

基于 OceanBase 的 SQL 优化研究

唐勇 甘国操 向海霞 杜武全

中国民航信息网络股份有限公司重庆分公司, 401122

摘要: 随着信创工作的推进, 国产数据库 OceanBase 作为众多应用的最佳选择, 掌握该数据库的 SQL 优化方法显得越来越重要。本文根据 SQL 作为影响性能的重要原因, 分析常见 SQL 问题后, 再阐述 SQL 性能问题定位、解决和验证等关键步骤的具体操作方法, 以普及 SQL 优化, 解决应用性能问题。

关键词: OceanBase, 性能优化, 执行计划

DOI:10.69979/3041-0673.24.2.039

1. 背景介绍

随着公司信创工作落实的推进, 应用国产化迁移的清晰, 数据库要求的明确, 越来越多的应用选择向国产分布式数据库OceanBase (以下简称: OB) 转移。无论是基于传统的ORACLE、MySQL应用迁移过来的, 还是直接基于OB研发的新应用, 最终都会呈现出大量的应用采用OB作为基础的数据库软件来支撑上层应用, 相应的要求业务应用研发人员需要同步更新升级知识体系, 从基于传统数据库实例或主备集群模式的研发升级到OB这种基于分布式实现的准内存数据库, 掌握基于其上的SQL优化能力, 作为对于研发人员能力要求的重要性日益凸显。

2. 掌握 SQL 优化的必要性

随着国产化工作的全面铺开, 应用越来越多的数据服务转由OB支撑, 数据库作为基础软件, 对应用的稳定运行、性能保障发挥着至关重要的作用。在当前OB优化器自身不够强大, 无法实现智能优化的情况下, 研发人员对分布式准内存数据库的技术能力亟需提升, 以充分发挥数据库优势, 降低软件本身不足的影响; 互联网的快速发展, 人们工作节奏加快, 对应用性能的要求提高, 掌握基于OB的SQL优化的需求显得日益重要。

2.1 OB 优化器的局限

传统数据库历经多年的沉淀, 以Oracle为代表的SQL优化器已相当成熟, 可基于代价对SQL重写优化, 最大限度降低对性能的影响。而OB作为新发展的数据库, 其优化主要还是基于规则的, 基于代价的能力较弱, 仅支持少数几种情况, 我们在利用开源社区提供优化方案和国产厂商持续完善优化器的同时, 更要主动掌握其运行原理, 发挥其优势, 从系统性能问题出

发, 关注并解决问题以满足业务需求。

2.2 技术能力的升级

我们经多年实战磨炼, 对以Oracle为代表的传统数据库已有较扎实的技术功底, 业界亦形成较完善的技术生态, 而OB作为具有自主知识产权的国产化数据库, 与传统的数据库无论在存储引擎、事务引擎上, 还是SQL引擎方面都存在较大差异, 亟需我们放低心态, 更新技术体系, 以充分发挥其优势优化性能, 其中性能调优、SQL优化等技术能力显得尤为重要, 据统计近 70%的性能问题可通过优化SQL在代码层面解决。

2.3 性能期望的提高

随着人工智能等新技术的普及, 人们工作节奏的加快, 对系统的性能要求不断提高, 从用户可接受的 10 秒响应时间到新统计揭示的 300 毫秒内, 且有进一步降低容忍度的趋势, 系统响应时间逐渐成为影响用户满意度的重要因素。作为应用研发人员希望提升系统性能, 改善用户体验, 性能优化成为提升非功能性指标的重要手段, 而以数据库为支撑的数据处理和查询又处于核心地位, 故研发人员掌握SQL优化, 对提升系统性能, 改善用户体验的重要性日益突显。

3. SQL 性能原因分析

综上所述, 我们认识到基于OB的SQL优化是研发人员能力要求的重要组成, 应了解常见的性能问题是怎么导致的, 才能制定有效措施解决问题。根据多年项目实践和查阅国内官方调研结论可知, 导致性能问题的主要原因是开发人员编写的SQL脚本, 因未深入理解底层执行原理, 未充分发挥数据库优势, 未在数据库设计和使用中重视性能, 如SQL中用星号(*)代替列名, 这样书写简单, 但导致大量的查询回表, 降低性能。在数据库

