进行资源上的隔离. 目前, 资源隔离主要有 3 种实现模式, 具体如图 1 所示.

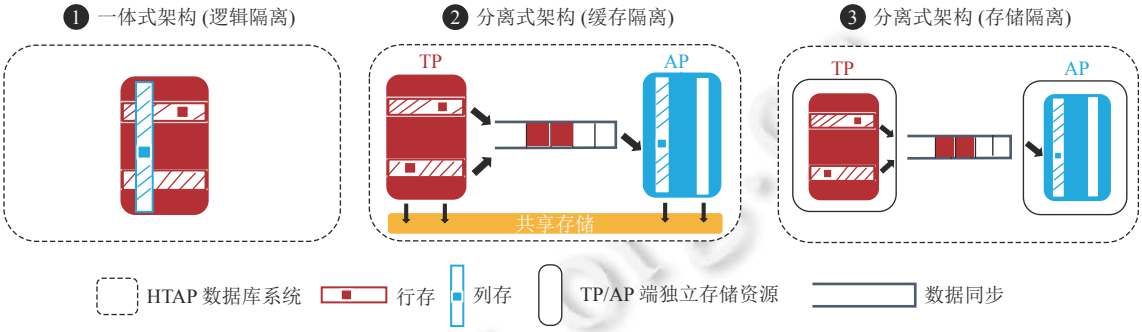



图 1 基于资源隔离的 HTAP 数据库系统分类

(1) 逻辑隔离 (unified storage): 在一体式架构下, 通过逻辑隔离基于同一套资源同时服务 AP 负载和 TP 负载, 如 HyPer[5,6]、MemSQL[10].

(2) 缓存隔离 (share storage): 分离式架构下基于缓存层隔离, 即 TP/AP 端共享一套底层存储资源, 而在缓存层上进行资源隔离, 如 PolarDB[14]、Aurora[15].

(3) 存储隔离 (share nothing): 分离式架构下基于存储层隔离, 即 TP/AP 端的物理资源不再共享, 各自拥有独立的存储引擎和计算引擎, 如 TiDB[12].

1.2.2 数据共享

无论使用何种资源隔离方式, 为了 AP 访问新鲜数据, 系统皆需要支持高效的数据共享, 即 TP 端生成的数据能通过某种方式被 AP 端快速访问到, 如基于同一套单副本存储[5]、多副本间数据拷贝[3,4]/传输[12,13]等. 然而对于 HTAP 数据库系统来说, 资源隔离程度越高, 数据共享的难度越大, 所以如何在资源隔离的情况下高效地实现数据共享是 HTAP 数据库系统需要解决的核心问题之一. 具体地, 以上 3 种资源隔离方案会面临不同的数据共享挑战.

逻辑隔离 (unified storage): 在一体式架构下, 数据库可以通过类似租户或虚拟化等方法实现 TP 任务和 AP 任务在单系统上的逻辑隔离. 通过共享单个存储引擎实现混合负载的数据共享, 即在事务层上使用多版本并发控制机制 (MVCC)[26]共享最新的快照给 AP 任务, 或是通过写时复制 (copy-on-write, COW) 为 AP 任务创建最新的物理快照[27]. 其中, MVCC 通过维护数据的多个物理版本, 允许数据旧版本读, 而不阻塞 TP 并发写入, 该设计理念受到 HTAP 数据库系统的青睐[28]. 但 TP 端的高吞吐, 可能造成短时间内系统积累大量的版本, 以版本链的方式组织数据对 TP 的点读操作较友好, 但会严重影响 AP 大范围扫描性能. 同时, 运行时间较长的 AP 事务也会阻塞过期版本回收, 从而影响 AP 的扫描性能和 TP 处理性能[29]. 另外, 采用 COW 方式进行数据共享, 若 AP 并行度大, 会占用大量内存, 易对 TP 的性能造成负面影响. 故而, 如何根据 TP/AP 负载访问模式的差异设计一体式的版本组织方式是逻辑隔离下数据共享面临的主要挑战.

缓存隔离 (share storage): 缓存资源隔离架构可以避免缓存层及以上的资源冲突. 混合负载共享一套存储引擎, 通过 AP 端独立维护缓存快照的方式将数据共享给 AP 任务, 并允许多个 AP 端维护不同快照的数据版本, 具有较高的读取效率. 但该方法会导致缓存层与存储层数据不一致, 故而如何从数据不一致的缓存层和存储层中追踪全局一致的快照版本是该隔离方式面临的主要挑战.

存储隔离 (share nothing): 存储隔离架构为了实现数据共享, TP 端的存储引擎需要自动将日志或数据同步给

AP 端的存储引擎. 这种方案类似 ETL 式的物理隔离, 试图避免两者之间的资源冲突. 但 AP 端对新鲜数据的需求与 TP 端的高吞吐 (大量数据更新) 制约了两个存储引擎的数据一致性. 故而, 如何实现在完全解耦的存储下实现高效的数据同步是一个核心挑战.

针对这 3 种资源隔离方案引入的版本同步和追踪的挑战, HTAP 数据库系统需要设计有效的数据共享方式, 如为了解决逻辑隔离的挑战, 数据追踪时往往对存储格式进行转化; 为了解决缓存隔离的挑战, 数据版本追踪时需要对齐不一致的版本; 为了解决存储隔离的挑战, 数据同步时可以阶段性同步数据. 基于对现有 HTAP 数据库系统的数据共享方式的总结, 本文从 TP 端到 AP 端的数据生产、同步、追踪的整个生命周期出发, 归纳了不同的数据共享模型, 其约束了 AP 端访问 TP 端产生的数据版本的模式. 进一步, 我们探讨了数据共享模型下的主流优化手段.

### 1.3 相关工作

随着 HTAP 数据库系统的发展, 出现了一批综述类工作对其技术进行概括和总结. Özcan 等人[30]从实现的角度整理了 HTAP 数据库系统的发展历史, 并着重分析、比较了不同的存储架构, 包括行列混存和分离式两套存储的具体实现方式. Hieber 等人[31]则以分布式架构为中心, 总结了 HTAP 数据库系统的设计要点和具体实现, 讨论分布式场景下的并发控制、故障恢复、垃圾回收等问题. 另一方面, Psaroudakis 等人[32]、Raza 等人[22]主要分析了吞吐性能和新鲜度之间的相互影响, 前者设计了针对 HTAP 数据库系统的混合评测负载, 并通过实验定量分析其对彼此的影响, 后者在实验的基础上, 进一步讨论资源隔离问题, 并给出一套资源调度算法. 张超等人[33,34]根据存储结构对 HTAP 数据库系统进行分类, 总结了 HTAP 不同于 OLTP/OLAP 数据库系统在诸如内存与磁盘的行/列存异构存储设计、查询优化、资源调度等技术上的关键要点, 并对比了近 10 年内典型的支持行列共存的 HTAP 数据库系统, 深入地分析了 HTAP 数据库系统的实现技术及不同设计的优劣. 而本文侧重于对不同一致性模型下数据共享方式的分析, 旨在为不同应用场景下性能稳定的 HTAP 数据库系统研发提供思路. 文献 [35] 也展望了机器学习技术在面向 HTAP 数据库系统中存储模型自动化选取、混合负载资源调度等方向上的未来应用. 这些工作都从设计架构和实现效果两个方面综述 HTAP 数据库系统, 但都缺乏对 HTAP 数据库系统如何在天然的架构选择下, 实现有效的数据共享策略的总结和分析.

### 1.4 论文结构

本文第 2 节从一致性模型的角度出发, 定义了数据共享模型, 归纳 HTAP 数据库系统在不同一致性级别下的数据共享模型实现机制和实现特征, 随后围绕业界主流的十余款 HTAP 数据库系统, 进行数据共享模型方案的案例分析. 第 3 节概述了现阶段 HTAP 数据库系统为了提高数据共享时版本同步、追踪、回收的效率和减少对应开销所提出的优化方案. 第 4 节基于对数据共享模型的归纳, 分析并展望了 HTAP 数据库系统未来可能的优化方向.

## 2 HTAP 数据库系统数据共享模型

### 2.1 数据共享模型

数据版本是 TP 任务和 AP 任务数据共享的基础, 系统中事务每次写入产生的新数据会被唯一标识, 该标识文中统一称为版本标识号. 为了深入剖析 HTAP 数据库系统数据共享模型, 本节首先给出数据版本的定义; 进一步, 根据 TP 端版本更新操作和 AP 端版本读取操作在版本标识号上存在的关系, 给出 3 类数据共享模型的定义. 其实例如图 2 所示. 注意, 为了介绍简单, 示例中每个事务仅包含一个数据库访问操作, 即写或读, 每个事务均只访问数据项 $x$. 在图 2 中, $w_i(K)$ 表示在某一时刻事务 $T_i$ 针对数据项 $x$ 写入/修改的值为 $K$; $w_i r(K)$ 则表示在某一时刻读数据项 $x$ 的值为 $K$, 这一结果是 $w_i$ 对应事务 $T_i$ 所写入/修改的值.

**定义 1 (版本更新操作)**. 版本更新操作 $w$ 是已提交事务 $T$ 对数据库的修改操作. 基于事务的原子性, 操作 $w$ 的执行时间点为事务 $T$ 的提交时间, 记为 $w.t$.

例 1: 如图 2 所示, TP 客户端 A 共发起两次版本更新操作, 即 $w_1$, $w_3$, 分别对应事务 $T_1$, $T_3$ 将数据项 $x$ 修改为 2 和 0, 其执行时间点分别为 $w_1.t = t1$, $w_3.t = t5$.

**定义 2 (版本更新序列)**. 版本更新序列 $S_n$ 是执行时间点小于等于 $w_n.t$ 的全部版本更新操作构成的线性序列,

$S_n=\{w_1,w_2,\ldots,w_n\}(\forall i<j,\ w_i.t<w_j.t)$. 记版本更新序列 $S_n$ 的长度为 $n$, 即 $|S_n|=n$.

**定义 3 (前缀更新序列)**. 前缀更新序列 $S_k$ 是版本更新序列 $S_n$ 的一个前缀线性子序列, 即 $S_k=\{w_1,w_2,\ldots,w_k\}$ $(k\leqslant n\ \&\ \forall i<j,w_i.t<w_j.t)$ [36].

例 2: 如图 2 所示, 在 $t5$ 时刻的版本更新序列为 $S_3=\{w_1,w_2,w_3\}$, 且 $|S_3|=3$. 其前缀更新序列包括 3 个, 分别为 $S_1=\{w_1\}$ ,$S_2=\{w_1,w_2\}$,$S_3=\{w_1,w_2,w_3\}$.

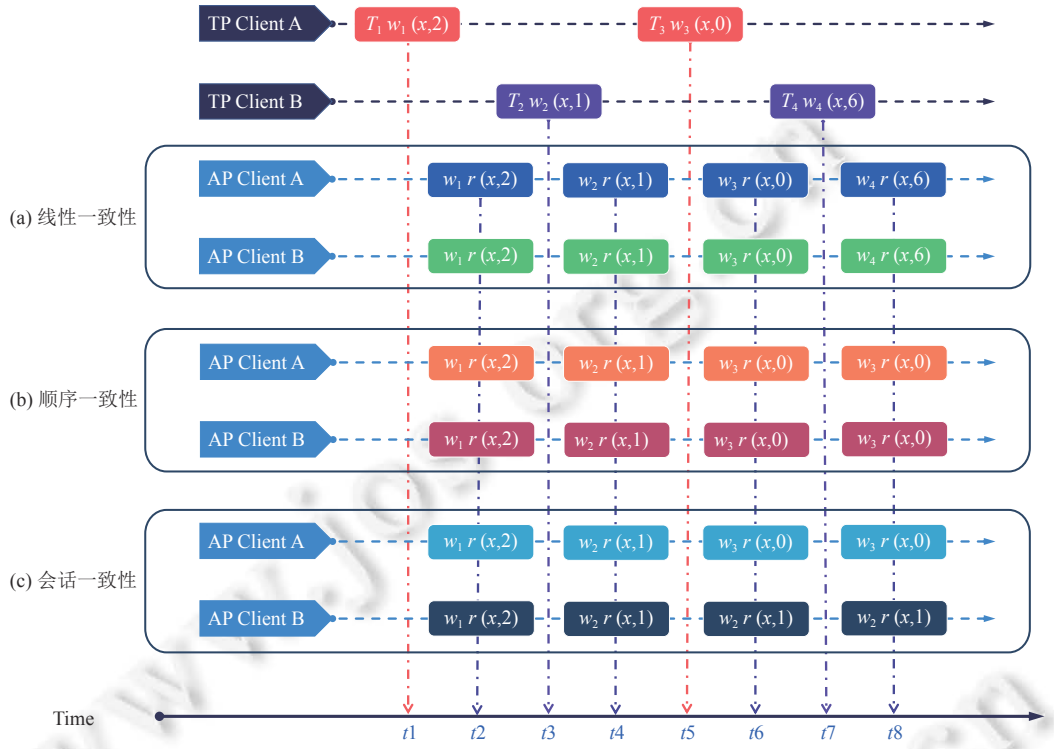



图 2 不同一致性条件下读写数据 $x$

随着 TP 事务的推进, 数据会随着版本更新操作的执行被不断修改 (演进), 产生新数据或更新当前数据. 通过线性执行版本更新序列, 系统可以得到确定的数据版本 (如定义 4 所示).

