计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330835
61(3): 540−554, 2024

OceanBase 分布式关系数据库架构与技术

阳振坤　杨传辉　韩富晟　王国平　杨志丰　成肖君
（北京奥星贝斯科技有限公司　北京　100102）
（zhengxiang@oceanbase.com）

Architecture and Technology of OceanBase Distributed Relational Database

Yang Zhenkun, Yang Chuanhui, Han Fusheng, Wang Guoping, Yang Zhifeng, and Cheng Xiaojun
（*Beijing OceanBase Technology Co., Ltd., Beijing* 100102）

Abstract　Relational database is the key information infrastructure of today's society. The Internet and digitization have brought high concurrency and massive data. Due to their centralized architectures, the processing power and storage capacity of traditional relational databases are stretched. OceanBase is a distributed relational database based on commodity PC servers. It achieves online horizontal scalability, automatic lossless disaster recovery from data center failure and high-ratio data compression. It has been used in finance, government affairs, telecommunication systems, Internet, etc. We introduce the architecture and some key technologies of OceanBase, including distributed transaction processing, LSM-tree-based storage system and distributed SQL optimizer. In addition, we explain in detail the high availability and data consistency of OceanBase, which can ensure that RPO is 0 and RTO is less than 8 seconds. At the same time, it also introduces OceanBase's multi-tenant mechanism, which adopts a native multi-tenant design within the cluster to implement multiple independent database services in the cluster. Based on the Sysbench and TPC-H evaluation benchmarks, comparative experimental results show that 1) in a stand-alone mode, the performance of OceanBase is 1.27 times to over 2 times that of MySQL; 2) in a single-master mode, the performance of OceanBase is 1.25 times to nearly 2 times that of MySQL; 3) in a multi-master mode, the performance of OceanBase is 1.09 to 3.1 times that of MySQL, and for complex OLAP queries, the performance of OceanBase is 6 to 327 times that of MySQL.

Key words　relational database; distributed transaction; LSM-tree-based storage; distributed SQL optimizer; multi-tenant

摘　要　关系数据库是当今社会的关键信息基础设施，互联网和数字化带来了高并发和海量数据，传统关系数据库均为集中式架构，处理能力和存储容量都捉襟见肘. OceanBase 分布式关系数据库基于通用 PC 服务器，不仅实现了在线水平伸缩，还实现了机房故障自动无损容灾以及高倍率数据压缩等，已经应用于金融、政务、通信和互联网等行业. 介绍了 OceanBase 分布式关系数据库的系统架构和关键技术，包括分布式事务处理、基于 LSM-tree 的存储系统以及分布式 SQL 优化器. 详细阐述了 OceanBase 数据库的高可用和数据一致性，包括 RPO 为 0 和 RTO 小于 8 s. 也介绍了 OceanBase 数据库多租户机制，即采用了集群内原生多租户设计，在集群内实现多个互相独立的数据库服务. 基于 Sysbench 和 TPC-H 评测基准，对比实验结果表明：1）在单机模式下，OceanBase 的性能是 MySQL 的 1.27 倍至 2 倍多；2）在单主模式下，OceanBase 的性能是 MySQL 的 1.25 倍至近 2 倍；3）在多主模式下，OceanBase 的性能是 MySQL 的 1.09 倍至 3.1 倍，对于 OLAP 的复杂查询，OceanBase 的性能是 MySQL 的 6 倍到 327 倍.

收稿日期：2023-10-19；**修回日期**：2023-12-14
通信作者：杨传辉（rizhao.ych@oceanbase.com）

关键词 关系数据库；分布式事务；基于 LSM-tree 存储；分布式 SQL 优化器；多租户

中图法分类号 TP311.13

1970 年 Edgar. F. Codd 博士发明关系模型[1]以及随后 SQL 语言的出现，关系数据库系统从 1980 年代起逐步发展成为了业务信息系统的基石. 由于业务信息系统的稳定性、可用性、实时性和高性能的要求以及事务的原子性(Atomicity)、数据的一致性(Consistency)、事务操作的隔离性(Isolation)、事务修改的持久性(Durability)(即 ACID 属性)，关系数据库的技术门槛非常高，当前全世界广泛使用的关系数据库系统都是集中式系统，可以离线、垂直扩展(scale-up)，难以在线、水平扩展 (scale-out)，处理能力和容量有限.

1990 年代，随着互联网的发展和普及，以电子商务为代表的各种线上业务产生了百倍、千倍于线下实体店(商场、酒店、工厂等)的并发访问量和数据量，远远超出了集中式关系数据库的处理能力和存储容量，对应用系统进行拆分并对数据库进行分库分表几乎成为了唯一可行的手段. 然而，这种削足适履的办法意味着应用系统的重构，低效率、低性能的跨库访问和跨库事务，外键、全局唯一约束、全局索引等只能被束之高阁，数据分析无法直接进行，数据管理的成本显著增加.

分布式关系数据库具备在线、敏捷的水平伸缩能力，能够很好地克服业务的高并发以及海量数据的挑战. 然而，与集中式关系数据库相比，分布式关系数据库更加复杂，在高可用、数据一致性、事务性能等方面面临更大的技术挑战.

2019 年，OceanBase 分布式关系数据库[2-3]通过了国际事务处理性能委员 (TPC) 的 TPC-C 联机事务处理基准测试[4]并打破 Oracle 保持了 9 年的性能记录. 2021 年，OceanBase 通过了 TPC-H 联机分析处理基准测试[5]并登顶性能榜首 (@30 000 GB). OceanBase 是迄今为止唯一获得了 TPC-C 和 TPC-H 性能榜首的数据库.

1 OceanBase 系统架构

作为一个分布式关系数据库，OceanBase 集群由若干可用区(Zone)组成，通常是 1～5 个 Zone，图 1 是一个 3-Zone 集群. 每个 Zone 包含 1 个或多个 OceanBase 节点，称为 OBServer. 每个 OBServer 由 SQL 引擎、事务引擎和存储引擎等构成. 每个 Zone 都包含了整个系统的全部数据，Zone 内每个 OBServer 管理其中

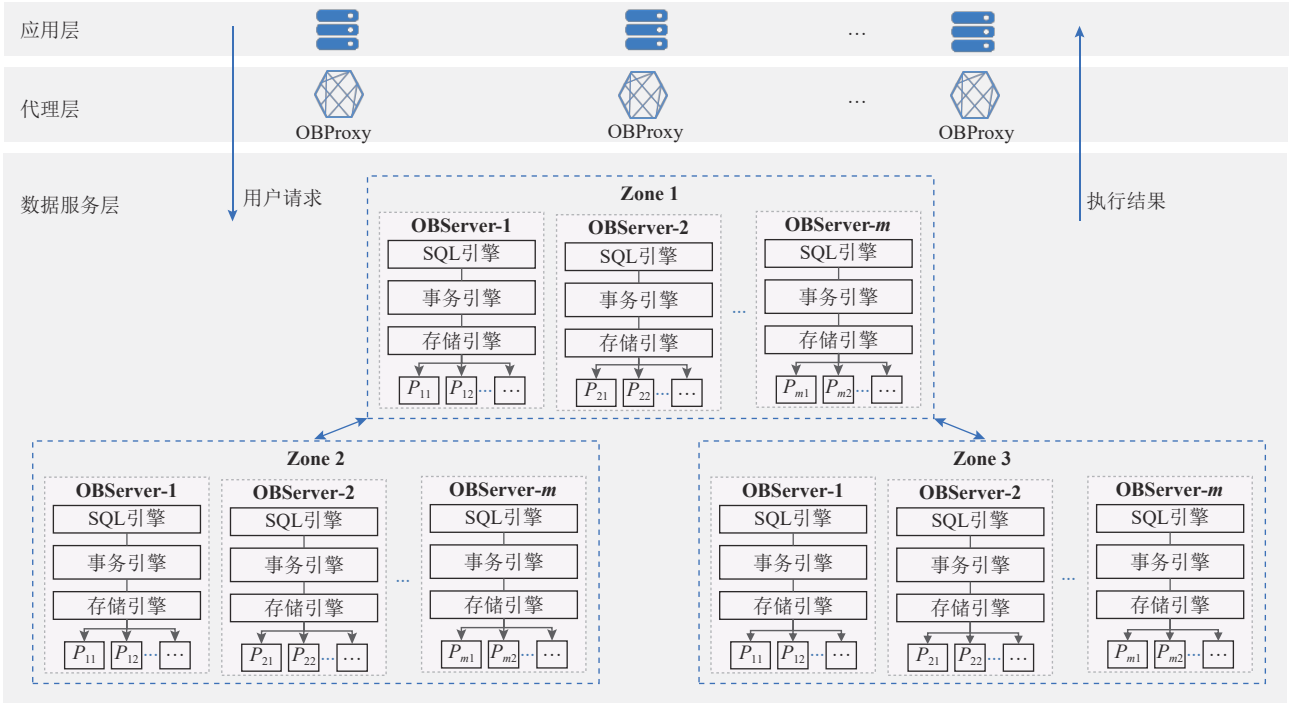


Fig. 1 The system architecture of OceanBase

图 1 OceanBase 系统架构

的部分数据即若干数据分片, 单个 Zone 内各个 OBServer 管理的数据彼此没有交集. 每个数据分片在每个 Zone 里都有一个副本, 这些副本构成该数据分片的 Paxos 组[6].