设计时，过度追求范式要求，减少存储冗余，但导致大量关联查询，类似情况举不胜举。其次，数据库软件本身的不足也带来性能影响，如数据库的统计信息不准，导致在数据查询时不能生成最优的执行计划；OB生成执行计划目前主要是基于规则，基于代价的能力相对较弱，导致不能随着系统环境变化而智能优化，同时OB是分布式数据库，为降低收集信息对性能的影响，一般集群合并是每天一次，而统计信息在合并时才更新，进而导致数据滞后优化不佳。

4. 具体实施方式

在梳理性能原因时，不难发现在实施优化中，首先要解决的是对SQL的调优，其次是对数据库吞吐量的优化。对单条SQL的优化是SQL性能调优的基础，又分单表和多表两种，单表重点要用好索引，减少不必要的排序等，同时用好分区控制查询量，提升效率；多表要优化连接算法，有效利用查询改写充分发挥分布式并行计算优势，提升性能。

针对吞吐量的优化，首先可通过ob_read_consistency对读一致进行设置，充分发挥分区优势；通过primary zone的设置分散主分区压力，结合业务分散到适当的OB Server上，在避免热点和降低跨机分布式中取得最佳平衡点；通过客户端路由策略及业务热点查询分区设置，做好分区均衡分布，避免跨机分布式事务影响，减少跨机跨分区的关联查询等。当然，通过技术手段找到影响性能的慢SQL，再聚焦分析其性能问题，针对性的优化将解决近80%的问题，故下面就慢SQL的定位、分析和解决进行阐述。

4.1 问题定位综述

要优化SQL提升性能，首先是准确定位SQL，再定位问题原因，才能采取有效措施解决问题，那如何精准定位呢？在OB中有多种方式定位，可通过设置trace参数在生成的慢日志中查看，可在OCP监控平台中查看，也可用全局SQL审计表gv\$sql_audit找到SQL后，通过分析耗时和内存、磁盘IO等资源消耗，再找到影响性能的SQL，确定节点服务器、端口、租户和Trace等关键信息，为后续精准定位执行计划奠定基础。

对单条SQL执行性能的分析，一般是通过查看执行计划定位问题，因执行计划中清晰展示SQL的执行流程及资源消耗，确定与预估不一致或对性能影响明显步骤，再制定解决措施，一般有立竿见影的效果，故会查看和理解分析执行计划就成为SQL优化的基础，是具体分析SQL执行流程的重要手段。

研发人员可执行EXPLAIN命令，查看优化器针对特定SQL生成的逻辑执行计划，分析其是否使用已有的索引，是否存在没必要的IO操作，是否有对业务无效的回表操作等，进而确定调优方向和措施。

4.2 问题解决步骤

通过前面分析，找到慢SQL是问题解决的关键，如何找到慢SQL的重要性就提升，至少可通过sql_audit结合SQL_TRACE、慢SQL日志、OCP监控平台等三种方式定位，下面以sql_audit查找慢SQL的方法为例做介绍，其他方法可参考实现。

sql_audit虚拟表是OB内置的全局SQL审计表，记录着租户在本周期内操作的每条SQL，可查到每次请求的客户端，执行请求的server信息和状态信息，等待事件及执行各阶段耗时等，从性能优化角度需重点关注以下四类数据。

Retry次数，重试次数(RETRY_CNT字段)。若该次数较多，可能是锁冲突或切主等情况导致请求重复，需结合其他数据和业务情况，定位原因后再采取相应措施。

Queue time，排队时间(QUEUE_TIME字段)。若该值过大，可能是CPU不够用，分析是什么任务导致，该任务是否存在优化空间，还是硬件确实无法支撑当前业务。

GET_PLAN_TIME，获取执行计划时间(GET_PLAN_TIME字段)。若该值过大，可能是未命中plan cache，都是硬解析完成，一般会通过IS_HIT_PLAN=0体现出来。当该类数据较多时，要分析SQL编写是否有改进空间，以提高plan cache命中率，减少硬解析次数。

EXECUTE_TIME，请求执行时间。若该值较大，则需进一步查看是等待事件耗时长还是逻辑读次数多，若是前者要关注等待事件原因，再采取措施解决；若是后者，可能是大账户涉及数据处理内容模块增加，可分解执行或结合业务实际优化执行时机等，如利用夜维等低峰时段处理大量逻辑读的内容。