**定义 4 (数据版本)**. 数据版本 $V_n$ 为版本更新序列 $S_n$ 的线性执行结果. $V_n$ 的版本标识号记为 $|S_n|$, 即 $|V_n|=|S_n|=n$.

例 3: 如图 2 所示, 在 $t4$ 时刻, 执行 $S_2=\{w_1,w_2\}$ 得到数据版本 $V_2$, 即 $x=1$. 在 $t5$ 时刻, 执行 $S_3=\{w_1,w_2,w_3\}$ 得到数据版本 $V_3$, 即 $x=0$.

**定义 5 (版本读取操作)**. 版本读取操作 $r$ 发起对数据库的查询, 记读取数据版本为 $V_r$, 执行时间点为 $r.t$. 文中定义通过会话 $s$ 发起的版本读操作为 $r^s$. 单个会话内多次读取之间需要满足时间维度上的单调性, 在同个会话 $s$ 内, 版本读操作 $r_i^s$ 得到的数据版本 ($V_{r_i^s}$) 不能比 $r_i$ 之前的任意读取操作 $r_j^s$ 读取的版本 ($V_{r_j^s}$) 更旧, 即 $\forall r_i^s.t>r_j^s.t,\ |V_{r_i}|-|V_{r_j}|\geqslant 0$.

**定义 6 (数据共享模型)**. 在给定时刻, 数据共享模型 $C$ 定义在数据项 $X$ 上版本读操作、版本写操作之间的版本约束, 包括:

(1) 写操作产生的版本 ($V_n$) 与当前读操作读取版本 ($V_r$) 之间的版本约束 $C(V_n,V_r)$, 即最新写版本与当前读版本的版本差 $p$:

$$C(V_n,V_r)=|V_n|-|V_r|=n-|V_r|\leqslant p,\ 0\leqslant p\leqslant n.$$

(2) 在同一时刻, 会话 $s_1$ 中读操作 $r_i$ 读版本 ($V_{r_i^{s1}}$) 与会话 $s_2$ 中读操作 $r_j$ 读版本 ($V_{r_j^{s2}}$) 之间的版本约束

$C\left(V_{r_i^{s1}}, V_{r_j^{s2}}\right)$, 即同一时刻不同会话读版本之间的版本差 $q$:

$$C\left(V_{r_i^{s1}}, V_{r_j^{s2}}\right)=\left\|V_{r_i^{s1}}\right|-\left|V_{r_j^{s2}}\right\| \leqslant q,\ r_i^{s1}.t=r_j^{s2}.t,\ 0 \leqslant q \leqslant n,$$

其中, $C(V_n, V_r)$ 表明当前读版本与最新写操作产生的数据版本之间的版本标识号差异, 差异越大, 读新鲜数据的能力越弱; $C\left(V_{r_i^{s1}}, V_{r_j^{s2}}\right)$ 表明同一时刻, 任意两个会话 $s_1, s_2$ 读到版本 $V_{r_i^{s1}}, V_{r_j^{s2}}$ 的版本标识号差异.

HTAP 数据库系统中读取版本的新鲜程度反映为读写版标识号差异, 而差异的程度是由数据库的一致性模型决定的. 主流的一致性模型包括线性一致性、顺序一致性和会话一致性. 其中, 线性一致性以牺牲响应时间和性能的方式换取数据的高新鲜度; 而会话一致性通过适当放松对新鲜度的要求换取更高的性能; 顺序一致性则处于两者之间, 对性能和数据新鲜度进行折中以满足使用需要. 以下我们分别说明 3 种一致性模型与数据共享模型之间的具体对应关系.

#### 2.1.1 线性一致性

线性一致性于 1990 年被首次提出[37], 在线性一致性约束下, HTAP 数据库系统应当保证, 对于任意版本读操作 $r$, $r$ 读取的数据版本 $V_r$ 必然由 $r$ 执行时已完成的最新版本更新序列 $S_n$ 产生. 也就是说, AP 端读取的版本与当前 TP 端版本不存在差距, 即 $p=0$. 因此, AP 客户端在同一时刻所读取到的数据都是一致且最新的, 即 $q=0$.

如图 2(a) 所示, 两个 TP Client 在数据项 $X$ 上执行写操作. 以 $t8$ 时刻为例, 由于 TP 端写入保持全局有序, 此时已完成的更新序列为 $S_4=\{w_1, w_2, w_3, w_4\}$, 此时数据项 $X$ 值的演变顺序为 2→1→0→6; 而不同的 AP Client 在 $t2$, $t4$, $t6$, $t8$ 时刻读到的值分别为 2, 1, 0, 6 与版本更新序列始终保持一致, 即读取数据项 $x$ 的版本总是相应时刻版本更新序列的最大前缀序列, 即分别为 $S_1, S_2, S_3, S_4$.

#### 2.1.2 顺序一致性

在顺序一致性约束下, HTAP 数据库系统应当保证任意读操作 $r$ 读到的数据版本 $V_r$ 应由 $r$ 执行之前的一个前缀更新序列产生, 即 $p=n$; 相同时刻对同一个数据项的多个读操作 $r_i$ 读到的数据版本 $V_i$ 相同, 即 $q=0$, 即各 AP 节点在同一时刻读到的数据一致, 但不保证是最新的.

如图 2(b) 所示, 以 $t8$ 时刻为例, 在版本更新序列为 $S_4=\{w_1, w_2, w_3, w_4\}$ 时, 数据项 $X$ 经历的演变过程是 2→1→0→6, 而两个 AP 客户端 (Client A/B) 在 $t2$, $t4$, $t6$, $t8$ 时刻读到的 $X$ 值都是 2, 1, 0, 0. 其中 $t8$ 时刻与 $t6$ 时刻读取到一样的值, 即基于更新序列的前缀更新序列 $S_3=\{w_1, w_2, w_3\}$ 进行了数据读取, 也就是说 AP 端并不保证读取到 $x$ 最新的数据版本.

#### 2.1.3 会话一致性

在会话一致性约束下, HTAP 数据库系统应当保证会话 $s$ 的任意读操作 $r^s$ 读的数据版本 $V_{r^s}$ 由 $r^s$ 执行之前的一个前缀更新序列产生, 即 $p=n$. 但对于相同时刻执行的对同一个数据项的多个读操作, 所读取的数据对应的前缀序列不一定相同, 即 $q=n$. 在该一致性模型下, AP 端读取的版本与 TP 端版本可能具有差距, 且各 AP 节点读取版本可能不同. 因此, 不同 AP 端在同一时刻所读取到的数据既非最新的, 也非完全一致的.

如图 2(c) 所示, 两个 AP 客户端 (Client A/B) 在 $t2$, $t4$, $t6$, $t8$ 时刻读到的值分别为 2, 1, 0, 0 和 2, 1, 1, 1, 即不同的 AP 请求在不同会话下读取的值不一样, 但整体上与某个前缀更新序列一致, 如 $t8$ 时刻分别对应 $S_3, S_2$, 符合会话一致性约束.

### 2.2 数据共享模型实现

数据共享模型实现涉及数据生产、同步、消费的 3 个步骤, 分别对应如下 3 个子问题 (如图 3 所示).

- 数据版本标识号分配: 即如何以一个确定的版本标识号来标识当前数据版本, 在数据产生时如何为数据分配标识号以及在查询到来时如何为查询分配标识号.
- 数据版本同步: 即如何同步 TP 端的更新序列到 AP 端, 以确保 AP 端在满足不同一致性约束下读到某个版本标识号对应的数据版本.
- 数据版本追踪: 即当查询到来时, 如何在本地或跨多个节点获取某个一致性快照下的数据版本.

实际上, 由于向 AP 端共享最新的版本会引起对 TP 端资源的占用, 所以高性能和高新鲜度存在天然的矛盾, 这

两个目标的权衡最终体现在数据版本的确定、同步、追踪处理的各个环节. 所以数据共享模型需要解决的问题可以归纳为保证 TP 端维持稳定吞吐的前提下, 如何控制 AP 端读到 TP 端实时生产数据的延时以及读取数据的一致性.

一致性模型的强弱决定了 TP 所修改的数据被 AP 可见的速度, 具体反映为数据共享模型中 TP 和 AP 数据版本之间的差异. 下文将从一致性模型的角度进一步分类 HTAP 数据库系统, 探讨其内部数据共享模型的具体实现机制.

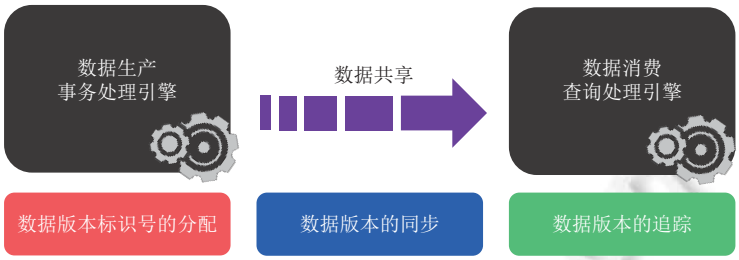



图 3 数据生命周期不同阶段的子问题

### 2.2.1 线性一致性

线性一致性模型要求读请求返回执行时刻对应的最新数据版本, 为此 HTAP 数据库系统需要为 TP 端和 AP 端使用同一套版本标识号管理器, 以保证两者有相同的标识基准. 在线性一致性模型中, 标识号的分配一般是以事务为单位, 即基于事务提交时间生成数据版本标识号.

由于在线性一致性模型下, 读操作始终读取最新的数据, 其对数据版本的同步速度要求严格, 容易导致资源竞争. 对于单机数据库系统, 影响主要体现在多副本合并存储所带来的资源竞争; 对于分布式数据库系统, 线性一致性约束要求各节点完成数据同步后才能读取, 因此若各节点间数据独立则可以一定程度上避免这一问题, 但这一数据划分方式无法支持负载均衡并造成主节点在事务管理上的瓶颈问题. 所以, 在单机情况下, 快速同步所带来的资源竞争会导致 TP 端的吞吐下降, 而多机不同副本间的同步等待时间过长会增大读请求的延迟, 因而对分析性能产生负面影响.

在线性一致性中, 对于单机数据库系统, AP 查询主要基于并发控制协议追踪最新的数据版本. 而对于分布式数据库系统来说, AP 端需要确认查询发起前的最新版本已完成同步, 才能追踪到最新的数据版本. 确认机制一般有两种, 一种是读节点主动问询主节点的最新版本标识号; 另一种是主节点定期推送最新版本标识号给只读节点, 使只读节点获知这段时间内数据的修改状态.

综上所述, 线性一致性模型适合对一致性要求高, 需要强一致性读的场景; 与此同时, 高一致性通常是以数据库整体性能下降为代价实现的.

### 2.2.2 顺序一致性

顺序一致性模型要求同一时刻的版本读取返回相同前缀序列对应的数据版本, 且不晚于上一时刻返回的版本. TP 和 AP 端可以使用不同的标识号管理器进行相对独立的数据读取, 但 AP 端需要共用同一套标识号管理器以确保 AP 端读取版本的一致性. 需要注意的是, AP 端的标识号管理器通过回放的版本确定标识号, 而不是重新申请分发.

由于顺序一致性不要求读取 TP 端最新的数据版本, 所以数据库系统可以适当降低同步频率, 如可以基于 Epoch 进行阶段性同步, 只同步同一阶段的最终版本, 减少同步数据量, 减轻繁重的同步任务对资源的抢占, 对保持 TP 负载高吞吐有积极的影响.

顺序一致性适当放松数据新鲜度的要求, 一般只读取当前同步到的版本, 避免了 AP 负载的同步等待. 在追踪已同步到的版本快照时, 根据同步的分区数量有不同的追踪方法. 同步队列为单个队列时, 数据库可以利用已同步的标识号作为全局标识号进行读取. 同步队列为多分区并行同步时, 需要以当前所有分区已同步的最小标识号作为读请求的版本.

综上所述, 由于数据库允许在顺序一致性下适当降低同步速率、放松数据新鲜度的要求, 所以更适合处理写密集的负载, 很好地避免了批量读取数据造成的长时间同步延迟. 但在顺序一致性模型下, 减轻同步压力是以降低

数据新鲜度为代价的. 如果数据库需要更好地提高 AP 端读取数据的新鲜度, 则需要提高同步的频率, 但可能会增大同步任务对资源的抢占, 从而干扰写负载的执行, 即降低 TP 端性能.

2.2.3 会话一致性

会话一致性模型要求读请求返回某个前缀序列对应的数据版本, 且不晚于会话内上一时刻返回的版本. 为此, TP 和 AP 端可以使用不同的标识号管理器进行独立的数据版本管理, 且各 AP 端之间的标识号管理器也相互独立. 需要注意的是, 每个 AP 端的标识号管理器通过回放的版本确定各自的标识号, 而非重新分发标识号.