尽管应用系统可以通过与任何一个 OBServer 建立连接来访问 OceanBase 数据库, 但通常情况下应用系统通过 OBProxy(OceanBase Database Proxy)来访问 OceanBase 数据库, 以屏蔽 OBServer 节点乃至 Zone 的上下线、负载均衡等导致的数据在 OBServer 之间的迁移等.

2 高可用和数据一致性

高可用和数据一致性是业务系统对关系数据库的根本要求, 大部分业务都要求至少 99.99% 即 4 个 9 的可用率, 重要业务和关键业务则要求 99.999% 即 5 个 9 的可用率. 传统集中式关系数据库采用了高可靠的服务器和存储以及基于主备镜像的主库+备库的高可用解决方案, 但这个方案有 2 个不足:

1)成本高昂. 高可靠服务器和高可靠存储的价格十分昂贵, 显著地增加了成本.

2)主副本(主库)故障后备副本(备库)数据有损. 若备副本与主副本完全同步, 则主副本的每一笔事务必须到达备副本并持久化后才能认为成功, 一旦备副本故障或者主副本与备副本之间的网络异常, 那么主副本的写入将被阻塞, 从而导致服务中断, 这对于大部分业务是无法接受的. 因此尽管主副本与备副本之间号称主备镜像, 但实际上备副本数据通常略微落后于主副本, 一旦主副本故障, 则备副本的数据并不完整即修复点目标 (recovery point object, RPO) 大于 0, 这使得业务恢复面临较大的挑战, 特别是在金融等数据一致性要求较高的业务场景.

分布式关系数据库由多个节点组成, 整个系统出现节点故障的概率随着节点数的增加而快速增加, 如表 1 所示.

Table 1 Availability Rate of Distributed System

表 1 分布式系统的可用率 %

单个节点的可用率	节点数	
	10	100
99.999	99.99	99.90
99.99	99.90	99.01

因此, 即使每个节点的可用率都是 5 个 9 (99.999%), 当节点数达到 10 时整个系统的可用率只有 99.99%, 而当节点数达到 100 时整个系统的可用率只有 99.90%, 这已经无法满足大部分业务的需求, 而普通 PC 服务器的可用率显著低于 5 个 9.

为了解决分布式关系数据库高可用和数据一致性问题, 2013 年 OceanBase 引入 Paxos 分布式一致性协议[6].

如图 2 所示, 数据库每个分区(未分区的表被当作单分区表, 下同)由 1 个主副本和 2 个备副本组成, 主副本执行了 1～5 共计 5 个事务, 每个事务都在提交前把事务日志同步给备副本 1 和备副本 2, 其中备副本 1 收到了事务 1、3 和 4 的日志, 备副本 2 收到了事务 1、3、4 和 5 的日志, 由于事务 2 没有在超过半数参与者上持久化成功, 因此事务 2 被回滚, 其余 4 个事务成功提交.

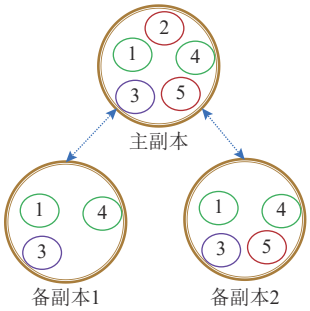


Fig. 2 Three-replica distributed architecture of OceanBase

图 2 OceanBase 的 3 副本分布式架构

Paxos 分布式一致性协议的引入带来的显著好处:

1) 主副本故障后系统可在短时间内自动选举出新的主副本, 新的主副本通过与活着的备副本握手快速补齐自身可能缺失的事务数据, 然后继续提供服务.

2) 系统无需高可靠的服务器和存储等硬件. 即使是普通的 PC 服务器, 2 台服务器同时出故障的概率也非常低. 对可用性要求特别高的业务, 可以采用 5 副本即 1 个主副本和 4 个备副本, 这样即使 2 个节点同时故障, 数据库服务也可以在短时间内无损、自动地恢复.

基于 Paxos 分布式一致性协议, 从 0.5 版本开始, OceanBase 实现了少数副本, 例如 3 副本(图 3a)时单个副本故障、5 副本(图 3b)时 2 个副本故障后, RPO 为 0, 且 RTO(recovery time object)小于 30 s, 从 4.0 开始, RTO 降为了 8 s.

3 分布式事务

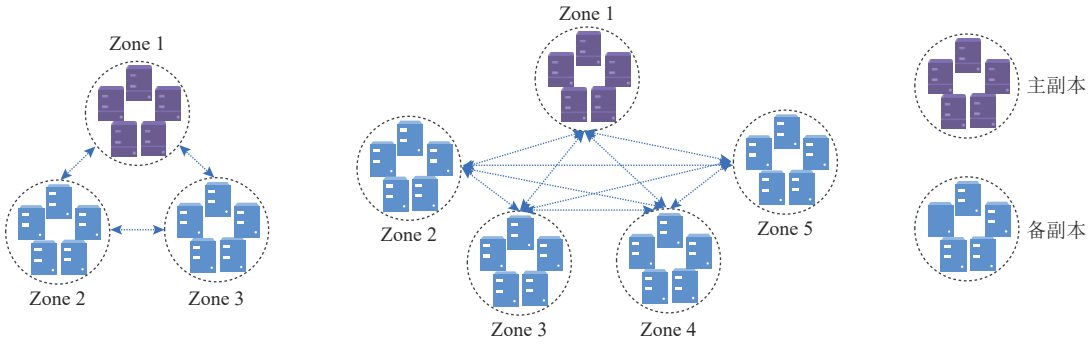


Fig. 3 Three replicas and five replicas in OceanBase

图 3 OceanBase 的 3 副本和 5 副本

OceanBase 的数据以分区为单位进行管理，任何时刻每个分区都属于且只属于一个日志流，日志流用于记录所包含的分区的数据操纵语言(data manipulation language, DML)对应的 WAL(write ahead log) 日志，OceanBase 通过在日志流中记录事务操作的日志完成对事务的提交.

数据库的事务功能要保证 ACID 四个基本特性[7]，在数据库系统内部有 2 个重要的模块来实现这 4 种特性，即故障恢复模块和并发控制模块.

故障恢复模块通过日志系统和事务提交协议实现原子性和持久性这 2 个特性. 日志系统保证了事务的修改日志的持久化，并且在出现故障等异常时也能将日志恢复出来，进而恢复事务的全部操作，也就保证了事务的持久性. 事务提交协议通过日志持久化的原子性，以实现事务的原子性.

并发控制模块用于协调多个并行执行的事务的操作顺序，通过识别并控制有冲突的多个修改操作和读取操作，保证事务执行的隔离性语义和数据的一致性.

接下来首先介绍日志系统，然后介绍事务提交协议，之后将并发控制内容分成写写操作和读写操作两部分进行介绍.

3.1 Paxos 一致性协议

OceanBase 将事务日志存入多台机器的多个副本中，比如 3 副本或 5 副本. 如果少数副本出现故障(如机器掉电、硬盘损坏等)，日志可以从活着的多数副本中恢复出来，保证数据不会丢失. Paxos 一致性协议[6]用于在各个副本之间同步事务产生的日志，实现了少数派副本故障时自动容灾切换和故障恢复.

一致性协议有选举和日志同步 2 个主要的部分.

1)选举. OceanBase 的选举协议针对无主选举和有主改选 2 种场景分别处理. 当系统刚启动时或者主副本节点故障时，各个选举的参与者发现没有主副本后会进行无主选举，OceanBase 无主选举协议会在最短的时间内选举出优先级最高的副本作为新的主副本节点，并开始提供服务. 有主改选发生在主副本节点认为自己不再能胜任主副本节点的工作或人工干预时，主副本节点会将作为主副本的角色转让给其他节点，同时自身降级为备副本节点.

2)日志同步. 正常工作时，主副本节点把数据库的事务日志同步给其他副本，所有副本按照一致性协议对该日志进行确认，当日志获得多数成员的确认后即完成了日志的写入. 以常见的 3 副本为例，主副本将日志同步给其他 2 个副本，当包括主副本在内的任意 2 个副本接受了日志并在本地硬盘中成功持久化，则日志就写入成功，出现任意副本的故障时，剩余的 2 个副本中至少会有一个副本包含刚写入的日志，保证日志不会丢失.

3.2 两阶段提交

在单个日志流中，一个事务的日志通过一致性协议获得多数派确认后事务的修改就被持久化地记录了下来，事务也就完成了原子提交. 对于一个分布式事务，事务的修改涉及多个日志流，日志要在对应的日志流中记录，此时，任何一个日志流的持久化成功都不能代表整个事务的提交，需要事务涉及的所有日志都持久化成功才能保证事务提交的完整性. OceanBase 通过“两阶段提交协议”保证分布式事务的原子性.

经典的两阶段提交协议中[8]，协调者与参与者都会执行多次持久化日志的操作，协调者记录事务的执行流程和最终提交状态，参与者记录局部事务的持久化状态，从使用者视角看到分布式事务的提交延迟需要至少 4 次日志提交的延迟之和. OceanBase 实现了优化的两阶段提交协议，显著缩短了事务提交的延迟.

为了让两阶段提交协议可以在更短的时间内完

成, OceanBase 的策略是不依赖协调者记录的事务的状态日志, 而是依赖参与者记录的操作日志, 在两阶段执行过程中如果出现了系统故障, 那么由参与者互相校验事务的日志是否记全了, 最终依然可以通过确认超过半数的参与者都记全了各自的操作日志, 来确认事务最终提交的状态. 如果不是超过半数的参与者都记全了日志, 那么事务就没有完成提交, 最终就是回滚状态. OceanBase 的两阶段提交协议将事务的提交延迟减小到只有一次日志同步, 显著提高了事务提交的效率.

如图 4 所示, 当事务开始提交时有 2 个参与者 P0 和 P1, OceanBase 选择某个参与者作为协调者, 该协调者只有内存状态没有持久化状态. 协调者首先会给所有的参与者发送事务的 Prepare 请求, 各个参与者收到此请求后, 会将事务中所做的修改提交到日志中并进行持久化. 待 Prepare 日志持久化成功后, 参与者发送 Prepare ok 消息给协调者. 当协调者收到所有参与者的 Prepare ok 消息后, 事务的所有修改均已提交到日志中并持久化成功, 无论出现何种异常事务均可以恢复, 所以此时事务事实上已经提交完成, 因此协调者应答客户端事务提交成功. 与此同时, 协调者发送 Commit 给各个参与者, 通知他们提交事务, 因为事务实际已经提交, 所以各参与者的提交动作可以异步完成.

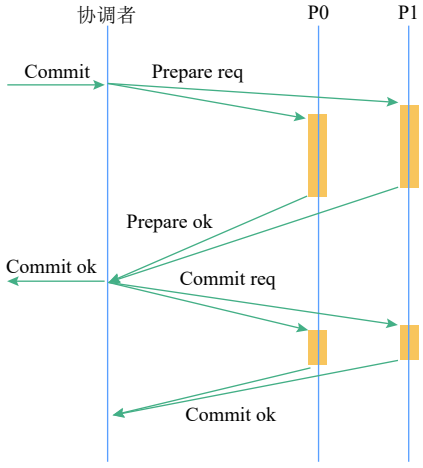


Fig. 4 Two-Phase Commit Protocol in OceanBase

图 4 OceanBase 两阶段提交协议

3.3 写写之间的并发控制

2 个同时执行的事务如果修改同一条数据, 后执行修改操作的事务需要等待先执行修改操作的事务完成后才能执行, 这样才能保证对于数据修改的正确语义, 也才能保证数据的一致性[7,9].

在 OceanBase 中, 事务修改的每一行数据都记录事务的标识, 此标识也用来标记该行数据是否正在被事务修改. 当一个事务执行修改操作时, 首先需要判断被修改的行是否有其他事务的修改标识, 如果有则判断对应的事务当前的状态, 如果事务是正在执行中的状态, 那么当前行依然被活跃事务修改, 后续的事务必须等待之前的事务执行结束, 如果事务已经处于提交或者回滚状态, 那么当前行则可以被后续的事务继续修改.

3.4 读写之间的并发控制

OceanBase 采用多版本并发控制[10]的方法来处理并发的事务读取操作与修改操作. 在 OceanBase 中, 事务修改的数据都不是原地更新, 而是生成新的版本, 对于同一行数据, 每次修改后的数据版本都存在. 数据的版本号是全局的, 在事务提交时确定事务的提交版本号, 作为此事务修改的所有数据的版本.

在 OceanBase 里, 提供事务版本号的服务叫全局时间戳服务(global timestamp service, GTS), 此时间戳服务每秒可以处理几百万次请求, OceanBase 采用了聚合方式, 每台服务器同一时间并发的事务可以聚合起来取时间戳, 大大降低对于时间戳服务的请求. 在 2020 年OceanBase 的第2 次TPC-C 测试中,OceanBase 每分钟要执行 15 亿个事务, 时间戳服务完全可以支持.

读取操作使用的版本号有 2 种获取方法, 对于可串行化隔离级别(Serializable)的事务, 在事务开始时从全局时间戳服务获取 1 次, 作为整个事务所有读取操作使用的版本号. 对于读已提交(Read Committed)隔离级别的事务, 在每条 SQL 语句开始执行时从全局时间戳获取一次, 作为本语句的读取版本号.

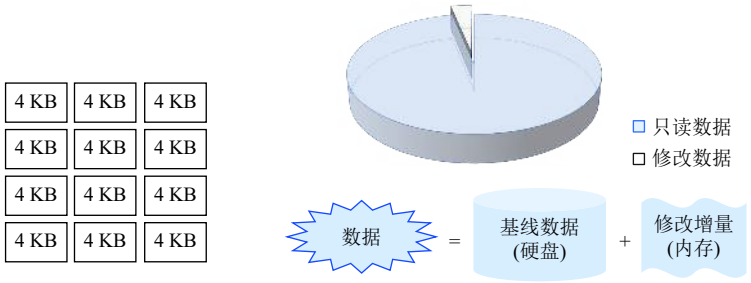
读取操作执行的过程中, 对于每一行需要读取的数据会选择小于等于读版本号的最大的数据版本作为读取的版本. 这样的操作能够实现所有的读取操作读取到的是系统的一个快照, 因为对于一个事务来说其修改的数据使用的是同一个提交版本号, 所以读取操作要么全部读取到, 要么全都不读取, 不会出现读取一部分的情况.

4 LSM-tree 存储系统

关系数据库需要保证事务的持久性, 为了抵御突发的电力等故障, 数据需要保存到硬盘(机械磁盘或固态盘)等非易失性存储介质即持久化, 事务的日志需在事务提交前持久化. 由于各种硬盘都是块设备即按一定尺寸大小的块对齐进行读写, 比如 4 KB

等，通常关系数据库都采用了定长块（4 KB，8 KB 等）原地更新的方式，如图 5(a) 所示.

OceanBase 选择了基于 LSM-tree[11] 的存储架构，如图 5(b) 所示. 跟传统的定长块原地更新的存储架构相比，LSM-tree 存储有显著的不同，即修改数据（修改增量）记在内存、基线数据只读等.



(a) 定长块原地更新　　(b) OceanBase LSM-tree存储

Fig. 5　Storage engine of OceanBase

图 5　OceanBase 存储引擎

LSM-tree 存储有显著的优势：

1）写入性能好，且存储设备无需随机写能力.

2）周期地（每天）进行修改增量与基线数据合并，由于这样通常避开业务高峰，因此系统的 CPU、I/O 和网络资源等有一定的空闲，故可对数据进行一些较为消耗 CPU、I/O 和/或网络等的处理，比如数据压缩、数据统计等，且不对系统性能产生负面影响. 这使得 OceanBase 通常有 3 倍或更高的数据压缩率，降低用户存储成本，却不会降低业务吞吐量或增加业务延时. 表 2 是部分用户业务从 MySQL/Oracle 迁移到 OceanBase 时迁移前后数据量的对比.

Table 2　Some User Business Data Compression Rates of OceanBase

表 2　部分用户业务的 OceanBase 数据压缩率

原数据库	原数据量	OceanBase 数据量	压缩倍率
MySQL	1.5 TB	500 GB	3.00
MySQL	2.7 TB	900 GB	3.00
MySQL	3.6 TB	1.2 TB	3.00
Oracle	1.8 TB	500 GB	3.60
Oracle	7.8 TB	800 GB	9.75
Oracle	12 TB	3.5 TB	3.43
Oracle	20 TB	2 TB	10.00
Oracle	30 TB	10 TB	3.00

3）基线数据只读，这简化了数据存储的数据结构和数据缓存的实现.

4）既适合行存又适合列存，解决了基于定长块原地更新的数据库的列存数据难以实时更新的难题，使得数据库既能进行交易处理，又能进行分析处理.

当然，LSM-tree 存储也有一些劣势：

1）读需要融合修改增量和基线数据.

2）数据写入较多时，修改增量需要转储到硬盘以释放内存.

3）需要周期地（每天）进行修改增量与基线数据的合并.

4）SQL 执行的代价模型既需要考虑基线数据又需要考虑修改增量.

针对这些劣势，OceanBase 在设计和实现中进行了相应的处理和优化，减少了这些劣势带来的性能和功能等方面的影响.