会话一致性允许不同会话读取不同的数据版本, 所以在 3 种模型中数据同步要求最宽松. 数据库在会话的缓存中维护增量版本以用于 AP 端读取数据. 单机数据库系统一般基于 COW 机制创建会话快照以维护增量数据版本, 在内存中进行数据拷贝以实现同步; 分布式数据库系统则一般会在会话的缓存中回放版本以同步数据, 而各会话缓存中的数据版本和已在存储中持久化的版本可能不一致, 所以当 AP 查询读取的数据未命中缓存时, 分布式数据库系统需要从存储中获取与缓存中版本对齐的版本数据, 从而保证缓存和存储快照的一致性.

单机 HTAP 数据库系统一般通过读取当前会话缓存的快照进行数据版本的追踪; 而在分布式场景下, 当从存储中获取数据版本时, 可能存在不同的数据版本追踪方式. 如果底层存储支持版本追踪, 为了避免等待底层存储同步到与缓存一致的版本, 就要求缓存的版本始终不早于存储的版本, 那么 AP 端可以从存储中读取与缓存一致的较旧版本, 读取延迟较低, 但是会降低读取的数据新鲜度; 如果存储不支持版本追踪, 那么要求缓存的版本始终不晚于存储的版本, 那么 AP 端则需要在读取后将存储中的版本回放到与缓存一致的数据版本上, 提升了数据新鲜度, 但增大了延迟.

综上所述, 会话一致性模型的实现一般是基于以缓存版本为基础的快照一致性, 在单机中由于创建多个会话的数据快照, 内存数据拷贝效率较高, 同步代价低, 但往往会带来较大的内存开销; 而在分布式场景下由于缓存中需要同步的日志量少, 同步效率高. 但是, 在维护缓存与存储之间的一致性要求下, 数据库需要设计复杂的机制来保证快照完整性, 同时, 当数据库系统存在多个 AP 会话时, 应用层需要解决多个 AP 节点宕机重连导致的数据版本不一致问题.

从上述 3 种一致性模型的总结来看, 数据共享模型与分布式数据库的一致性模型间存在一定的关联, 但不完全相同. 对于同一数据不存在多个物理副本的情况, 数据共享模型约束了读取时该数据版本的可见性, 而一致性模型不考虑这种情况. 对于同一数据存在多个物理副本的情况, 数据共享模型的约束与一致性模型的要求相对应, 但关注的角度不同. 数据共享模型会约束数据库读取之前写入的版本的新鲜程度, 不关注各物理副本内容的一致性; 而一致性模型要求各物理 (或多数) 副本内容相同[38], 不关心版本的差异. 两者对数据的约束最终都依赖于多副本之间的同步方式, 以达成数据在某个维度上的一致性状态.

因此, 数据共享模型的具体实现与一致性强度的具体实现紧密相关, 不同的一致性级别对应数据共享模型中不同强度的约束, 在 HTAP 数据库系统中具有不同的实现机制, 表现出不同的特征 (总结见表 1, 其中 $n$ 为当前版本更新序列的长度). 在线性一致性模型下, TP 端和 AP 端共用一套标识号管理器以实现标识号的分配, 并通过快速地同步保证两者的数据版本同步推进, 使得读请求可以读取到最新的版本; 在顺序一致性模型下, AP 端可以具有独立的一套标识号管理器, 基于批次的方式进行数据同步, 使得在读请求获取的版本与最新版本之间有统一的时间偏移; 在会话一致性模型下, 每个 AP 端的会话可以有不同的标识号管理器, 但会话内基于缓存一致性有序推进数据版本, 因此 AP 端所见版本与最新版本之间存有不统一的版本偏移.

表 1 HTAP 数据库系统数据共享模型特征

| 一致性级别 | $p$ | $q$ | 版本标识号 | 版本同步 | 版本可见性 |
|---|---|---|---|---|---|
| 线性一致性 | 0 | 0 | TP/AP使用统一标识号管理器 | 基于事务号 | 当且仅当最新版本可见 |
| 顺序一致性 | $n$ | 0 | TP/AP拥有独立标识号管理器 | 基于日志号或Epoch号 | 各AP所见版本与最新版本有一致的版本偏差 |
| 会话一致性 | $n$ | $n$ | TP有统一的标识号管理器; 每个AP有独立标识号管理器 | 基于缓存一致性 | 各AP所见版本与最新版本可有不同的版本偏差 |

### 2.3 数据共享模型案例分析

除了各个一致性模型下所满足的一致性特征之外, 还有一些在数据生命周期影响数据库性能的实现手段值得关注. TP 端数据自诞生伊始直至被 AP 端读取消费, 其存在如上所述的完整共享过程和生命周期; 但在具体实现上, 也存在部分 HTAP 数据库系统或是学术研究尝试从并发控制机制本身, 如对 MVCC 进行优化实现对混合负载的处理, 则此时 TP/AP 间不存在严格意义上的数据同步.

数据版本标识号分发阶段, HTAP 数据库系统在不同的一致性模型下具有不同的版本标识号实现和管理方法. 对于采用 MVCC 机制的数据库系统[4,8], 标识号通过版本戳实现; 对于通过日志落盘确定版本的数据库系统, 标识号通过 LSN 实现[13,15]; 对于采用 COW 等机制的数据库系统[5], 标识号不表现为具体的版本戳/日志号数值, 而是通过创建一个旧版本数据快照进行版本的粗粒度标识, 以体现数据新旧程度. 更进一步, 在版本标识号分发时, 线性一致性下会使用单机 TSO (timestamp oracle)[39]、全局 TSO 和原子钟等分配 MVCC 的版本戳, 在顺序一致性下会使用已同步的日志号或已同步的最小 Epoch 日志号来返回读请求的版本[16], 在会话一致性下则会使用缓存中已同步的日志号[14,15].

在数据版本同步阶段, 一般而言, 现代 HTAP 数据库系统往往都会通过内存/网络数据拷贝[3,4]或是日志同步[12]的方式进行 TP/AP 端数据的共享, 且一般需要补充一些额外的功能以支持数据的同步与回放, 如 TP 端针对发送日志的控制手段以及 AP 端从行存到列存格式的转换等. 同时同步的内容和范围也会对 TP 和 AP 的性能产生影响. 大部分数据库会采用写操作结束后异步更新的方式, 但也有部分数据库支持同步备份以满足高可用等需要, 同步更新延缓了事务的提交, 进而影响吞吐. 目前 HTAP 系统普遍以日志作为同步内容, 并通过回放日志等方式将数据更新到对应的副本中, 直接针对数据进行同步的方式并不多见. 由于需要同步大量数据版本, 基于回放日志的方式更新版本会带来较大的磁盘 IO 开销, 甚至包括格式转换时对存储和计算资源的消耗. 与之相对的同步方式是直接拷贝数据, 这避免了格式转换的计算资源消耗, 但会带来较大的 IO 传输开销. 另一方面, 在同步数据范围上, 大部分数据库同步全量的修改, 少数数据库只需要在每个周期结束时同步当前周期所产生的最终版本. 但周期设置得过短时, 频繁地同步会一直对性能造成影响, 而周期设置得过长会导致延迟过大. 也正因为同步方式的不同, 对于需要读取最新版本的接收端而言, 等待最新版本同步的延迟也有所不同.

在数据版本追踪阶段, AP 端进行分析查询的性能很大程度取决于数据版本的组织方式和全局一致性快照的确认机制. 在多副本的情况下, 不同副本间可能采用不同的版本组织方式: 多版本副本大多采用链式数据结构进行存储, 可能会根据链上版本产生的时间组织成从新到老 (newest-to-oldest, N2O) 或从老到新 (oldest-to-newest, O2N) 的形式, 通常采用遍历版本链的方式检索合适的版本; 优化分析查询的副本也可能只保存单副本, 从而减少检索版本的开销. 对于不同的数据库, 追踪全局一致快照的方式有所不同. 当选取当前时间戳对应版本作为读取版本时, 可能需要等待 TP 端完成相应数据的同步; 当选取某个较旧版本时, 又会损失一定的新鲜度.

综上所述, 在版本生命周期的 3 个环节中, 存在各种因素需要权衡, 因此不同数据库会选择不同的实现方式以适用于不同的应用场景. 在表 2 中, 根据前文一致性模型的定义对主流 HTAP 数据库进行分类, 并依据分类和具体的实现手段对各个系统进行了总结.

#### 2.3.1 支持线性一致性的数据库系统

HyPer[6]是基于内存的单机数据库, 其最新的版本采用了 MVCC 机制以支持 HTAP 业务, 其中 TP 就地更新数据并将旧版本镜像增量存储在每个事务独立的缓冲区中. 所有进入系统的新事务都与 3 个变量相关联: 即 startTime、事务号和 commitTime. startTime 和事务号在事务产生时自动分配, commitTime 在事务提交时直接从本地单机获取, 查询只能看到时间戳先于 startTime 的数据版本, TP 在就地更新的过程中, 会将新版本的时间戳暂时更新为事务号, 由于事务号一定远远大于时间戳 (时间戳从 0 开始, 事务号从 $2^{63}$ 开始), 这样就保证了还未提交的数据只能由自身看到. 就地更新和旧版本数据镜像增量存储不仅保证了 AP 高效的扫描, 还使 HyPer 具备轻量且细粒度的可串行化验证机制, 从而简化了 TP 事务的隔离逻辑.

表 2 HTAP 数据库系统特征总结

| 一致性 | 数据库 | 数据标识号分配 | | | | 数据版本同步 | | | | 数据版本追踪 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 方式 | 精度 | 分配方式 | | 补充功能 | | 同步范围 | 同步格式 | 组织方式 | 确认机制 |
| | | | | TP | AP | TP | AP | | | | |
| 线性 | HyPer (new) | 事务号 | 时间点 | 单机TSO | | — | — | — | — | 链式 | 读取读时已提交的最新版本 |
| | Greenplum | 事务号 | 时间点 | 全局TSO | | — | — | — | — | 链式 | |
| | SAP HANA | 事务号 | 时间点 | 单机TSO | | — | — | 全量 | 数据 | main单版本, delta链式 | 首先读取delta中读时已提交的最新版本, 数据缺失时从main中读取 |
| | Hyrise | 事务号 | 时间点 | 单机TSO | | — | — | 全量 | 数据 | | |
| | FoundationDB | 事务号 | 时间点 | 全局TSO | | — | — | — | — | | |
| | Oracle | 事务号 | 时间点 | 单机TSO | | — | — | 全量 | 日志 | buffer链式, 内存单版本 | 首先读取内存中读时已提交的最新版本, 数据缺失时通过日志读取 |
| | SQL Server | 事务号 | 时间点 | 单机TSO | | — | — | 全量 | 数据 | 行存链式, 列存单版本 | 首先读取列存中读时已提交的最新版本, 数据缺失时从行存中读取 |
| | MemSQL | 事务号 | 时间点 | 全局TSO | | — | — | 全量 | 数据 | 链式 | 读取读时已提交的最新版本, 数据缺失时等待数据完成同步 |
| | TiDB | 事务号 | 时间点 | 全局TSO | | 发送日志&确认最新版本号 | 日志回放&格式转换 | 全量 | 日志 | 链式 | |
| | Spanner | 事务号 | 时间点 | 原子钟 | | 发送日志 | 日志回放 | 全量 | 日志 | 链式 | |
| 顺序 | F1 Lightning | 日志号 | Epoch | 单机TSO | 已同步日志号 | 发送日志 | 日志回放 | 全量 | 日志 | 链式 | 读取当前已同步版本 |
| | IDAA | 日志号 | Epoch | 单机TSO | 已同步日志号 | 发送日志 | 日志回放 | 全量 | 日志/数据 | 链式 | |
| | BatchDB | 日志号 | Epoch | 单机TSO | 已同步日志号 | 发送日志 | 日志回放 | 全量 | 日志 | TP链式, AP单版本 | |
| | Vegito | 事务号 | Epoch | 全局TSO | 已同步全局最小Epoch | 发送日志&分割Epoch | 日志回放&格式转换 | 部分 | 日志 | 块式 | |
| | MySQL Heatwave | 事务号 | 时间点 | 单机TSO | 已同步事务号 | 发送数据 | 格式转换 | 全量 | 数据 | TP链式, AP单版本 | |
| | ByteHTAP | 日志号 | 时间点 | 单机TSO | 已同步日志号 | 发送日志 | 日志回放 | 全量 | 日志 | main单版本, delta链式 | 读取读时已同步的版本, 该版本通过合并main和delta得到 |
| 会话 | HyPer (old) | 事务号 | 时间点 | 单机TSO | | — | — | — | — | — | 当创建会话时, 创建新的数据快照, 并读取该快照下的数据 |
| | Aurora | 日志号 | 时间点 | 单机TSO | 缓存已同步日志号 | 发送日志 | 日志回放 | 缓存部分, 存储全量 | 日志 | 链式 | 读取缓存中的版本, 数据缺失时从存储中读取版本一致的快照 |
| | PolarDB | 日志号 | 时间点 | 单机TSO | 缓存已同步日志号 | 发送日志&刷脏控制 | 日志回放&缓存版本对齐 | 缓存部分, 存储全量 | 日志 | 缓存链式, 存储单版本 | |

Greenplum[9]作为分布式的行列混存数据库, 只有在分布式事务中才需要设置全局的事务号, 对于仅在本地执行的事务, 则由节点自行维护版本, 不需要全局统一. 在开始分布式事务时, Greenplum 的主节点首先对所有节点

的当前本地事务号与分布式事务号进行映射, 以获取全局快照点. 映射时主节点需要与所有节点进行通信, 开销较大. Greenplum 架构中使用同一个副本来同时处理 TP 和 AP 任务, 并采取行列混存的模式以保证读写性能, 因此当 TP 负载对数据版本进行修改后, AP 端的查询可以直接读取对应的数据, 不需要进行多副本之间的同步. 另外, Greenplum 使用了基于虚拟化技术的单节点资源隔离以保证性能, 但是其不需要等待网络同步, 不存在等待导致的额外延迟.