5　分布式 SQL 优化器

查询优化器是关系数据库系统的核心组件之一，是衡量关系数据库系统成熟度的“试金石”. 查询优化理论诞生已有四十多年，学术界和工业界已经形成了一些比较完善的查询优化框架，其中包括 IBM System-R[12] 的优化框架和 Volcano[13]/Cascades[14] 的 Top-down 优化框架. 然而，围绕查询优化的核心难题始终没有改变，即如何利用有限的系统资源来选择一个“好”的执行计划. 作为一个分布式关系数据库系统，OceanBase 需要克服分布式查询优化的挑战.

OceanBase 查询优化器的设计参考了 System-R 的设计，图 6 展示了 OceanBase 查询优化器的总体框架. OceanBase 查询优化器主要由计划优化模块和计划管理模块组成，其中计划优化模块负责生成最优的执行计划，计划管理模块负责缓存和演进执行计划. 接下来，依次介绍 OceanBase 的查询改写机制、分布式计划生成机制、计划缓存机制和计划演进机制.

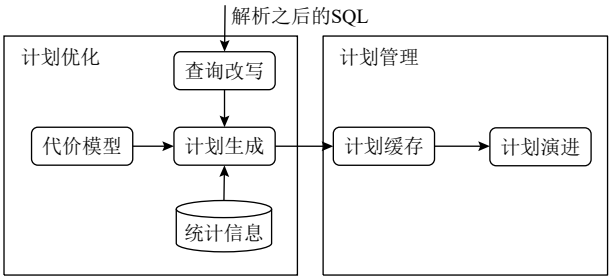


Fig. 6 Query optimizer framework of OceanBase

图 6 OceanBase 查询优化器框架

5.1 查询改写

查询改写是指利用查询改写规则对查询语句进行重写或转换，以便更好地利用数据库的索引、优化技术和查询处理算法来提升查询性能. OceanBase 设计并实现了大量的查询改写规则，涵盖了不同的关系代数之间的相互转换. 这些策略一部分来自于已有的文献，如谓词移动[15]、子查询提升[16]、分组上拉和分组下压[17]等；然而，当前已有的研究并没有也无法穷尽所有的改写策略，很多已有的改写策略仍有提升的空间，并且还有许多值得探索的新策略. OceanBase 在服务大量的客户过程中，对业务 SQL 使用模式进行抽象并提出了许多新的改写策略.

例如，已有的数据库系统实现了 OR 展开的技术来优化以下查询：

```
SELECT * FROM T1, T2 WHERE T1.C1 = T2.C1 OR T1.C2 = T2.C2;
=>
SELECT * FROM T1, T2 WHERE T1.C1 = T2.C1
UNION ALL
SELECT * FROM T1, T2 WHERE LNNVL(T1.C1 = T2.C1) AND T1.C2 = T2.C2;
```

原始查询中，T1 和 T2 表之间只有一个 OR 条件的连接谓词，只能进行笛卡尔积. 经过 OR 展开后，T1 和 T2 表之间是两次等值连接，可以应用更多的连接算法. 但是，该策略仅适用于内连接，不适用于其他连接类型. 为此，OceanBase 设计实现了针对不同连接类型的 OR 展开策略，能够将外连接、半连接、反连接上的 OR 连接谓词展开为等值连接.

当前，OceanBase 查询优化器已经积累了多种针对不同关系代数式的改写策略，能够满足复杂查询的多个方面的改写需求. 除了一些常见的简单改写策略之外，其中有少部分复杂策略是基于已有文献实现的，更多复杂策略则是基于业务场景的抽象得到的. 在整个模块的发展过程中，查询优化器还积累了许多基础改写算法，例如判定 2 个关系代数的包含关系和判定谓词的空值拒绝性质等. 这些基础算法在不同的场景中被不同的改写策略复用，使得 OceanBase 能够快速实现多种新型的改写策略.

5.2 计划生成

计划生成主要负责枚举所有等价的执行计划，并根据代价模型选择代价最小的执行计划. OceanBase 使用自底向上的动态规划算法来进行计划枚举[12]. 首先，它会枚举基表路径，然后再枚举连接顺序和连接算法，最后按照 SQL 语义的顺序枚举其他算子. 与单机关系数据库系统相比，OceanBase 的分布式架构在计划枚举中引入了许多额外的复杂因素，如分区信息、并行度、分区裁剪和分布式算法等. 这些因素从根本上增加了分布式计划枚举的复杂性，并显著扩大了其计划枚举空间.

为了应对分布式计划枚举的复杂性，业界大多数系统[18]采用了二阶段的计划枚举方法. 第一阶段假设所有表都是单机的，并依赖现有的单机计划枚举能力选择一个本地最优的执行计划. 第二阶段将第一阶段的单机执行计划通过适当的规则转换成一个分布式执行计划. 这种二阶段的分布式计划枚举方法通过枚举部分计划空间来降低分布式计划枚举的复杂性，但会导致计划次优的问题. 核心问题在于第一阶段优化时没有考虑分区信息，因此可能会选择次优的连接顺序和单机算法.

为了解决二阶段计划次优的问题，OceanBase 采用了一阶段的计划枚举方法. 计划枚举的其中一个目标是为执行计划中的每个算子选择一种具体的实现方法. 在单机的场景下，算子的实现方法只需要考虑单机的实现，但在分布式的场景中，算子的实现方法除了要考虑单机实现之外，还需要考虑其分布式的实现. 以数据库中的连接算子为例，常见的单机算法有 hash join、merge join 和 nested loop join 等，常见的分布式方法有 partition-wise-join, hash-hash distribution join 和 broadcast distribution join 等. OceanBase 一阶段计划枚举的核心思想是同时枚举算子的本地算法和分布式算法，并使用分布式代价模型来计算代价，而不是通过二阶段的方式分阶段地枚举算子的本地算法和分布式算法. 通过这种方式，OceanBase 可以从根本上解决二阶段计划枚举导致的分布式计划次优的问题.

相较于二阶段的计划枚举方法，OceanBase 一阶段的计划枚举方法能够完整枚举出所有的分布式计划空间，但也会引入计划枚举效率问题，尤其是在复杂查询的场景下. 为了解决这一问题，OceanBase 发

明了大量快速裁剪计划的方法，并新增了新的连接枚举算法来支持超大规模表的计划枚举，从而根本性地提升了分布式计划枚举的性能. 我们的实验结果也表明，在 OceanBase 中，可以在秒级内完成对 50 张表的分布式计划枚举.

5.3 计划缓存

查询优化是数据库中非常耗时的一个过程. 随着查询复杂度的增加，相应的计划数量也会呈指数级增长，查询优化本身也变得异常复杂. 因此，对于执行速度本身很快的查询来说，查询优化带来的开销也变得不可忽视. 为了降低查询优化的性能开销，与其他关系数据库一样，OceanBase 对执行计划进行了缓存.

与 Oracle 的 RAC 架构不同，OceanBase 的 shared-nothing 分布式架构给计划缓存带来了额外的挑战. 由于 OceanBase 中特定的分布式算法对分区的位置信息有强烈的依赖，有时可能无法使用已缓存的执行计划来执行相同的查询. 例如，在 OceanBase 中，如果缓存的执行计划包含了 *A* 表和 *B* 表进行 partition-wise-join 的算子，而 partition-wise-join 要求 *A* 表和 *B* 表对应的分区位于同一台机器上，如果 *A* 表或 *B* 表发生了分区迁移，它们可能就无法执行 partition-wise-join 操作，这时缓存的执行计划可能就变得无效，需要重新生成计划.

为了解决这个问题，OceanBase 会为每一条 SQL 缓存多个执行计划. 在计划缓存阶段，OceanBase 会遍历执行计划，并从依赖位置信息的算子中提取位置信息约束，然后将该执行计划与约束一起缓存到计划缓存器中. 在计划匹配阶段，如果存在多个缓存计划，OceanBase 会按照平均执行时间从小到大的顺序进行匹配，找到第一个满足分区位置约束的计划进行执行. 如果没有满足约束的计划，就重新生成计划并将其加入到计划缓存器中. 通过这种方式，OceanBase 成功解决了分布式计划的缓存问题，并提升了分布式查询的性能.

5.4 计划演进

在关系数据库系统中，因为数据增删导致统计信息的变化、数据库的升级以及 Schema 的变更等，执行计划发生变化是一个比较常见的现象. 为了避免由于执行计划变化而导致的性能回退问题，Oracle 12c 引入了离线计划演进[19]的方案. 该方案的核心思想是始终执行经过验证的基线计划，而不执行未经验证的新计划. 只有在新计划经过离线演进并通过验证后，才可以使用它. 然而，离线计划演进的问题在于，在用户未进行干预之前，不能立即使用性能更好的新计划. 与 Oracle 的离线计划演进方案不同，OceanBase 提出了一种基于用户实时流量的在线计划演进方法，该方法的主要优势是能够快速接受性能更好的新计划.

图 7 展示了 OceanBase 的在线计划演进方法. 在优化器生成新的计划时，该方法首先检查是否存在历史基线计划. 如果不存在，则直接执行新计划并将其添加到基线计划集合中；否则，选择基线计划和当前新计划进行在线演进. 如果演进成功，就用新计划替换旧的基线计划，否则直接舍弃新计划.

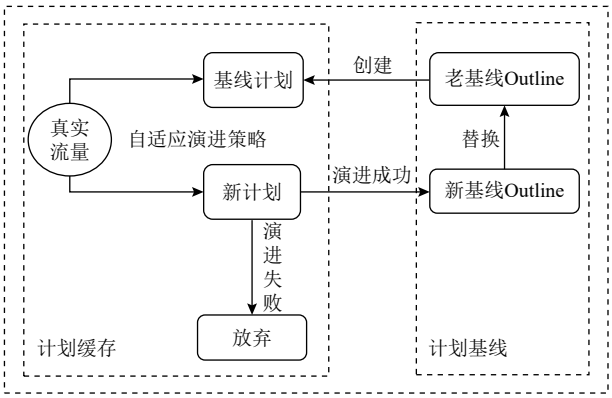


Fig. 7 Online plan evolution method of OceanBase

图 7 OceanBase 在线计划演进方法

OceanBase 使用内部表存储历史的基线计划集合. 为了完整地复现基线计划，OceanBase 在内部表中存储了原始的 SQL 和用于复现该计划的完整 Outline 数据. Outline 数据是 OceanBase 中一组可用于固定执行计划的 Hint 组合. 同时，为了降低在线演进可能导致的潜在性能问题，OceanBase 采用了自适应的演进策略，即根据当前新计划和基线计划的执行时间，动态地调整新计划和基线计划的流量比例. 如果新计划的执行性能优于基线计划，则将更多流量分配给新计划；否则，将更多流量分配给基线计划.

与其他关系数据库相比，OceanBase 的在线计划演进方法能够快速接受性能优越的新计划，避免了性能较差的新计划对数据库稳定性的影响，减少了数据库的运维工作.

6 多租户

OceanBase 数据库采用了集群内原生多租户设计，不依赖于外部容器等实现了一个集群内的多个互相独立的数据库服务. 通过实现原生多租户，可以实现 3 个功能：

1)实现资源隔离,使得每个数据库服务的实例不感知其他实例的存在,并通过底层控制确保租户数据的安全性;

2)简化了集群的部署和运维,能够原生提供安全、灵活的 DBaaS (Database as a service) 服务;

3)实现一个集群对原来多个数据库实例的资源整合,降低硬件资源总体成本.

6.1 租户的概念

租户(tenant)是一个逻辑概念. 在 OceanBase 数据库中,租户是资源分配的单位,是数据库对象管理和资源管理的基础,对于系统运维,尤其是对于云数据库的运维有着重要的影响. 租户在一定程度上相当于传统数据库的“实例”概念. 租户之间是隔离的,类似于操作系统的虚拟机或容器. 在数据安全方面,OceanBase 数据库不允许跨租户的数据访问,以确保用户的数据资产没有被其他租户窃取的风险. 在资源使用方面,OceanBase 数据库表现为租户独占其资源配额. 总体上来说,租户既是各类数据库对象的容器,又是资源(CPU、内存、存储 I/O 等)的容器.

租户有系统租户和用户租户 2 种类型. 系统租户是集群默认创建的租户,与集群的生命周期一致,负责管理集群和所有租户的生命周期. 因为系统租户只用来管理其他租户,并存储集群的元数据,为了设计简单,系统租户仅支持单点写入,不具备扩展能力.

用户租户是为用户创建的租户,对外提供完整的数据库功能,支持 MySQL 和 Oracle 两种兼容模式. 用户租户支持服务能力水平扩展到多台机器上,支持动态扩容和缩容,内部会根据用户的配置自动创建和删除日志流.

新建一个租户的时候,管理员需要定义其资源单元配置(UNIT_CONFIG)、所在 Zone 列表(ZONE_LIST)、每个 Zone 中资源单元数目(UNIT_NUM)这 3 个因子. 其中,资源单元配置包括 CPU、内存、存储 IOPS、存储空间限制等物理资源的规格限制. 系统允许通过在线调整租户的资源单元配置,实现资源的垂直扩容缩容;通过在线调整资源单元数目,实现租户资源的水平扩容缩容;通过增删 Zone 列表,实现租户数据和服务副本数的增删,从而调整单个租户的容灾等级,也可以用来实现租户数据的跨数据中心迁移.

6.2 资源隔离

OceanBase 数据库中把资源单元(UNIT)当作给租户分配资源的基本单位,一个资源单元可以类比于一个包含 CPU、内存、存储资源的容器. 一个节点上可以创建多个资源单元,在节点上每创建一个资源单元都会占用一部分该节点的相应资源.

一个租户可以在多个节点上放置多个资源单元,但一个租户在单个节点上只能有一个资源单元. 一个租户的多个资源单元相互独立,每个节点控制本地多个资源单元间的资源分配. 类似的技术是 Docker 容器和虚拟机,但 OceanBase 并没有依赖 Docker 或虚拟机技术,而是在数据库内部实了资源隔离. 在数据库内原生实现资源隔离有 4 个优势:

1)资源共享,降低单个租户成本;

2)支持轻量级租户,如资源单元可以支持 0.5c CPU 规格;

3)实现快速的规格变化和快速迁移;

4)便于实现租户内更细粒度的资源隔离和优先级管理.

通过实现资源隔离,用户租户之间可以实现:

1)内存隔离;

2)CPU 在隔离的同时,允许共享;

3)存储 IOPS 在隔离的同时,允许共享.

6.2.1 CPU 的隔离

为了控制 CPU 分配,每个资源单元可以指定 MIN_CPU, MAX_CPU 这 2 个参数. MIN_CPU 决定了在一个调度周期内最少给该租户保留的时间片,MAX_CPU 决定了最大分配的时间片. 当节点上实际使用的 CPU 低于所有资源单元的 MAX_CPU 总要求时,所有资源要求都得到满足;否则,超出部分按 MAX_CPU 与 MIN_CPU 之间的差值决定的权重共享. 实现时,节点内通过控制活跃线程数来控制租户 CPU 的占用,但由于 SQL 执行过程中可能会有 I/O 等待、锁等待等,所以 1 个线程无法用满 1 个 CPU,节点会按需给每个虚拟 CPU 启动多个线程.

在典型的工作负载中,大量高并发的短事务,往往和少量低频但资源消耗较多的“大查询”混合在一起. 相比于大查询,让短查询尽快返回对用户更有意义,即大查询的查询优先级更低,当大查询和短查询同时争抢 CPU 时,系统会限制大查询的 CPU 使用,即用于大查询的线程被限制在一定的比例. 一方面,当一个线程执行的 SQL 查询耗时超过阈值,这条查询就会被判定为大查询,一旦判定为大查询,如果大查询使用的线程比例达到上限,则该大查询的线程会进入等待状态,为其它工作线程让出 CPU. 当一个工作线程因为执行大查询被挂起时,作为补偿,系统按需新建工作线程. 另一方面,系统会根据执行历史,在开始执行前就判定一个查询为“大查询”.

6.2.2 IOPS 的隔离

除了存储容量，数据库硬盘（包括本地硬盘、云盘等）的 IOPS 也是影响性能的关键因素. 有时用户希望不同租户的硬盘带宽可以完全隔离，就像他们运行在不同的物理硬盘上一样；有时用户则希望系统提供一定的灵活能力，租户间实现闲时共享、忙时隔离. OceanBase 使用 iops.{min, max, weight}三元组描述一个租户的硬盘（云盘等）带宽规格，iops.min 描述硬盘带宽的下限，这部分资源为租户预留资源；iops.weight 描述租户的资源权重，如果一个租户实际使用的硬盘带宽超过了 iops.min，那他通过 iops.weight 与其他租户划分资源；iops.max 描述硬盘带宽的上限，租户使用的硬盘带宽不允许超过该上限.

如果租户要求硬盘带宽完全隔离，则设置 iops.max = iops.min，该租户的硬盘带宽按照 iops.min 的值刚性保留. 如果租户希望自己的硬盘带宽除了保底能力外，还有一定的弹性上浮空间，可以配置 iops.max > iops.min，并根据业务自身的权重配置合理的 iops.weight，从而按需使用共享的硬盘带宽. 作为关系数据库，iops.min 预留硬盘资源的能力对在线事务处理提供稳定的时延是很重要的. 而对于 OLAP 型业务负载，则可以把 iops.min 配置为 0，完全共享硬盘带宽，以最大化利用资源.

6.3 资源单元的均衡

系统需要对资源单元进行管理，并通过把资源单元在多个节点间调度，对系统资源进行有效利用.