SAP HANA[4]是基于行列混合存储的单机内存数据库. 在 SAP HANA 中数据存放分为两个部分: 主库 (main store) 和增量库 (delta store). 其中增量库采用 MVCC 机制对写操作进行了优化, TP 和 AP 请求都是从单机直接获取时间戳, TP 进行数据更新时只会对增量库进行更新, 增量数据会定期合并到主库中, 而主库中的数据有很高的压缩比率并对读操作进行了优化. 当 AP 进行读时, 需要同时读取主库和增量库中的数据并进行合并. 因此在数据版本方面, 主库维护一个只读副本, 增量库基于 MVCC 版本链维护一个读写副本. 由于 AP 端支持直接从增量分块读取数据, 因此 SAP HANA 无需等待副本同步, 避免了多副本同步的开销和同步延迟.

Hyrise[3]是单机的内存数据库, 它的存储结构采用行列混存分区. 各分区间的数据互不重合, 且每个分区内划分为主库和增量库以存储数据, 不需要进行分区间的数据同步. Hyrise 处理查询与上述两种数据库类似, 基于其单机特性可以简单地分配事务号, 不需要网络请求. Hyrise 的写入采用类似 SAP HANA 的机制, 在 TP 端进行版本更新时, 先对增量库通过追加版本实现更新、修改和删除, 再定期将增量库中的内容进行压缩编码并合并到以列存形式存储的主库中, 从而实现不同副本之间的同步. 在进行 AP 查询时先根据索引定位需要查询的分区, 再基于分区的存储形式从中提取数据. Hyrise 与 SAP HANA 采用了类似读取机制, 不需要等待完成多副本同步便可以读取新版本的数据, 因而不存在同步导致的延迟.

SQL Server[8]是基于行列混存的单机数据库, 它采用基于内存的行存引擎 Hekaton 和基于磁盘的传统 SQL 引擎来支持行存, 采用增量库 (tail index) 和主库 (CSI) 的组合来实现列存索引以加速 AP 查询. SQL Server 采用 MVCC 机制进行版本的管理, TP 端和 AP 端都可以直接从单机上获取时间戳. 当发生版本更新时, 数据库同时更新 Hekaton 和增量库中的数据, 并将主库中的对应数据标记为删除. 随后, 数据库通过后台进程异步地将增量库中的数据压缩合并到主库中, 从而实现版本同步, 此时数据库会更新 Hekaton 中的相关记录, 使 AP 端查询可以直接访问更新后的主库, 这一更新过程会影响 TP 负载的执行. 与 SAP HANA 类似, SQL Server 的 AP 读取请求会同时从增量库和主库中读取数据, 因此不存在等待同步的开销. 为了减少频繁同步带来的资源开销, SQL Server 只同步冷数据, 将频繁修改的热数据保留在增量库中.

Oracle[6]也通过结合内存型列存 (in-memory compression units, IMCU) 和行存缓存 (row-based buffer cache), 扩展了原有 OLTP 业务以实现对混合负载的支持. 在 TP 负载进行写入时, 缓存通过维护数据多版本, 基于单调递增的时间戳 (system change number, SCN) 标识事务. 同时会在列存创建一份一致快照下的数据, 并通过 SCN 进行创建时刻的标识. 当新的 AP 查询到来时, 会通过一个元数据管理模块 (snapshot metadata unit, SMU) 检测列存数据的有效性, 如有早于当前 AP 查询 SCN 号的更新, 则通过扫描增量的事务日志以保证读取最新数据.

FoundationDB[11]是单版本的分布式数据库, 采用单机节点集中授时, 它对 HTAP 负载的支持与之前系统不同, 不区分 TP 端和 AP 端, 通过把系统的事务管理和存储解耦, 使读写操作访问不同的路径实现隔离. 对于 TP 负载的写操作, 提交前会申请 TSO 以确定事务号, 但仅会进行日志预写, 还需要后续存储异步拉取并回放日志, 对应的版本才对读操作可见. 每次写操作需要等待所有分片所在的日志服务器都收到最新版本写入通知才算提交完成, 若遇到某一部分单机数据的热点写入, 整个集群的日志写入都将被拖慢. 由于 FoundationDB 采用异步的方式进行同步, 在 TP 负载的写入提交后对应的版本可能还未完全同步到所有分片的存储中, 因此, 每次执行 AP 读请求时需要申请 TSO 并判断该 TSO 对应的最新已提交版本是否完成同步, 若未同步完则选择等待或者从已经同步完的副本上发起二次读, 以保证读版本正确性, 这也会增加读延迟.

MemSQL (于 2020 年 10 月更名为 SingleStoreDB)[10,40,41]作为分布式数据库, 拥有中心化的主节点用于接收查询请求, 并对事务进行管理. 其他节点存储不相交的数据并执行主节点所转发的请求. MemSQL 采用较为复杂的数据存储格式, 具有三副本, 使用行存的二级索引和临时内存块优化 TP 负载性能, 并使用列存形式的 LSM-Tree

(log-structured merge-tree) 以优化 AP 查询性能, 再通过一个行存副本保证高可用. 当 TP 端执行更新数据版本的请求时, MemSQL 先将数据转换为内存中的行存副本, 等修改完成再快速更新到存储. 因此, 数据库在执行写后, 同步地将数据复制到高可用副本中, 再异步地进行读写副本间的同步, 而 AP 端在进行查询时需要等待数据库完成异步的副本同步, 这一过程会产生一定的延迟. 但是由于行存和列存副本存储在同一节点上, 同步代价较小, 额外延迟不高.

TiDB[12]是一款分布式 HTAP 数据库系统, 将数据依据主键值划分为多个连续的块 (Region), 并以 Region 为单位分散副本到多个节点中, 形成 Raft 组. 各个组内部基于 Raft 一致性协议[42]进行同步, 支持多节点读写. 在每个组内, TiKV 节点采用行存处理 TP 负载, 作为 Raft 组中的 Master/Follower 节点进行即时同步, 同时 TiDB 维护了一个额外的列存 TiFlash 节点处理 AP 负载, 它作为 Raft 组中的 Learner 节点 (需保证内部状态与其余节点相同, 但不参与 Raft 组的选举和投票[43]) 接收数据的同步, 但并不要求 TiKV 产生的数据即时更新到 TiFlash 节点. TiDB 为了维护准确的版本顺序, 虽然支持多个节点同时进行读写操作, 但在各节点的数据不完全独立的情况下, 全局范围内必须只有唯一的节点用于分配 TSO. 在执行 TP 负载产生新数据版本时, TiKV 节点会申请 TSO 确定新版本的事务号, 然后采用基于 Raft 协议的同步机制异步地更新日志到其他节点, 接收日志的节点回放日志操作以更新数据版本, 实现数据同步. 异步更新降低了同步更新导致的 TP 吞吐下降. 在 TiFlash 节点接收到 AP 读请求并开始执行时, TiFlash 会为该读请求申请 TSO, 并等待 TiKV 将这一 TSO 对应的最新已提交版本同步到自身节点再读取这一版本的数据, 这会导致一定的延迟.

Spanner[44]与 TiDB 架构类似, 拥有多个节点组, 各个副本基于 Paxos 一致性协议[45]进行同步. 但 Spanner 的数据中心在全球范围内分布, 若采用单节点管理 TSO 带来的通信代价难以忍受, 因此在不同的数据中心内均采用不同的多个持有物理时钟的节点进行时间戳分配, 并借助这些节点间的通信将误差控制在一定范围内, 即 Spanner 在维护全局时间戳的同时采用多个物理时钟进行时间戳分配. Spanner 在 TP 事务提交时申请时间戳, 以这一时间戳作为基准, 基于 Paxos 协议进行版本日志同步. AP 端在执行读操作时也会申请 TSO, 并等待该 TSO 所对应的最新版本同步到节点中再读取. 同样由于同步本身的耗时, 读请求在读取最新数据时需要等待日志同步完成, 导致一定的延迟.

综上所述, 支持线性一致性的数据库系统要求读请求返回最新的版本. 单机数据库可以通过共用一份数据以满足要求, 但这种数据存储和使用方式无法同时优化 TP/AP 负载的执行性能, 因此, 采用存储格式不同的增量/主 (delta/main) 分块组织数据是主流的解决方案. 而分布式数据库需要快速地全量日志同步以保证读取新鲜的数据版本, 但这会抢占大量 TP 端的资源, 影响 TP 负载的性能. 为此, 大部分数据库选择将相同数据的 TP/AP 副本保存在同一物理节点上以避免大量的日志同步. 即便对于副本保存于多节点的情况, 数据库也会采用异步更新的方式以降低即时同步对 TP 的影响.

#### 2.3.2 支持顺序一致性的数据库系统

F1 Lightning[13]本质是一种类 ETL 的 HTAP 系统, 以在线即时的 CDC (change data capture) 的方式, 即实时提取并记录源数据库的操作以供下游服务消费, 取缔了离线的 ETL 操作, 支持混合负载处理. 该系统在逻辑上保持 TP 端与 AP 端的强分离. TP 端发起的事务由全局 TSO 分配序列号, 当有新的数据版本产生时, 它通过调用 CDC 接口, 异步进行事务日志的重构和回放, 以不阻塞 TP 写入的方式将新数据组织到 AP 端以列存形式存储备份. 同时, 由于分布式场景下事务日志回放时间的不确定性, F1 Lightning 支持 AP 端快照一致的读取, 即与 TP 端某个时间点下读取的数据保持一致, 存在一定的时延, 并不保证强实时性.

IDAA[46]是一款 IBM 研发的 HTAP 数据库系统, 部署了事务处理能力较强的 Db2z 作为 TP 端, 部署列式存储的 DB2 Warehouse 作为 AP 端, 并在 DB2 Warehouse 之上扩展了一个服务模块, 该模块包括数据变更的监视与应用执行组件 InSync 和更新控制组件. 数据的标识号由事务的操作顺序确定. 当发生版本更新时, TP 端以适合列存使用的日志形式将版本同步到 AP 端, AP 端通过更新控制组件决定数据同步的粒度大小, 即选择 InSync 使用粗粒度的表级/分区级的批量更新或是细粒度的增量修改完成同步. 对于读请求, 由 InSync 判定读请求的版本是否已同步到 AP 端, 若未完成同步则等待至最大等待时间, 再选择直接读取 TP 端中的对应版本或是直接终止查询. AP 端的 InSync 比 IBM 的 CDC 更轻量、更通用地维护阶段性的版本加载位点. 当完成连续两个版本加载位点之间

的所有查询后, 两者的数据版本可被清除. IDAA 在系统中实现了查询可选择从 TP 或 AP 端读取的设计方案, 支持在未同步完全为查询的延迟带来较大优化.

BatchDB[20]基于数据拷贝实现 HTAP 服务, 即为 TP/AP 端维护独立数据副本. 值得关注的是, 其引入了批处理这一理念, 即 TP 请求和 AP 查询都会置于一个请求队列批量处理, 实现数据同步和 TP 写入的轻量隔离. TP 端通过全局 TSO 获取时间戳后, 基于 MVCC 将 TP 写入组织成多个数据版本, 接着将更新置入一个更新队列中, 定期更新 AP 端唯一的快照版本数据. 而当批量的 AP 查询到来时, 这一批 AP 查询都会访问到该时间点下唯一的快照数据版本. 这样的批处理设计优势在于分摊了数据同步代价, 避免了遍历版本链的扫描代价, 也减小了存储和垃圾回收的代价. 当然, 这类批查询处理很容易带来查询短板效应, 即单个慢查询会遏制整体查询的处理性能, 另外, AP 端查询可能需要等待快照更新造成不可避免的延时.

Vegito[16]是一款 HTAP 数据库系统原型, 其创新性地提出了一种基于 Epoch 的数据同步机制. Vegito 一方面为了实现高可用而保持三副本的设计, 将传统三副本划分为主副本, TP 副本, AP 副本, 即本质上仍是利用多个备份为读写场景分别服务. 另一方面, Vegito 在数据同步中引入了 Epoch 式的时间戳划分方式: TP 主副本端由全局 TSO 主动分配当前事务所处的 Epoch 号, 确定当前事务处于哪一个分段/批次. 接着系统将 TP 端事务操作写入到日志中并行地同步数据, 接着再异步地应用到 AP 端进行数据和索引的更新, 此时 AP 端可以读取到某个 Epoch 下已经同步完成的稳定版本. 这种将多个事务主动划分为固定分段的方式, 很大程度上缓解了单个事务到来时, 个体申请全局时间戳竞争的代价.