1）资源单元的分配，即新建一个资源单元时，系统需要决定其分配到哪个节点上.

2）资源单元的均衡，即在系统运行过程中，系统根据资源单元的资源规格等信息对资源单元进行再平衡的一个调度过程.

单一类型资源的多机负载均衡比较容易实现，只需要计算单一资源的利用率，并通过资源单元的迁移达到每个节点上的资源均衡. 当系统中有多种资源需要进行分配和均衡时，仅使用其中一种资源的占用率去进行分配和均衡很难达到较好的分配和均衡效果，为此，在多种资源（CPU 、内存）均衡和分配时，为每种资源分配一个权重，作为计算节点总的资源占用率时该资源所占的百分比. 而每种资源的权重，依赖于集群中该资源的总体使用率，即某种资源使用的越多，则该资源的权重就越高.

6.4 云服务

2022 年，国内公有云数据库已经占据整个数据库市场一半以上的份额[20]. 作为一款分布式关系数据库，水平伸缩能力使得 OceanBase 十分契合公有云服务.

OceanBase 云服务有 2 种类型：租户级实例服务和集群级实例服务. 使用租户级实例服务时，使用者不感知 OceanBase 集群资源，只要按需对租户进行扩容和缩容等操作，以满足业务发展的需求并节省成本. 使用集群级实例服务时，使用者不仅可以按需对集群进行扩容和缩容，还可以在集群内创建和管理租户，对这些租户按需进行扩容和缩容等，通过充分发挥租户级资源共享的能力，以最大化利用集群资源，满足业务的需求并降低成本.

OceanBase 已经在多个国际主流云平台上提供 OceanBase Cloud 云服务，OceanBase 的 MySQL 和 Oracle 兼容性大大降低了业务迁移到 OceanBase Cloud 的风险和成本，目前已有大量用户使用 OceanBase Cloud 云服务，并且在快速增长中.

7 性能测试和对比

与 OceanBase 类似，MySQL MGR 使用 Paxos 协议实现分布式一致性（Oracle 等尚无类似的技术方案）. 本文使用比较常见的 Sysbench、TPC-H 测试工具对 OceanBase 和 MySQL 的 OLTP 和 OLAP 性能进行了对比. MySQL 版本是 8.0.31 Community Server，OceanBase 版本是 4.2.1，测试使用的数据库服务器是阿里云 r7.8xlarge ECS，硬件配置是 32 vCPU(Intel Xeon Platinum 8369 B CPU @ 2.70 GHz) 、256 GB 内存，日志盘和数据盘使用阿里云云盘，云盘的最大性能吞吐量可以达到 350 MB/s.

测试工具的版本是 Sysbench 1.1. Sysbench OLTP 测试一共创建 30 张表，单张表有 100 万行的数据量，压测客户端部署在同机房一台独立的服务器上，压测时长为 5 min，在压测时会调整客户端的压测并发数，让 MySQL 和 OceanBase 跑出各自最好的性能. Sysbench OLTP 读写混合场景是个读多写少的多表事务，一个事务包括 18 条读写 SQL，包括点查询、范围查询、求和、排序、去重、增删、更新等，OLTP 只读场景只运行其中的 14 条读 SQL，OLTP 只写场景只运行其中的 4 条写 SQL，性能指标 TPS 指的是每秒处理的事务数.

7.1 单机模式

MySQL 和 OceanBase 都提供了单机部署的形态，这种数据库部署形态下的高可用完全依赖硬件的可靠性，数据库软件本身没有高可用能力.

在单机模式下，所有表的数据集中在一台服务

器上. 图 8 表明, OceanBase 在各种场景下的性能都优于 MySQL; 在只写事务的场景下, OceanBase 的性能是 MySQL 的 2 倍多; 在其他 2 个场景下, OceanBase 的性能是 MySQL 的 1.27 倍.

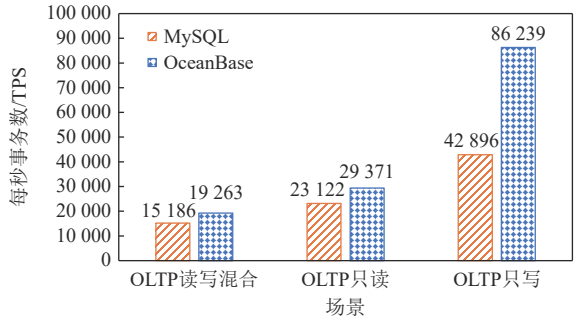


Fig. 8 Performance comparison between OceanBase and MySQL in stand-alone mode

图 8 单机模式下 OceanBase 和 MySQL 的性能对比

7.2 单主模式

与 OceanBase 类似, MySQL MGR 是基于 Paxos 协议的分布式数据库集群, 事务提交必须经过半数以上的节点同意方可提交. 本文采用比较常用的三节点的部署进行分布式数据库集群的性能对比. 单主模式表示一个节点为主、另外两个节点为备, 备副本节点从主副本节点同步事务日志, 只有主副本节点能够提供读写访问, 当主副本节点故障时, 其中 1 个备副本节点会被自动选举为新主, 兼顾数据一致性和可用性.

下述性能测试采用了 3 台 ECS.r7.8xlarge 服务器, sysbench 测试的参数跟单机模式相同, 为了避免对数据库性能的影响, 客户端和 OceanBase 的代理服务部署在独立的机器上, OceanBase 代理服务在分布式部署模式下负责路由请求到数据所在的数据库服务器的组件. 在分布式集群单主模式的部署下, 所有表格的主副本节点会集中在一台服务器上, 类似单机模式, sysbench 测试运行的事务都是单机多表事务, 写事务的日志需要同步到超过半数(2 台)的机器, 预期写性能比单机部署会有所下降, 读性能影响不大.

OceanBase 的 3 副本单主部署下, 读写混合、只读和只写 3 个场景的性能较单机部署分别下降 8%、3%、20%; MySQL 的 3 副本单主部署下, 读写混合、只读和只写三个场景的性能较单机部署分别下降 12%、1%、19%. 在 OLTP 只写场景下, OceanBase 的性能是 MySQL 的近 2 倍, 读写混合和只读场景 OceanBase 的性能分别是 MySQL 的 1.34 倍和 1.25 倍, 如图 9 所示.

7.3 多主模式

分布式集群的多主模式指任何一个数据库节点都可以写, 即 3 个节点都是主副本节点, 每条用户请求访问的数据可能分布在不同的主副本节点上, 3 个数据库节点是完全对等的.

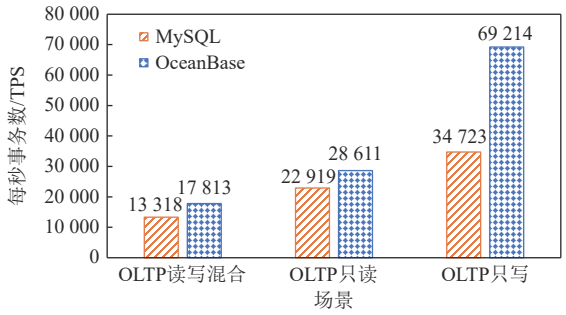


Fig. 9 Performance comparison between OceanBase and MySQL in three-replica single-master mode

图 9 3 副本单主模式 OceanBase 和 MySQL 的性能对比

性能测试采用的数据库服务器是阿里云 3 台 ECS.r7.8xlarge 型号的 ECS 服务器, Sysbench 测试使用的数据量和表数量跟单机模式、分布式集群单主模式相同. 为了避免对数据库性能的影响, 客户端和 OceanBase 的代理服务部署在独立的机器上, 并且确保压力机不会是性能的瓶颈. 在多主模式的部署下, Sysbench 30 张表格的主副本节点会随机分布在 3 台 ECS 服务器上. Sysbench OLTP 是个多表的测试场景, 对应的数据库事务基本上都是需要从 3 台机器上读写数据的分布式事务, 3 台数据库服务器的资源开销是相当的.

OceanBase 多主模式相对于单主模式, OLTP 读写混合场景的 TPS 提升了 86%, 只读事务场景的 TPS 提升了 156%, 只写事务场景的 TPS 提升了 49%. 除了只写事务场景, MySQL 多主模式较单主模式也有不错的性能提升, 但性能较 OceanBase 仍然有较大的差距, 读写混合、只读和只写 3 个场景下, OceanBase 的性能分别是 MySQL 的 1.27 倍、1.09 倍和 3.1 倍, 尤其在 OLTP 只写场景下, OceanBase 的性能有 3 倍多的优势, 如图 10 所示.