ByteHTAP[47]是一款读写分离的 HTAP 数据库系统, 与 TiDB 类似, 采用两套不同的存储引擎分别处理 TP/AP 负载, 并使用 Metadata Service 存储版本信息. 其中, TP 端数据采用行存存储, AP 端通过类似 SAP HANA 的方式, 基于行存的多版本 Delta 分块和行列混存的无版本 Base 分块存储数据版本, 并随着版本的推进定期将旧版本的数据转化为列存, 存储在 Base 分块中, 以提高 AP 端的查询速度. 当 TP 端产生新数据版本后, 数据库会将版本对应的 Redo Log 同步到其他存储节点上, 保证在超过多数节点已持久化对应日志的情况下提交事务, 实现高可用. 同时, 读写节点会根据持久化顺序为每个 Redo Log 分配一个日志序列号 (LSN), 并在提交时交与数据库的 Metadata Service 保存. 数据库周期性从 Metadata Service 获取当前已提交的最新 LSN, 并令 AP 端读取不超过该 LSN 的最新数据版本. 当不能保证小于等于这一 LSN 的版本已被完全同步时, 数据库需要等待对应的数据版本完成同步, 因而导致一定的延迟.

MySQL Heatwave[48]是一个具有单写节点的分布式数据库, 它通过在单机 MySQL 上增加分布式、列式存储、可扩展的查询处理引擎 Heatwave 节点来提高 AP 端查询的效率. 每个 Heatwave 节点采用列存形式在内存中维护一部分数据. MySQL Heatwave 数据库与 MySQL 数据库一样, 使用单机 MySQL 上的行存 innoDB 引擎负责处理所有 TP 端请求和版本戳分发. 当发生版本更新时, 新版本首先会被存储到 innoDB 中, innoDB 实时将数据发送到 Heatwave 节点, Heatwave 节点使用数据驱动的分析策略决定数据列存合并转换的周期. 当一个查询到来时, MySQL 的查询优化器会自动选择将这个查询本地执行还是将其下推到 Heatwave 中, 由于 Heatwave 是一个云服务插件, 因此 AP 对 TP 干扰较小.

综上所述, 支持顺序一致性的数据库系统普遍采用分离式的架构, 从而适合为 TP/AP 引擎管理独立的数据副本以避免相互干扰且便于优化. 同时, 顺序一致性模型中基于 Epoch 的同步设计较为灵活, 即可以根据业务的需求均衡性能和新鲜度指标, 选取合适的 Epoch 粒度. 此外, 该类数据库具有阶段性、批量同步的特征, AP 端普遍不再存版本链, 主流的方案会选择维护 AP 端单版本或是每个 Epoch 的快照以支持读操作. 值得关注的是, 为了提高数据库的新鲜度, 一些数据库 (如 ByteHTAP[47]) 也支持读取未更新到 AP 端数据上的日志.

#### 2.3.3 支持会话一致性的数据库系统

HyPer[5]是基于内存的单机数据库, 在它的早期版本中, TP 和 AP 使用同一个数据副本, TP 端单线程串行执行. 当 AP 端读请求到来时, HyPer 通过 fork 系统调用来创建新进程并将其与 TP 进程共享内存映射, 以构建满足数据一致性的只读快照. 在创建快照时, 如果存在事务正在执行, 则需要通过 undo log 让数据库得到事务开始前的一致性状态. 由于 TP 和 AP 使用同一个副本, 当 TP 端修改数据后, 需利用 COW 机制对修改的内存页创建旧数据

拷贝, 未完成的 AP 会话仍可基于旧数据进行查询, 而新的读会话会在最新 TP 数据上创建只读快照, 实现了多个会话的独立快照读取.

Aurora[15,49]是一款云原生 HTAP 数据库系统, 主打日志即数据库. Aurora 采用存算分离架构, 计算节点包含一个写节点和多个读节点, 所有计算节点共享同一个多版本存储引擎, 使用 Quorum 协议[50]维持数据一致性. Aurora 基于日志号进行数据组织. 首先, 写节点根据事务操作序为其生成日志号, 然后将日志写入存储引擎并传递给读节点. 当写节点产生的日志被超过半数的存储节点记录后即可返回. 之后, 存储节点将日志异步回放到自己的存储副本中, 只读节点将日志异步回放给缓存中存在的数据. 读请求发起时, 只读节点根据缓存中最新的版本快照点确定查询的读取快照, 如果查询的数据只在缓存中, 则直接返回, 否则从存储引擎中读取所需的版本进行补缺. 读取的数据将缓存到只读节点, 并在后续的日志到达时, 由只读节点回放并推进其版本. 为了使缓存需要的版本数据始终能在存储中找到, 缓存中的版本快照点应始终小于存储中已回放的版本快照点.

PolarDB[14,51]也是一款采用存算分离架构的云原生 HTAP 数据库. 其提出 PolarFS[14]来实现写节点和只读节点的数据共享. PolarFS 基于 Raft 协议可以在应用层和存储块层之间构建高可用的分布式文件系统. 与 Aurora 类似, PolarDB 同样基于日志号进行数据共享, 写节点需要将日志同时传递给 PolarFS 和只读节点, 写节点将日志写入 PolarFS 后即可返回. 但与 Aurora 不同的是, PolarFS 不具备日志回放能力, 需要写节点将脏数据块本地落盘以实现数据共享, 因此存储中的数据为单版本. 所以, 只读节点无法依据缓存中的版本快照点从存储中获取相同版本快照点的数据. 为了保证缓存与存储的版本一致性, 写节点会对 PolarFS 中的数据版本标记“日志检查点”, 只读节点从存储中获取数据后, 只需要从该检查点回放日志到缓存中的版本快照点即可对齐版本. 需要注意“日志检查点”不能超过任何只读节点的版本快照点, 否则只读节点有可能查询到未来版本的数据, 使版本无法保证顺序推进. 此外, 当出现写密集负载时, 热点数据块标识号激增, 远超只读节点的版本, 导致该数据块无法落盘. 这种情况下, 当写节点推进“日志检查点”时, 由于存储引擎中缺失该热点数据块的历史版本, 无法形成完整的全局数据快照完成推进检查点. 这样的话, 只读节点需要回放的日志序列越来越长, 降低了查询性能. 为此, PolarDB 提出了将热点块的历史版本周期性分离为 Shadow Page, 从而可以将其落盘以推进“日志检查点”.

综上, 支持会话一致性的 HTAP 数据库系统基本都是采用基于缓存一致性原则进行数据的读取, 所以重点在于如何同步缓存中的数据以达成一致的版本快照. 作为单机数据库系统, 旧版本 HyPer 通过 COW 机制从内存中创建快照以供缓存使用, 并通过读取当前会话缓存追踪数据版本, 以较小的同步代价发挥了内存型数据库系统的优势. 而对于分布式数据库系统而言, 存储引擎是否支持版本追踪, 对于此类 HTAP 数据库系统性能和新鲜度的影响较大. 如果缓存不命中的话, Aurora 和 PolarDB 都需要从存储中进行数据读取. 但由于 Aurora 底层存储是有版本的且存储的版本相较于缓存更新, 所以为了不影响读取的延迟, Aurora 会读取与缓存中一致的较旧的版本, 降低了读取的数据新鲜度; 而 PolarDB 因其存储的数据无版本且滞后于缓存版本, 所以往往需要将存储中拉上来的数据回放到与缓存一致的版本快照点, 存在一定的延迟, 但也带来新鲜度上的提升. 值得关注的是, 缓存一致性的数据库对于热点读比较友好, 当有 AP 查询到来时 Aurora 和 PolarDB 都会从缓存中直接读取所需数据.

## 3 HTAP 数据库系统数据共享优化

基于对以上各类 HTAP 数据库系统的技术总结, 本节重点阐述目前学术界针对现有的数据共享模型提出的可行的优化方向, 涉及版本同步、追踪、回收的各个阶段, 并归纳总结至表 3.

### 3.1 版本同步优化

对于 HTAP 数据库系统来说, TP 和 AP 端的强隔离可以实现性能的最大化, 但对操作版本的一致性要求会阻碍这一目标的实现, 故而 AP 与 TP 的版本同步的高效实现是核心技术点. HTAP 数据库系统中的版本同步相当于在 AP 端按可串行序进行一次对 TP 事务的回放, 从而保证同一时间点上数据的一致性. 最理想情况下, TP 对数据的修改实时为 AP 所见, 即版本同步时间为零. 然而, 实际情况下, 由于存储架构的差异, 不同同步方式对性能有着不同的影响.

在 3 种一致性模型下, 同步主要有两种形式, 基于日志回放的同步和基于数据拷贝的同步. 但值得注意的是,

一致性模型只约束同步的粒度和范围, 但不限制实现方式, 各类 HTAP 数据库系统可以按照负载的需求自主选择同步方式. 具体来说, 在同步粒度和范围的约束上, 线性一致性需要以细粒度的版本标识号方式进行数据同步, 而对于顺序一致性而言, 则可以放松同步粒度, 以批处理的方式同步数据, 会话一致性则进一步可以在每个会话内独立按需进行数据同步.

表 3 数据共享优化策略

| 数据共享优化策略 | 分类 | 优化方向 | 代表性工作 | 策略描述 | 优势 |
|---|---|---|---|---|---|
| 数据版本同步优化 | 基于日志回放的同步优化 | 日志持久化速率 | [52–54] | 基于新型存储介质加速落盘; 并行写入日志; 优化写入日志量与检查点间隔 | 减少持久化延迟, 缓解中心化日志写入压力, 也减小了日志同步量 |
| | | 日志回放效率 | [17] | 基于哈希映射确定回放位置; 基于字典压缩算法顺次回放 | 以较小的数据回放代价维护AP列存有序, 便于后续读取 |
| | 基于数据拷贝的同步优化 | 数据拷贝粒度 | [46] | 根据数据状态调整数据同步粒度 (表/分区等) | 提高同步时数据传输的灵活性 |
| | | 数据合并时间选取 | [55] | 基于深度学习根据负载状态自适应合并同步数据 | 缓解了数据合并带来的性能瓶颈 |
| 数据版本追踪优化 | 顺序扫描 | 多版本组织方式 | [29,56] | 基于多版本友好的新型数据结构组织数据 | 加速版本搜索 |
| | | 版本维护粒度 | [16,17,57] | 多粒度版本维护 (列/表) | 便于同时访问同一版本下的不同数据 |
| | | 冷热异构存储 | [58–60] | 调整LSM-Tree存储结构以优化冷热版本访问 | 加速对更新鲜的数据版本的访问 |
| | 索引扫描 | 多版本索引构建 | [61–65] | 基于版本信息优化B-Tree构建多版本索引 | 以少量的存储代价加速读取索引数据的多版本信息 |
| 数据版本回收优化 | — | 长事务版本识别 | [28,66,67] | 基于事务和版本生命周期关系提前释放因长事务挂起而无法被清理的版本 | 避免长事务阻碍垃圾回收进程 |

基于日志回放的同步主要包括串行回放和并行回放两种形式. 基于全局序的串行回放常见于 TP/AP 分离的数据库系统, 分批次将修改应用到 AP 端. 基于日志的并行回放是当前主流的解决方案, 比如 TiDB 基于 Raft Learner 的 Redo Log 并行回放、Aurora 基于 Quorum 的 Redo Log 缓冲区和存储并行回放、Vegito 基于 Epoch 的并行日志回放等.

基于数据拷贝的同步分为内存拷贝和网络拷贝. 内存拷贝常见于传统的单机数据库系统, 通过内存中行列变换或者是 TP/AP 负载使用同一套存储, AP 可以直接读取 TP 修改, 其数据新鲜度最高, 但是会在资源上与 TP 任务产生直接竞争. 网络拷贝则常见于分布式场景下, 节点之间进行数据传递, 其网络通信代价较高.

#### 3.1.1 基于日志回放的同步优化

基于日志回放的同步是数据库系统中最常用的同步方式, 它一般包括 3 个阶段: 日志持久化阶段、日志同步阶段和日志回放阶段. 对日志同步的优化目标是降低时延. 当前的主要优化方向包括并行化写入日志、降低检查点生成频率、减少日志量和升级日志写入介质等.

Haubenschild 等人[52]针对日志持久化阶段进行了优化. 其首先使用持久化内存作为存储介质, 利用其字节寻址的特性, 无需等待块满即可将日志落盘, 减少了日志持久化延迟; 其次将不同事务绑定到不同分区执行, 并设计了日志依赖处理规则, 使得多个分区可以并发地持久化日志. 最后, 其提出根据日志的生成速度决定检查点的间隔, 以减少脏页写回的次数. 对于写入日志量、检查点等技术的优化, 降低了日志持久化的时延.

Xia 等人[53]关注到高吞吐数据库在日志持久化阶段写日志时中心化发戳冲突、多核并发写时依赖事务日志全局排序这两方面的问题, 提出使用局部日志戳发放和 LSN Vector 进行依赖关系的传递, 缓解多核并发日志写竞争. Jung 等人[54]发现多核并发日志推进时会有未补齐的空洞, 影响持久化效率, 采用 grasshopping 算法推进补齐

点, 让中心化的日志记录更快、扩展性更好.

Boroumand 等人[17]提出 Polynesia 优化回放日志更新到列存数据的时延. 对于日志同步阶段, 其将列存数据上的日志同步分为 3 个阶段, 分别为合并, 定位和复制. 合并阶段首先通过扫描各个线程的日志, 按提交 ID 顺序排序成一个最终日志; 定位阶段, 将最终日志上的每一项通过哈希映射逐个发送到各自需要修改的列对应的列缓冲区中; 复制阶段, 将这些列缓冲区中的日志传输至 AP 端副本进行数据更新. 接下来, 对于日志回放阶段, 为了维护列式存储的有序且解决列规模太大易带来的排序代价过高问题, 该工作通过将原先为了压缩而维护的字典和日志回放的更新字典进行合并, 用于原来的列存数据, 最终将原先排序更新的时间压缩至线性空间, 且该工作通过软硬件协同的方式加速了回放的速率.

#### 3.1.2 基于数据拷贝的同步优化

基于日志回放的同步方式, 虽然比较灵活且在空间开销上获得了较大优势, 但是无法避免日志存取的 IO 开销, 而基于数据拷贝的同步方式往往按批实现 AP 与 TP 端数据的一致性, 以磁盘块为单位的读取在 IO 开销上有其独特的优势, 因此对于改动量小且分布位置不集中的场景来说, 回放日志引起的数据复制传输量小, 较为灵活, 而对于修改位置集中的批量更改来说, 使用全表级或分区级别的数据复制传输能够有效减少 IO 开销[46], 更适合在 Epoch、ETL 或者类 ETL 的批量同步中使用.

IBM 的 IDAA[46]在混合负载处理上, 除了通过 CDC 监视事务日志、对日志以指定时间间隔的微批次实现增量更新外, 对于 AP 端数据的大批量初始化、大变更、无法明确一个分区是否被更改情况, 可以以不同粒度 (以表或分区为单位) 进行数据同步. 所以是否使用大批量数据同步和数据修改量的大小、分布等有关系工作[55]提出同步回放时增量数据和历史数据的合并时间受多种因素影响难以预测, 且合并时需要考虑不同分区负载的平衡, 设计了一种基于学习的 (learning-based) 框架和多轮平衡策略, 以实现自适应的同步数据合并.

### 3.2 版本追踪优化

MVCC 和 LSM-Tree 正被广泛应用于各类数据库中, 并面向 OLTP 负载展开了一系列优化. MVCC 有效地控制了 TP 写入的数据版本和 AP 读取版本的数据可见性, 提高了事务并发; 而 LSM-Tree 良好的顺序写入磁盘的优势也极大优化了写密集的 TP 场景性能. 但这两种技术都无法高效地支持 AP 大规模扫描任务.

首先, 传统的 MVCC 机制并没有对 TP 和 AP 负载进行明确区分, 混合负载往往共享一套版本链, 这使得在实际执行时会面临诸多挑战. TP 负载大多是单点读写而 AP 负载大多有范围扫描, 这种差异使得版本管理本身存在矛盾, 优化很难兼顾. 并且随着 TP 负载的写入增加, 版本数量急剧增长, AP 负载在遍历和选择版本时就将会遇到较大的扫描代价[57,66]. 另一方面, AP 查询若长期持有某个版本快照, 使得垃圾回收机制无法及时有效地进行版本清理, 版本的持续积累也已被论证会给系统带来极大性能降级[28], 进而影响 TP 负载写入性能.

其次, LSM-Tree 设计的分层结构会随着时间的推移将较冷的数据向底层迁移, 由于数据的不同版本可能由于合并时间的不同分散在 LSM-Tree 的不同层中, 难以直接定位和访问. 因此对数据进行读取常会涉及多次随机访问, 降低了数据的读取性能[58], 不利于对批量数据的扫描和定位.

#### 3.2.1 顺序扫描

针对版本链追踪的问题, 传统的 MVCC 设计会以每一个元组为单位独立构建版本链. 当进行扫描时, 其追溯的代价非常大. 所以 Kim 等人[29]设计了一个被称为 vWeaver 的数据结构, 在面向同一个数据项的版本链追踪上结合概率模型改进了跳表, 将搜索时间复杂度缩减为 O(lg$N$); 在不同数据记录之间的版本链中引入额外的指针连接, 基于搜索大概率会发生在相邻数据版本之间的假设, 将记录间的相邻版本进行连接, 因此整体的版本链就被编织串接起来, 在一定程度上加速了版本链的追踪和扫描.

Kim 等人[56]同时考虑了版本链扫描和版本回收两个问题, 指出传统的 MVCC 设计中快速的版本清理会导致频繁的磁盘 IO, 同时造成版本链断裂使得版本链需要修复重组才能继续检索, 也会影响版本的追踪与定位效率. 因此他们提出可以将版本追踪和版本清理两个动作分离开, 一方面沿袭了将最新版本与正在修改的数据置放在一起并将剩余的旧版本独立存储于额外空间的思想[67]; 另一方面借鉴传统 UNIX 文件系统中 Inode 索引的设计理念[68], 构建了一套轻量的版本索引结构加速对旧版本块的搜索, 将数据版本与索引版本分离开; 同时设计了一种类似线

段树的树状结构, 基于数据旧版本可见性与活跃事务所在的时间分段 (Epoch) 的比较进行适时批量的垃圾回收. 这种解耦式设计以引入额外的存储和维护代价换取了 AP 的处理效率.

Sharma 等人[57]提出在一套系统内 TP 与 AP 负载不应该共享同一套版本链, 通过在 TP 端维护最新的版本和在 AP 端维护一系列版本快照的方式来解决 MVCC 的弊端. 具体来说, TP 负载的执行仍基于 MVCC 维护版本链, 而当 AP 负载到来时, 系统会基于最新的版本在 TP 端建立一份虚拟快照以便修改, 而原有的版本链会逻辑地移动到 AP 端. 最终在不阻塞 TP 写入的情况下, 通过在 AP 端维护多个快照版本链服务于 AP 负载的访问请求, 实现对混合负载的支持. 这种方式一方面大大缩减了版本链搜寻的长度, 另一方面垃圾回收机制可以自然地对过期的快照进行所有版本的回收清理, 避免了在原版本链中定位垃圾版本的繁复工作. 但该方法可能会因为建立快照期间需要大量的互斥锁而降低系统吞吐. 同时, 也出现了一系列针对快照维护粒度的研究工作. 对于版本链完全分离的追踪优化, Shen 等人[16]基于真实业务中 AP 查询一般会访问最新数据版本的判断, 提出可以在 AP 端为每个版本以表的粒度独立维护一个快照, 这样当 AP 查询到来时, 可以线性地针对某个稳定版本进行扫描, 避免多版本链的扫描代价. 但这种设计的劣势主要在于尽管表的快照会以增量的形式进行维护, 但其粗粒度带来的额外存储代价依然是非常可观的. Boroumand 等人[17]则提出 TP 的更新往往会呈现一定的业务特征, 如对同一列的数据进行数据更新. 所以 AP 端可以以列为粒度维护版本链, 而不需要更新未被修改的列版本, 从而避免为这些列重复生成相同的快照, 适当降低了创建快照的频率. 其本质属于对表粒度和行粒度维护版本链的一种权衡实现.

一些 HTAP 数据库系统的 AP 端与 TP 端一样使用 LSM-Tree 进行数据组织, 但也针对 AP 负载的特性对其结构进行优化, 以加快读取的性能. 由于 LSM-Tree 天然会随着时间将数据向底层迁移压缩, 在这种机制下, 上层的数据版本较新, 但会面临频繁的更新, 低层的数据版本较旧但稳定. 对此, Saxena 等人[58]优化了 LSM-Tree 的结构, 使其每层支持不同的存储格式, 使用行存的方式存储上层的数据, 便于快速修改; 在合并上层的数据进入下层级时, 数据的存储格式变为列存, 并使 AP 查询直接访问下层的旧版本. 尽管降低了数据访问新鲜度但可以提高 AP 的读取性能. Dai 等人[59]则通过逐层训练预测模型的方式快速预测需要读取的数据位置. Zhong 等人[60]则提出一种称为 REMIX 的索引数据结构, 通过在 LSM-Tree 存储结构中加入层内和层间的跳转指针支持了数据的有序访问, 包括范围扫描和多粒度的二分查找, 减少了遍历的代价, 最终加速 AP 数据读取.

#### 3.2.2 索引扫描

除却顺序扫描, 索引扫描也是支持高效访问数据的技术之一. 但在数据库中建立索引也会带来额外的维护代价. 与传统的数据相对稳定的 OLAP 数据库索引维护代价相比, 混合负载中 TP 负载会引入大量的数据版本更新, 增加索引的存储和维护代价[61]. 与此同时, 存在多个数据版本时, 存储需要判定每个数据项的各个版本的可见性, 这大大降低了索引访问性能. 因此, 需要设计考虑版本信息的新型索引结构来提高数据读取效率.

为了避免对数据版本的遍历, 需要将版本信息引入索引, 进而产生了多种不同的优化方法. 时间分割 B 树[62]在 B 树的基础上扩展了版本维度, 支持从版本和主键两个维度分别对索引进行分割. 当数据项的版本更新时, 在版本维度上基于 B 树追加版本而非修改数据, 以提高检索版本的效率. MVBT[61]则在 B 树的基础上增加了版本链, 在索引中维护数据项的多个版本, 并顺序连接以方便检索. MV-IDX[63]则将版本链直接连接在索引对应的数据项后, 可以直接检索的同时减少了索引维护的代价. 受到这些改进的启发, Riegger 等人[64]在分区 B 树的基础上实现了版本可感知的多版本分区 B 树 (MV-PBT), 这一数据结构将版本作为分区的依据, 将不同版本的数据划分在不同的索引分区中. 当发生了版本更新时, 数据库将新的版本数据插入最新分区, 并同时维护对应的旧版本映射. 当需要进行搜索时, MV-PBT 按分区从新到旧进行扫描, 直到找到满足可见性的版本. 这些优化通过将版本信息维护在索引中, 在面对混合负载导致的频繁版本更新时有效减免了判断版本可见性带来的开销. 但另一方面, 由于索引中增加了版本信息, 其维护代价也有所提高. 为此, Sun 等人[65]提出一个新的索引结构 P 树, 既将版本信息引入索引结构中, 又通过对多个索引的嵌套实现对索引中部分数据项的复用, 降低了修改索引的成本. 当数据库产生一个新版本时, 索引采用 COW 的方式生成一棵新的索引树, 并将没有修改的数据项指针指向旧版本索引, 更新新版本数据对应的指针, 从而轻量地维护版本信息. 通过这种方式, 数据库将索引与版本一一对应, 便于根据版本搜索索引, 从而避免了在判定可见性时遍历数据项的所有版本.

### 3.3 版本回收优化

SAP HANA、Hyrise、HyPer 等数据库通过对 MVCC 的优化, 提高了混合负载下版本追踪的效率. 但如何有效地回收旧版本, 避免长事务带来性能下降依然是个需要解决的难题.

垃圾回收的关键问题是如何确定垃圾数据版本. 传统的做法是维护一个全局活跃事务最小时间戳 $T_{\min}$, 若版本戳大于 $T_{\min}$, 则版本被继续保留, 也因而很容易被某个启动较早但执行较长的事务阻碍垃圾回收进程. Lee 等人[66]提出一种基于间隔 (interval) 的回收机制, 使用一个间隔区间 [$s$, $e$) 标识某个数据项 $v$ 在不同快照版本下可被观测到的时间戳范围. 比如 $v_1$ 版本时, $v_2$ 为 4, 则 $v_1$ 对于启动时间戳在 [1, 4) 的事务可见. 与此同时, 如果在当前全局活跃的事务集合中, 并没有落在该戳间隔内的事务, 则该区间内 $v_1$ 的版本可以被回收. 这样就可以解除对全局最小时间戳的依赖, 也缓解了长事务造成中间版本不能回收的问题.

Bottcher 等人[28]也取缔了维护全局活跃事务集合的方法, 让不同的线程独立维护不相交的活跃事务子集, 以 O(#threads) 的时间复杂度确定全局最小时间戳, 一定程度上减小了全局共享互斥锁的竞争, 提高可扩展性; 另一方面通过在版本链中确定发起查询时当前可见的版本戳 $v_{\text{visible}}$ 以及正在写入的最新版本戳 $v_{\text{current}}$, 遍历 [$v_{\text{visible}}$, $v_{\text{current}}$] 区间内每个时间点下的数据版本是否在 $v_{\text{visible}}$ 之后有修改. 如果没有修改, 则其不会被任何事务访问删除. 论文也指出对于垃圾回收机制来说, 清理的粒度、执行的频率、精度等方面也会对性能产生较大的影响, 需要审慎考量.

Kim 等人[67]提出可以将最近可能会被访问的一个旧版本与数据元组共同存储在一起 (in-row), 剩余多个版本则存储在额外的存储空间内 (off-row). 同时, 他们面对 off-row 中旧版本回收问题, 也提出了与文献 [66] 类似的理念, 即比较版本生命周期和事务启动时间戳范围来确定版本是否可回收, 从而避免了长事务挂起的版本清理阻塞. 同时, 该工作在上述 in-row 和 off-row 中间引入了额外的版本缓冲空间 (buffer), 在 buffer 内提前对数据版本进行分类和预清理, 基于一定的阈值和事务执行时间判断该版本属于热版本、冷版本还是长事务持有的版本, 进而对其进行不同粒度的剪裁和清理, 降低长事务对整体性能的影响.