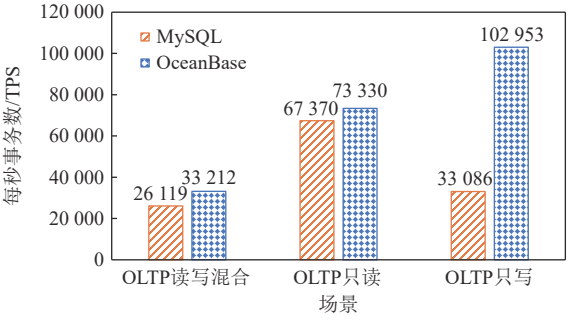


Fig. 10 Performance comparison between OceanBase and MySQL in three-replica multi-master mode

图 10 3 副本多主模式 OceanBase 和 MySQL 的性能对比

本文也对比了 MySQL 和 OceanBase 在多机 3 副本多主的集群模式下处理复杂 SQL 的性能，采用比较流行的 TPC-H 作为测试工具，测试采用的数据量是 100 GB. 实验数据表明，在多机 3 副本多主模式下 OLAP 复杂查询，OceanBase 的性能是 MySQL 的 6～327 倍，如图 11 所示.

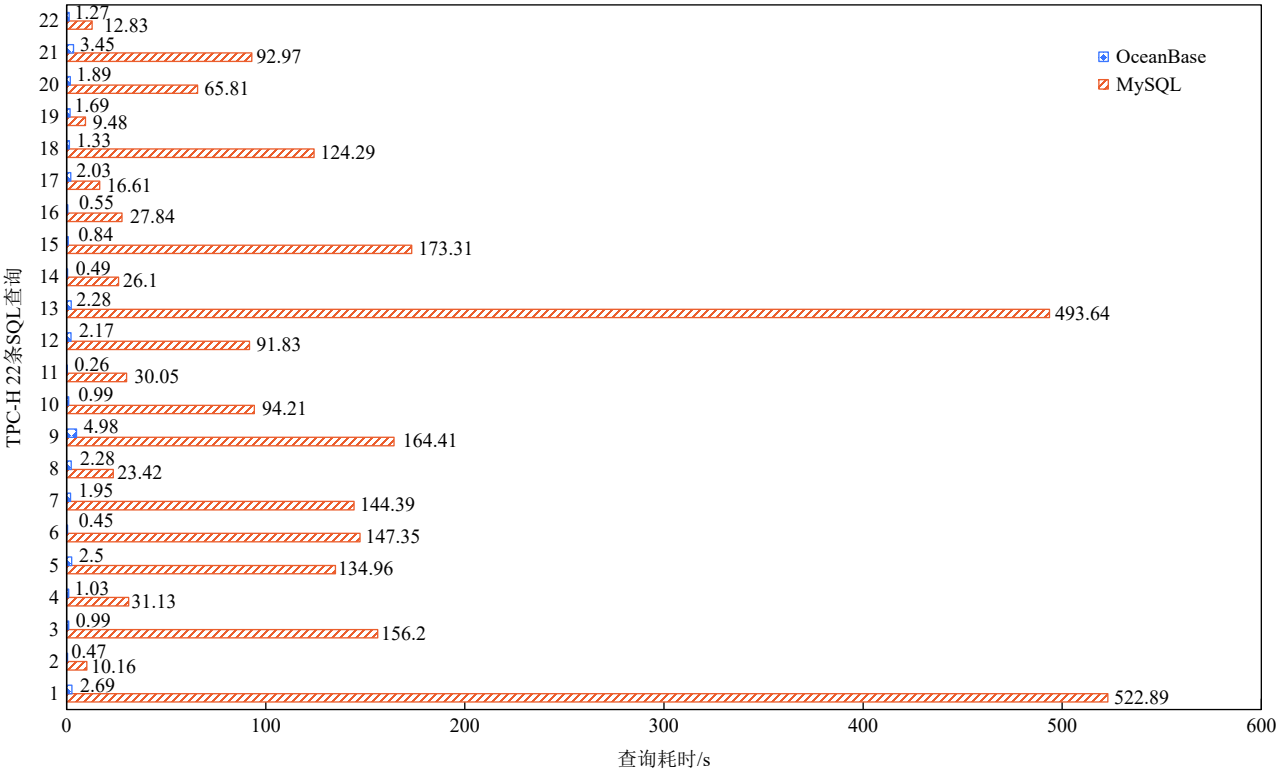


Fig. 11 Time-consuming comparison of TPC-H 22 SQL queries between OceanBase and MySQL

图 11 OceanBase 和 MySQL 的 TPC-H 22 条 SQL 查询耗时对比

8 相关工作

分布式关系数据库融合了分布式系统和传统关系数据库 2 个方面的理论和技术，两者的发展都对分布式关系数据库产生了显著的影响.

8.1 分布式系统

Google 分布式文件系统（GFS）[21] 实践了如何在一组普通 PC 服务器和普通数据中心网络之上构建一个可靠的海量数据存储系统来面向大文件提供高吞吐率的文件读写. GFS 每个文件由若干 chunk 组成，每个 chunk 有若干个副本（典型为 3），分别存储在多个 chunk server 上. 执行写入时，通过主从复制把数据同步写入到多个副本上. 为了自动容灾和提升写入性能，GFS 允许副本里出现重复的记录并允许空的占位记录，需要应用在读取时候特殊处理，这一设计与之前的要求强一致性甚至与 Posix 兼容的文件系统有显著不同. Google 分布式文件系统的 master-chunk server 架构、自动容灾处理、多副本、原子追加等经典设计，对之后的分布式存储系统产生了很大的影响，比如开源的 Hadoop 分布式文件系统（HDFS）.

MapReduce[22] 提供了一种用于海量数据处理的编程模型和分布式处理框架. 用户只需要定义一组简单的 map 和 reduce 函数，系统将自动把计算过程在一组普通 PC 服务器上并行化，并自动处理任务划分、任务调度、中间结果传输、节点故障容灾等. MapReduce 刻画了利用分布式系统分析处理海量数据的关键特征，对后来的大数据系统产生了深远的影响.

8.2 NoSQL 数据库

Bigtable[23] 构建在 GFS 之上，是一个用来管理 PB 级的结构化数据的分布式存储系统. Bigtable 的数据模型是一个支持元素级多版本的稀疏表，其中，元素的键和值都是无类型的二进制串. 为了支持实时更新，存储结构采用了 LSM-tree 的数据结构，对表的修改在记录持久化日志后，存储在内存 Memtable 中，并在后台按需与硬盘上的 SSTable 文件进行合并. Bigtable 的数据模型后来被称为宽列模型，是 NoSQL

类数据库的典型代表，开源项目 HBase 和 Cassandra 都受到其影响. Bigtable 只支持单行内的事务，不支持通用事务，也就无法通过维护二级索引来加速查询.

Dynamo[24] 是同时期另一个 NoSQL 系统，它是一个分布式键值存储系统. 为了在设计上达到极高的可用性目标，它的架构采用了去中心化的分布式架构，所有节点对等，整个系统中的组件避免任何中心节点. Dynamo 使用一致性哈希对数据进行分片，使得容灾处理和扩容、缩容时数据迁移量最小. 为了写操作的高可用，系统选择放弃故障时的强一致性，只保证最终一致性. Dynamo 无主副本节点的异步数据复制协议，适合于跨数据中心部署.

8.3 分布式关系数据库

Bigtable 虽然具有良好的扩展性，但是因为只支持单行事务，应用的并发修改无法确保数据集的一致性，提升了应用开发的难度和成本. 为了解决这个问题，Percolator[25] 在 Bigtable 的模型和服务之上，通过在行数据中添加版本和锁信息，对应用提供了支持快照隔离级别的通用事务. 但是因为任意的多行写操作都需要作为分布式事务处理，且对锁状态多副本持久化和延迟清理，这并不适合对事务处理时延有严格要求的 OLTP 系统.

Google Spanner[26-27] 实现了一个跨全球部署的分布式数据库，提供了外部强一致性，并支持 SQL 语言. Spanner 把数据进行分片，每个分片有多个副本，通过基于 Paxos 协议的分布式状态机保证多副本间的数据一致. Spanner 对后续的类似系统产生了显著影响，例如 CockroachDB[28] 和 YugaByteDB[29] 等. Spanner 开发了 TrueTime 时间服务以提供全球范围内误差很小的时间戳服务，避免了跨地域获取事务版本号的开销，但这要求读写操作必须在对应的时间戳所示时间之后进行从而增加了读写延时，同时 TrueTime 需要多台高精度的 GPS 原子钟以提供精确的时间，限制了它的业务应用和部署场景.

9 下一步工作

作为一个分布式数据库，OceanBase 实现了高可用以及在线水平扩展. 未来，OceanBase 期望能够在同一套系统内更好地处理多种不同的工作负载和数据类型，无论是事务型还是分析型，大数据还是小数据，结构化数据还是半结构化数据，满足企业从小到大成长过程中的全生命周期数据管理需求. OceanBase 系统的后续技术发展方向包括 5 个方面：

1)HTAP 混合负载. 多数业务不仅需要交易处理(OLTP)，还需要对业务进行多个维度的分析、生成各种报表(OLAP)等，当前 OceanBase 的 OLTP 功能比较丰富，而 OLAP 功能还有待进一步完善，OLTP 和 OLAP 事务处理的优先级、资源隔离等方面还有待进一步完善. 未来将进一步丰富 OLAP 功能，通过列式存储等技术提升 OLAP 的执行性能，并完善 OLTP 和 OLAP 混合执行能力.