## 4 研究展望与未来方向

不同的一致性模型对应不同的数据共享方式, 反映为 TP 写更新版本和 AP 版本读版本之间的版本标识号差异程度, 即最新更新版本和当前读版本的版本标识号差距 $p$, 以及同一时刻多个读版本的版本标识号差距 $q$, 如表 4 所示. $p$ 和 $q$ 的不同取值决定了数据库系统对性能和新鲜度指标的权衡. 本文从数据生命周期中最为重要的 3 个子问题出发, 对性能和新鲜度这两个指标的权衡进行了总结.

在数据的产生阶段, 即 TP 端的写事务生产新数据版本阶段, 一致性模型主要影响标识号的分配. 在线性一致性模型下, TP 和 AP 端的标识号同步分发, 从而保证高新鲜度, 即共享模型中 $p$=0 且 $q$=0, 这也同时要求实现快速的版本同步, 会对数据库的性能造成一定的影响; 在顺序一致性和会话一致性中, AP 端在回放产生数据版本时推进对应的标识号, 因此 AP 端的标识号与 TP 端相比可以存在一定的时延, 即 $p$=$n$, 导致了一定的新鲜度损失, 但也同时减轻了同步的压力, 有利于稳定数据库性能. 不同的标识号分配机制对应了数据新鲜度和数据库性能之间不同的权衡策略. 总而言之, 模型对一致性的要求越高, AP 端的标识号推进速度越快, 对数据库性能的影响越大. 因此, 根据应用场景对数据新鲜度和数据库性能的不同要求, 需要选择不同的数据库一致性模型.

表 4 不同一致性模型下数据共享模型的约束

| 一致性级别 | $p$ | $q$ |
|---|---|---|
| 线性一致性 | 0 | 0 |
| 带新鲜度阈值约束的一致性 | $\epsilon$ | 0 |
| 顺序一致性 | $n$ | 0 |
| 会话一致性 | $n$ | $n$ |

在数据的同步阶段, 即写副本同步到读副本或备份副本的过程中, 主要的同步方式分为日志同步和数据同步两种. 其中多数数据库日志同步采用将写节点的日志持久化后同步到其他节点上, 随后在对应节点回放日志的方

式推进多副本的版本. 这一阶段的优化主要是降低日志持久化和回放的耗时. 数据同步则是通过直接拷贝新数据版本快照的方式更新版本, 通过控制拷贝快照粒度减少拷贝的代价, 从而提高同步的效率.

在数据的消费阶段, 即分析查询访问数据阶段, 优化主要针对版本的组织方式. 版本组织需要解决 HTAP 场景下快速产生大量版本增加分析查询的版本扫描负担, 以及长分析查询导致的大量版本无法及时回收等问题. 学术界提出了多种新颖的数据结构以优化版本扫描效率的算法. 对于 LSM-Tree 的优化主要解决如何加速从树的多层结构里高效追踪某个数据版本的问题. 而索引扫描方面则重点在于结合版本和索引的信息加速 AP 的读取, 但当前的数据库产品大多采用简单的链式存储方案, 存在一定的优化空间. 对于版本回收的问题, 也有研究致力于准确识别和删除无效版本, 从而提高版本回收的效率.

总的来说, 目前 HTAP 数据库系统的优化主要集中于降低版本同步开销和多版本查询的代价. 前者关注一致性模型中标识号推进以及版本同步的流程; 后者关注在混合负载上支持高效的写入和查询. 针对 HTAP 场景的特征以及现有技术总结, 本文提出 3 个值得进一步关注的优化方向, 即数据同步范围自适应, 数据同步周期自调优以及顺序一致性下的新鲜度阈值约束控制.

### 4.1 数据同步范围自适应

主流的分离式 HTAP 数据库系统大都会直接将 TP 最新写入的修改批量同步到 AP 端进行数据重组和更新. 这一同步策略应用的前提是 TP/AP 端对数据的需求重合度较高, 且该批数据具有较高应用价值. 但实际上, TP/AP 端读写的热点数据未必保持一致, 即耗费大量资源进行数据同步可能并不能达到最大收益. 因此, 结合 TP/AP 端负载的特征和数据的状态, 设计自适应的数据同步策略, 按需进行数据的快速同步是 HTAP 数据库系统在性能优化时值得关注的问题.

### 4.2 数据同步周期自调优

HTAP 数据库系统通过日志在进行数据同步时, 往往需要设置一个同步周期间隔以实现对 TP 更新的批量传输, 如 Vegito[16]提出在设置相对较小的 Epoch 间隔时, 会由于频繁的小批量同步对 TP 端带来较大的性能影响, 反之较大的 Epoch 间隔则会带来较低的数据新鲜度. 所以同步周期的划分粒度同样需要在性能和新鲜度之间做出有效的权衡. 而近年来, 深度学习在系统自调优上已经有了非常广泛的应用[69–71], 通过负载预测和系统性能监测等方式可以实现系统自主调优, 以维持一个稳定的性能状态. 此类方法同样可以尝试应用在 HTAP 数据库系统同步周期的选取上, 通过实时监测系统性能水平, 数据新鲜度, 底层数据状态, 负载特征等信息对数据同步周期进行自调优, 以实现较好的同步效果.

### 4.3 顺序一致性的新鲜度阈值约束控制

不同一致性模型对读取版本的要求综合考虑了系统性能和数据新鲜度的需求, 同时具有不同的数据共享约束. 对于大部分的实际业务而言, 并不要求极高的数据新鲜度, 只需要保证数据新鲜度能满足业务场景的需要即可. 目前大部分 HTAP 数据库系统受一致性模型的约束, 完美的数据新鲜度和高吞吐往往不能同时保证. 线性一致性模型适用于对数据新鲜度高要求的场景, 但对数据库的吞吐和延迟有较大影响; 顺序一致性模型中无法严格保证新鲜度, 容易变为对 OLAP 数据库的简单加强, 无法实现实时分析的目标. OceanBase 提出的有界旧一致性读模型[72]和 IBM 的 IDAA[46]都对这一问题进行了探索. 在该一致性模型下, 若 AP 端同步速度过慢, 导致数据新鲜度低于某一阈值时, 数据库选择直接从 TP 端读取最新的数据, 在牺牲分析性能的同时保证数据新鲜度稳定在阈值之上. 但可能会遇到需要长期在 TP 端执行分析查询, 导致 TP 端性能降低以及对 AP 端资源的浪费. MongoDB 提出的 Decongestant[73]采用了类似的思路, 支持在主副本所在节点压力过大时读取其他节点中较不新鲜的副本, 但同样要求当其他节点新鲜度过低时只能在主副本读取. 因此, 结合 TP/AP 负载对吞吐和数据新鲜度的需求, 在特定的一致性机制下对数据新鲜度的阈值进行动态选择, 使 AP 端读取的版本与 TP 端最新版本的差距 $p$ 在某个既定阈值 $\epsilon$ $(0 \leqslant \epsilon \leqslant n)$ 之内 (如表 4 所示), 从而在提高性能的同时满足应用对数据新鲜度的需求是 HTAP 数据库系统在设计和实现一致性机制时值得关注的问题.

**References:**


[1] Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. 2014. https://www.gartner.com/en/documents/2657815

[2] What is hybrid transaction/analytical processing (HTAP)? 2014. https://www.zdnet.com/paid-content/article/what-is-hybrid-transactionanalytical-processing-htap/

[3] Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S. Hyrise: A main memory hybrid storage engine. Proc. of the VLDB Endowment, 2010, 4(2): 105–116. [doi: 10.14778/1921071.1921077]

[4] Färber F, Cha SK, Primsch J, Bornhövd C, Sigg S, Lehner W. SAP HANA database: Data management for modern business applications. ACM Sigmod Record, 2011, 40(4): 45–51. [doi: 10.1145/2094114.2094126]

[5] Kemper A, Neumann T. HyPer: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering. Hannover: IEEE, 2011. 195–206. [doi: 10.1109/ICDE.2011.5767867]

[6] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 677–689. [doi: 10.1145/2723372.2749436]

[7] Lahiri T, Chavan S, Colgan M, Das D, Ganesh A, Gleeson M, Hase S, Holloway A, Kamp J, Lee TH, Loaiza J, Macnaughton N, Marwah V, Mukherjee N, Mullick A, Muthulingam S, Raja V, Roth M, Soylemez E, Zait M. Oracle database in-memory: A dual format in-memory database. In: Proc. of the 31st IEEE Int'l Conf. on Data Engineering. 2015. 1253–1258. [doi: 10.1109/ICDE.2015.7113373]

[8] Larson PÅ, Birka A, Hanson EN, Huang WY, Nowakiewicz M, Papadimos V. Real-time analytical processing with SQL server. Proc. of the VLDB Endowment, 2015, 8(12): 1740–1751. [doi: 10.14778/2824032.2824071]

[9] Lyu Z, Zhang HH, Xiong G, Guo G, Wang HZ, Chen JB, Praveen A, Yang Y, Gao XM, Wang A, Lin W, Agrawal A, Yang JF, Wu H, Li XL, Guo F, Wu J, Zhang J, Raghavan V. Greenplum: A hybrid database for transactional and analytical workloads. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2530–2542. [doi: 10.1145/3448016.3457562]

[10] Nugroho DPA, Ismail HA. In-memory database and MemSQL. 2019. https://cs.ulb.ac.be/public/_media/teaching/infoh415/student_projects/2019/memsql.pdf

[11] Zhou JY, Xu M, Shraer A, Namasivayam B, Miller A, Tschannen E, Atherton S, Beamon AJ, Sears R, Leach J, Rosenthal D, Dong X, Wilson W, Collins B, Scherer D, Grieser A, Liu YN, Moore A, Muppana B, Su XG, Yadav V. FoundationDB: A distributed unbundled transactional key value store. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2653–2666. [doi: 10.1145/3448016.3457559]

[12] Huang DX, Liu Q, Cui Q, Fang ZH, Ma XY, Xu F, Shen L, Tang L, Zhou YX, Huang ML, Wei W, Liu C, Zhang J, Li JJ, Wu XL, Song LY, Sun RX, Yu SP, Zhao L, Cameron N, Pei LQ, Tang X. TiDB: A raft-based htap database. Proc. of the VLDB Endowment, 2020, 13(12): 3072–3084. [doi: 10.14778/3415478.3415535]

[13] Yang JC, Rae I, Xu J, Shute J, Yuan Z, Lau K, Zeng Q, Zhao X, Ma J, Chen ZY, Gaoy, Dong QL, Zhou JX, Wood J, Graefe G, Naughton JF, Cieslewicz J. F1 Lightning: HTAP as a service. Proc. of the VLDB Endowment, 2020, 13(12): 3313–3325. [doi: 10.14778/3415478.3415553]

[14] Cao W, Liu ZJ, Wang P, Chen S, Zhu CF, Zheng S, Wang YH, Ma GQ. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proc. of the VLDB Endowment, 2018, 11(12): 1849–1862. [doi: 10.14778/3229863.3229872]

[15] Verbitski A, Gupta A, Saha D, Corey J, Gupta K, Brahmadesam M, Mittal R, Krishnamurthy S, Maurice S, Kharatishvilli T, Bao XF. Amazon aurora: On avoiding distributed consensus for I/Os, commits, and membership changes. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 789–796. [doi: 10.1145/3183713.3196937]

[16] Shen SJ, Chen R, Chen HB, Zang BY. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In: Proc. of the 15th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2021). 2021. 219–238.

[17] Boroumand A, Ghose S, Oliveira GF, Mutlu O. Polynesia: Enabling effective hybrid transactional/analytical databases with specialized hardware/software co-design. arXiv:2103.00798, 2021.

[18] Jing CH, Liu WJ, Gao JT, Pei OY. Research and implementation of HTAP for distributed database. Journal of Northwestern Polytechnical University, 2021, 39(2): 430–438 (in Chinese with English abstract). [doi: 10.1051/jnwpu/20213920430]

[19] Milkai E, Chronis Y, Gaffney KP, Guo ZH, Patel JM, Yu XY. How good is my HTAP system? In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 1810–1824. [doi: 10.1145/3514221.3526148]

[20] Makreshanski D, Giceva J, Barthels C, Alonso G. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 37–50. [doi: 10.1145/3035918.3035959]

[21] Real-time access to real-time information. 2015. https://www.hunkler.de/files/downloads/oracle_wp_golden-gate-12c-real-t.pdf

[22] Raza A, Chrysogelos P, Anadiotis AC, Ailamaki A. Adaptive htap through elastic resource scheduling. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 2043–2054. [doi: 10.1145/3318464.3389783]

[23] Coelho F, Paulo J, Vilaça R, Pereira J, Oliveira R. HTAPBench: Hybrid transactional and analytical processing benchmark. In: Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering. L'Aquila: ACM, 2017. 293–304. [doi: 10.1145/3030207.3030228]

[24] Sirin U, Dwarkadas S, Ailamaki A. Performance characterization of HTAP workloads. In: Proc. of the 37th IEEE Int'l Conf. on Data Engineering (ICDE). Chania: IEEE, 2021. 1829–1834. [doi: 10.1109/ICDE51399.2021.00162]

[25] Cole R, Funke F, Giakoumakis L, Guy W, Kemper A, Krompass S, Kuno H, Nambiar R, Neumann T, Poess M, Sattler KU, Seibold M, Simon E, Waas F. The mixed workload CH-benchmark. In: Proc. of the 4th Int'l Workshop on Testing Database Systems. Athens: ACM, 2011. 8. [doi: 10.1145/1988842.1988850]

[26] Wu YJ, Arulraj J, Lin JX, Xian R, Pavlo A. An empirical evaluation of in-memory multi-version concurrency control. Proc. of the VLDB Endowment, 2017, 10(7): 781–792. [doi: 10.14778/3067421.3067427]

[27] Vinçon T, Knödler C, Solis-Vasquez L, Bernhardt A, Tamimi S, Weber L, Stock F, Koch A, Petrov I. Near-data processing in database systems on native computational storage under HTAP workloads. Proc. of the VLDB Endowment, 2022, 15(10): 1991–2004. [doi: 10.14778/3547305.3547307]

[28] Böttcher J, Leis V, Neumann T, Kemper A. Scalable garbage collection for in-memory mvcc systems. Proc. of the VLDB Endowment, 2019, 13(2): 128–141. [doi: 10.14778/3364324.3364328]

[29] Kim J, Kim K, Cho H, Yu J, Kang S, Jung H. Rethink the scan in MVCC databases. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 938–950. [doi: 10.1145/3448016.3452783]

[30] Özcan F, Tian YY, Tözün P. Hybrid transactional/analytical processing: A survey. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. ACM, 2017. 1771–1775. [doi: 10.1145/3035918.3054784]

[31] Hieber D, Grambow G. Hybrid transactional and analytical processing databases: A systematic literature review. In: Proc. of the 9th Int'l Conf. on Data Analytics. Nice: IARIA, 2020. 90–98.

[32] Psaroudakis I, Wolf F, May N, Neumann T, Böhm A, Ailamaki A, Sattler KU. Scaling up mixed workloads: A battle of data freshness, flexibility, and scheduling. In: Proc. of the 6th Technology Conf. on Performance Evaluation and Benchmarking. Hangzhou: Springer, 2014. 97–112. [doi: 10.1007/978-3-319-15350-6_7]

[33] Zhang C, Li GL, Feng JH, Zhang JT. A survey of key techniques of HTAP databases. Ruan Jian Xue Bao/Journal of Software, 2023, 34(2): 761–785 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6713.htm [doi: 10.13328/j.cnki.jos.006713]

[34] Li GL, Zhang C. HTAP databases: What is new and what is next. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 2483–2488. [doi: 10.1145/3514221.3522565]

[35] Li GL, Zhou XH, Sun J, Yu X, Yuan HT, Liu JB, Han Y. A survey of machine learning based database techniques. Chinese Journal of Computers, 2020, 43(11): 2019–2049 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2020.02019]

[36] Terry D. Replicated data consistency explained through baseball. Communications of the ACM, 2013, 56(12): 82–89. [doi: 10.1145/2500500]

[37] Herlihy MP, Wing JM. Linearizability: A correctness condition for concurrent objects. ACM Trans. on Programming Languages and Systems, 1990, 12(3): 463–492. [doi: 10.1145/78969.78972]

[38] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. ACM SIGACT News, 2002, 33(2): 51–59. [doi: 10.1145/564585.564601]

[39] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications. In: Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation. Vancouver: USENIX Association, 2010. 251–264.

[40] Prout A, Wang SP, Victor J, Sun Z, Li YZ, Chen J, Bergeron E, Hanson E, Walzer R, Gomes R, Shamgunov N. Cloud-native transactions and analytics in singleStore. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 2340–2352. [doi: 10.1145/3514221.3526055]

[41] Chen J, Jindel S, Walzer R, Sen R, Jimsheleishvilli N, Andrews M. The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. Proc. of the VLDB Endowment, 2016, 9(13): 1401–1412. [doi: 10.14778/3007263.3007277]

[42] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proc. of the 2014 USENIX Annual Technical Conf. Philadelphia: USENIX Association, 2014. 305–319.

[43] TiFlash architecture and principles. PingCAP. 2022 (in Chinese). https://book.tidb.io/session1/chapter9/tiflash-architecture.html

[44] Corbett JC, Dean J, Epstein M, *et al*. Spanner: Google's globally distributed database. ACM Trans. on Computer Systems, 2013, 31(3): 8. [doi: 10.1145/2491245]

[45] Lamport L. The part-time parliament. In: Malkhi D, ed. Concurrency: The Works of Leslie Lamport. New York: ACM, 2019. 277–317.

[doi: 10.1145/3335772.3335939]

[46] Butterstein D, Martin D, Stolze K, Beier F, Zhong J, Wang LY. Replication at the speed of change: A fast, scalable replication solution for near real-time HTAP processing. Proc. of the VLDB Endowment, 2020, 13(12): 3245–3257. [doi: 10.14778/3415478.3415548]

[47] Chen JJ, Ding YH, Liu Y, Li FS, Zhang L, Zhang MY, Wei K, Cao LX, Zou D, Liu Y, Zhang L, Shi R, Ding W, Wu K, Luo SY, Sun J, Liang YM. ByteHTAP: Bytedance's HTAP system with high data freshness and strong data consistency. Proc. of the VLDB Endowment, 2022, 15(12): 3411–3424. [doi: 10.14778/3554821.3554832]

[48] MySQL Heatwave. Real-time analytics for MySQL database service. 2021. https://www.mysql.com/products/mysqlheatwave/

[49] Verbitski A, Gupta A, Saha D, Brahmadesam M. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1041–1052. [doi: 10.1145/3035918.3056101]

[50] Gifford DK. Weighted voting for replicated data. In: Proc. of the 7th ACM Symp. on Operating Systems Principles. Pacific Grove: ACM, 1979. 150–162. [doi: 10.1145/800215.806583]

[51] Buffer space management. PolarDB. 2022 (in Chinese). https://apsaradb.github.io/PolarDB-for-PostgreSQL/zh/theory/buffer-management.html#lazy-checkpoint

[52] Haubenschild M, Sauer C, Neumann T, Leis V. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 877–892. [doi: 10.1145/3318464.3389716]

[53] Xia Y, Yu XY, Pavlo A, Devadas S. Taurus: Lightweight parallel logging for in-memory database management systems. Proc. of the VLDB Endowment, 2021, 14(2): 189–201. [doi: 10.14778/3425879.3425889]

[54] Jung H, Han H, Kang S. Scalable database logging for multicores. Proc. of the VLDB Endowment, 2017, 11(2): 135–148. [doi: 10.14778/3149193.3149195]

[55] He XL, Cai P, Zhou X, Zhou AY. Continuously bulk loading over range partitioned tables for large scale historical data. In: Proc. of the 37th IEEE Int'l Conf. on Data Engineering (ICDE). Chania: IEEE, 2021. 960–971. [doi: 10.1109/ICDE51399.2021.00088]

[56] Kim J, Yu J, Ahn J, Kang S, Jung H. Diva: Making MVCC systems HTAP-friendly. In: Proc. of the 2022 Int'l Conf. on Management of Data. Philadelphia: ACM, 2022. 49–64. [doi: 10.1145/3514221.3526135]

[57] Sharma A, Schuhknecht FM, Dittrich J. Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 245–258. [doi: 10.1145/3183713.3196904]

[58] Saxena H, Golab L, Idreos S, Ilyas IF. Real-time LSM-trees for htap workloads. arXiv:2101.06801, 2021.

[59] Dai YF, Xu YE, Ganesan A, Alagappan R, Kroth B, Arpaci-Dusseau AC, Arpaci-Dusseau RH. From wiscKey to bourbon: A learned index for log-structured merge trees. In: Proc. of the 14th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2020. 9.

[60] Zhong WS, Chen C, Wu XB, Jiang S. REMIX: Efficient range query for LSM-trees. In: Proc. of the 19th USENIX Conf. on File and Storage Technologies. USENIX Association, 2021. 51–64.

[61] Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P. An asymptotically optimal multiversion B-tree. The VLDB Journal, 1996, 5(4): 264–275. [doi: 10.1007/s007780050028]

[62] Lomet D, Salzberg B. The performance of a multiversion access method. In: Proc. of the 1990 ACM SIGMOD Int'l Conf. on Management of Data. Atlantic City: ACM, 1990. 353–363. [doi: 10.1145/93597.98744]

[63] Gottstein R, Goyal R, Hardock S, Petrov I, Buchmann A. MV-IDX: Indexing in multi-version databases. In: Proc. of the 18th Int'l Database Engineering & Applications Symp. Porto: ACM, 2014. 142–148. [doi: 10.1145/2628194.2628911]

[64] Riegger C, Vinçon T, Gottstein R, Petrov I. MV-PBT: Multi-version indexing for large datasets and HTAP workloads. In: Proc. of the 23rd Int'l Conf. on Extending Database Technology. Copenhagen: OpenProceedings.org, 2020. 217–228.

[65] Sun YH, Blelloch GE, Lim WS, Pavlo A. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. Proc. of the VLDB Endowment, 2019, 13(2): 211–225. [doi: 10.14778/3364324.3364334]

[66] Lee J, Shin H, Park CG, Ko S, Noh J, Chuh Y, Stephan W, Han WS. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1307–1318. [doi: 10.1145/2882903.2903734]

[67] Kim J, Cho H, Kim K, Yu J, Kang S, Jung H. Long-lived transactions made less harmful. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 495–510. [doi: 10.1145/3318464.3389714]

[68] Ritchie DM, Thompson K. The UNIX time-sharing system. The Bell System Technical Journal, 1978, 57(6): 1905–1929. [doi: 10.1002/j.1538-7305.1978.tb02136.x]

[69] Pavlo A, Butrovich M, Ma L, Menon P, Lim WS, Van Aken D, Zhang W. Make your database system dream of electric sheep: Towards

self-driving operation. Proc. of the VLDB Endowment, 2021, 14(2): 3211–3221. [doi: 10.14778/3476311.3476411]

[70] Pavlo A, Pavlo A, Angulo G, Arulraj J, Lin HB, Lin JX, Ma L, Menon P, Mowry TC, Perron M, Quah I, Santurkar S, Tomasic A, Toor S, van Aken D, Wang ZQ, Wu YJ, Xian R, Zhang TY. Self-driving database management systems. In: Proc. of the 8th Biennial Conf. on Innovative Data Systems Research. Chaminade, 2017. 1–6.

[71] Vargas-Solar G, Zechinelli-Martini JL, Espinosa-Oviedo JA. Big data management: What to keep from the past to face future challenges? Data Science and Engineering, 2017, 2(4): 328–345. [doi: 10.1007/s41019-017-0043-3]

[72] Yang ZF. What is the consistency of distributed storage systems? 2018 (in Chinese). https://zhuanlan.zhihu.com/p/34656939

[73] Huang CH, Cahill M, Fekete A, Röhm U. Decongestant: A breath of fresh air for MongoDB through freshness-aware reads. 2021. https://openproceedings.org/2021/conf/edbt/p135.pdf

**附中文参考文献**:

[18] 景荟弘, 刘文洁, 高锦涛, 裴欧亚. 面向分布式数据库的HTAP研究与实现. 西北工业大学学报, 2021, 39(2): 430–438. [doi: 10.1051/jnwpu/20213920430]

[33] 张超, 李国良, 冯建华, 张金涛. HTAP数据库关键技术综述. 软件学报, 2023, 34(2): 761–785. http://www.jos.org.cn/1000-9825/6713.htm [doi: 10.13328/j.cnki.jos.006713]

[35] 李国良, 周煊赫, 孙佶, 余翔, 袁海涛, 刘佳斌, 韩越. 基于机器学习的数据库技术综述. 计算机学报, 2020, 43(11): 2019–2049. [doi: 10.11897/SP.J.1016.2020.02019]

[43] TiFlash架构与原理. PingCAP. 2022. https://book.tidb.io/session1/chapter9/tiflash-architecture.html

[51] 缓冲区管理. PolarDB. 2022. https://apsaradb.github.io/PolarDB-for-PostgreSQL/zh/theory/buffer-management.html#lazy-checkpoint

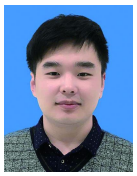
[72] 杨志丰. 分布式存储系统的一致性是什么? 2018. https://zhuanlan.zhihu.com/p/34656939

**胡梓锐**(2001－), 男, 博士生, 主要研究领域为HTAP 数据库基准评测, 数据库智能化.

**徐金凯**(1999－), 男, 硕士生, 主要研究领域为数据库基准评测.

**翁思扬**(2000－), 男, 博士生, 主要研究领域为数据库基准评测, 数据库负载仿真.

**张蓉**(1978－), 女, 博士, 教授, 博士生导师, CCF专业会员, 主要研究领域为分布式数据管理, 数据库基准评测, 数据流管理.



**王清帅**(1997－), 男, 博士生, 主要研究领域为面向应用的数据库负载仿真, 新型数据库基准评测.

**周烜**(1979－), 男, 博士, 教授, 博士生导师, CCF专业会员, 主要研究领域为数据库系统, 大数据处理技术.

**俞融**(1999－), 女, 硕士生, 主要研究领域为HTAP 数据库基准评测, 数据库系统.