2)Oracle/MySQL 兼容功能. 在功能支持方面，OceanBase 兼容绝大部分 MySQL 功能和大部分 Oracle 功能，包括存储过程、数据库安全等，然而，系统的查询优化能力与 Oracle 相比有一定的差距，数据库功能也不如 Oracle 丰富. 未来将加强 SQL 查询优化和并行执行能力，提升 Oracle 兼容性，进一步完善 MySQL 兼容性，进一步降低从 Oracle/MySQL 迁移到该系统的成本.

3)多模态数据处理. 多数业务不仅需要处理结构化数据，还需要处理其它类型的数据，例如地理信息数据、文档数据和键值数据等半结构化数据，并基于这些数据执行事务、分析、在线搜索等各种操作. 通过使用一套系统处理多种数据，能够简化企业的技术栈，降低企业的业务系统实现的复杂度. 当前，OceanBase 的关系模型数据管理功能比较丰富，其它数据模型管理功能还有不少需要完善之处. 未来将进一步丰富多模数据管理功能，并探索多种数据模型之间的互通操作.

4)多云原生数据库. OceanBase 支持多种不同的部署模式，包括公有云、专有云和独立软件部署. 公有云、多云、混合云将会是未来的趋势，因此，后续 OceanBase 将会进一步增强对全球主流云平台的支持和适配，并进行软硬件协同优化以实现更好的性价比.

5)自治数据库. 当前 OceanBase 运行维护参数较多，运行维护人员学习和使用有一定的学习成本. 未来我们计划利用机器学习和人工智能技术来分析系统运行环境状态和日志数据等信息，从而实现较为智能的动态系统参数调整、系统优化、在线预警、实时监测等，实现智能诊断、智能调优、智能运维等.

OceanBase 作为一个基础软件，生态构建至关重要. 2021 年 6 月 1 日，OceanBase 将内核代码(超过 300 万行)在 GitHub[30] 开源，目前已经有成百上千的企业使用 OceanBase 开源版本. 未来，OceanBase 会继续坚

持开源开放的路线，进一步加大对用户、开发者、合作伙伴的支持力度，基于开源社区的协作模式，通过大量用户的反馈不断迭代，不断提升 OceanBase 的成熟度，更好地满足用户的需求.

作者贡献声明：阳振坤提出论文思路、整体架构设计，参与文献调研，撰写和修订论文；杨传辉、韩富晟、王国平、杨志丰参与文献调研和论文撰写；成肖君参与文献调研和论文撰写，并负责性能对照实验.

参 考 文 献

[1] Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377−387

[2] Yang Zhenkun, Yang Chuanhui, Han Fusheng, et al. OceanBase: A 707 million tpmC distributed relational database system[J]. Proceedings of the VLDB Endowment, 2022, 15(12): 3385−3397

[3] Yang Zhifeng, Xu Quanqing, Gao Shanyan, et al. OceanBase Paetica: A hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster[J]. Proceedings of the VLDB Endowment, 2023, 16(12): 3728−3740

[4] Serlin O. TPC-C Details: 60, 880, 800 tpmC [EB/OL]. [2023-11-25]. https://www.tpc.org/1799

[5] Serlin O. TPC-H Result Details: 15, 265, 305 QphH@30000GB [EB/OL]. [2023-11-25]. https://www.tpc.org/3375

[6] Lamport L. The part-time parliament[J]. ACM Transactions on Computer Systems, 1998, 16(2): 133−169

[7] Gray J. The transaction concept: Virtues and limitations[C]//Proc of Int Conf on Very Large Data Bases. San Francisco: Morgan Kaufmann, 1981: 144−154

[8] Mohan C, Lindsay B, Obermarck R. Transaction management in the R* distributed database management system[J]. ACM Transactions on Database Systems, 1986, 11(4): 378−396

[9] Berenson H, Bernstein P, Gray J, et al. A critique of ANSI SQL isolation levels[C]//Proc of the 1995 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 1995: 1−10

[10] Bernstein P, Goodman N. Multiversion concurrency control−Theory and algorithms[J]. ACM Transactions on Database Systems, 1983, 8(4): 465−483

[11] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351−385

[12] Selinger P, Astrahan M, Chamberlin D, et al. Access path selection in a relational database management system[C]//Proc of the ACM SIGMOD Conf on Management of Data. New York: ACM, 1979: 23−34

[13] Graefe G, McKenna W. The Volcano optimizer generator: Extensibility and efficient search[C]//Proc of the IEEE Conf on Data Engineering. Piscataway, NJ: IEEE, 1993: 209−218

[14] Graefe G. The Cascades framework for query optimization[J]. IEEE Data Engineering Bulletin, 1995, 18(3): 19−29

[15] Levy A, Mumick I, Sagiv Y. Query optimization by predicate movearound[C]//Proc of Int Conf on Very Large Data Bases. San Francisco: Morgan Kaufmann, 1994: 96–107

[16] Kim W. On optimizing an SQL-like nested query[J]. ACM Transactions on Database Systems, 1982, 7(3): 443−469

[17] Chaudhuri S, Shim K. An overview of cost-based optimization of queries with aggregates[J]. IEEE Data Engineering Bulletin, 1995, 18(3): 3−9

[18] Kornacker M, Behm A, Bittorf V, et al. Impala: A modern, open-source SQL engine for Hadoop[C]//Proc of the 7th Biennial Conf on Innovative Data Systems Research. New York: ACM, 2015: 1−10

[19] Oracle. Adaptive SQL Plan Management (SPM) in Oracle Database 12c Release 1 (12.1) [EB/OL]. [2023-11-25]. https://oracle-base.com/articles/12c/adaptive-sql-plan-management-12cr1

[20] He Baohong. China Communications Standards Association. Database Development Research Report (2023) [EB/OL]. [2023-07-04]. https://www.c114.com.cn/market/39/a1236668.html(in Chinese)
(何宝宏. 中国通信标准化协会.数据库发展研究报告(2023)[EB/OL]. [2023-07-04]. https://www.c114.com.cn/market/39/a1236668.html)

[21] Ghemawat S, Gobioff H, Leung S. The Google file system[C]//Proc of the 19th Symp on Operating Systems Principles. Berkeley, CA: USENIX Association, 2003: 29−43

[22] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters[C]//Proc of the 6th Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 137−150

[23] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems, 2008, 26(2): 1−26

[24] DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store[C]//Proc of the ACM Symp on Operating Systems Principles. Berkeley, CA: USENIX Association, 2007: 205–220

[25] Peng D, Dabek F. Large-scale incremental processing using distributed transactions and notifications[C]//Proc of the USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2010: 1–15

[26] Corbett J, Dean J, Epstein M, et al. Spanner: Google's globally-distributed database[C]//Proc of the 10th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012: 251–264

[27] Bacon D, Bales N, Bruno N, et al. Spanner: Becoming a SQL system[C]//Proc of the 2017 ACM Int Conf on Management of Data. New York: ACM, 2017: 331–343

[28] Taft R, Sharif I, Matei A, et al. CockroachDB: The resilient geo-distributed SQL database[C]//Proc of the 2020 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2020: 1493−1509

[29] Cook B. YugabyteDB [EB/OL]. [2023-11-25]. https://www.yugabyte. com

[30] Yang Zhenkun. OceanBase [EB/OL]. [2023-11-25]. https://github.com/oceanbase

Yang Zhenkun, born in 1965. PhD. CCF fellow. One of the 1st Cheung Kong Scholars, Peking University. His main research interests include distributed system and database system.

阳振坤, 1965 年生. 博士. CCF 会士. 北京大学首批长江学者. 主要研究方向为分布式系统、数据库系统.

Yang Chuanhui, born in 1985. Master. Member of CCF. His main research interests include distributed system and database system.

杨传辉, 1985 年生. 硕士. CCF 会员. 主要研究方向为分布式系统、数据库系统.

Han Fusheng, born in 1985. Bachelor. His main research interests include transaction processing, storage engine, and consensus protocol in database system.

韩富晟, 1985 年生. 学士. 主要研究方向为数据库系统事务处理、存储引擎、一致性协议.

Wang Guoping, born in 1986. PhD. His main research interests include query processing and optimization in database system.

王国平, 1986 年生. 博士. 主要研究方向为数据库领域的查询处理和优化.

Yang Zhifeng, born in 1983. Master. His main research interests include query processing and resource scheduling in database system.

杨志丰, 1983 年生. 硕士. 主要研究方向为数据库系统查询处理和资源调度.

Cheng Xiaojun, born in 1982. Master. His main research interests include distributed system test and database system test.

成肖君, 1982 年生. 硕士. 主要研究方向为分布式系统的测试、数据库系统的测试.