为使SQL的执行时间或资源消耗符合预期，除上面提到的解决特定具体问题的方法外，常见问题可通过SQL优化解决，其主要操作方式有下面两种。

1) 单表访问做等价改写。在业务等价的基础上，充分利用数据表已有的索引等结构降低查询范围。同时避免查询无用的字段，减少不必要的回表，尽量通过索引得出业务的必需数据，该过程可通过Explain生成的执行计划判断是否达成最优。

2) 多表访问关注联接问题。对于多表访问既可用单表访问的优化手段, 同时还要关注访问路径、联接顺序和算法等, 主要思路包括能过滤的尽量提前做, 减少关联数据量; 联接顺序若联接两表结果集数据量差异较大时, 一般小表做驱动表, 减少数据关联循环次数; 有时联接算法也影响性能, 了解各种联接算法的实现原理对于优化也是非常重要的。对于OB这种分布式数据库, 可利用表组减少跨机分布式事务, 利用分区同步计算等机制充分发挥各节点并行优势; 利用数据分区特点, 进行分区裁减以减少数据量也是重要手段, 综合多种方式实现优化。

4.3 效果验证改进

综上所述, 确保SQL以最佳方式完成业务功能, 要理解explain的执行计划仅是理论的, 实际执行时未必会按预期的方式执行, 接下来就是要验证物理执行计划与逻辑执行计划的一致性, 如何判断是否一致呢。

通过Explain可查看如下图所示的逻辑执行计划。

SQL Query Plan	
1	=====
2	ID OPERATOR NAME EST.ROWS EST.TIME(us)
3	=====
4	0 TABLE GET T1 1 3
5	=====
6	Outputs & filters:
7	=====
8	0 - output([T1.ID], [T1.NAME], [T1.REMARK]), filter(nil), rowset=16
9	access([T1.ID], [T1.NAME], [T1.REMARK]), partitions(p0)
10	is_index_back=false, is_global_index=false,
11	range_key([T1.ID]), range[1; 1],
12	range_cond([T1.ID = 1])

将SQL在物理环境中真实执行, 通过v\$ob_sql_audit视图, 可找到SQL实际执行记录, 为过滤其他记录的干扰, 可使用Query_sql字段利用具体SQL文本找到关注的SQL, 记录下对应的执行计划ID, 即PLAN_ID, 再在v\$ob_sql_plan中, 利用刚查到的PLAN_ID找到对应的执行计划, 即物理执行计划, 可分析是否符合预期, 保持与逻辑执行计划一致。

当发现物理执行计划与逻辑执行计划不一致时, 可使用Hint通知优化器优化执行计划, 但对于已上线的业务, 若出现优化器选择的计划不优时, 而Hint一般需调整业务代码, 故采用在线绑定计划, 通过DDL将一组Hint加入到SQL中, 无须业务SQL更改, 使优化器根据指定的Hint对优化SQL执行计划, 该组Hint就称为Outline。通过Outline完成手工绑定执行计划, 解决优化问题就成为优化的常见解决方案。下面介绍手工绑定执行计划的方法和实现过程。

1) 创建 OUTLINE。OB数据库支持两种方式创建OUTLINE, 一种是通过用户执行的带参数的原始语句—sqlText, 如用CREATE [OR REPLACE] OUTLINE outlineName ON sqlText创建, 其中outlineName为OUTLINE名字, 方便以后查找使用, sqlText为使用hint的SQL, 该方式创建后, 再遇到仅去掉hint的SQL时就会使用该hint; 另一种是通过sqlId创建, 如CREATE OUTLINE outlineName ON sqlId USING HINT hint语句创建, 其中sqlId可通过v\$ob_sql_audit或v\$ob_sql_plan等查到, 建议使用该方法创建, 主要是这样可支持SQL参数化, 支持范围更大。在dba_ob_outlines中若能查到相应记录, 表示创建成功, 相应的SQL就会使用该hint优化执行计划, 完成手工绑定。

2) 生效OUTLINE。通过v\$ob_plan_cache_plan_stat视图查看对应SQL实际使用的执行计划, 确认是否生效, 可通过服务器IP和端口及租户ID和sqlId找到对应执行计划, 查看outline_id若与dba_ob_outlines中的outline_id相等, 则OUTLINE已生效, 若不一致或为-1则表示未生效, 一般可能是SQL不一致或创建OUTLINE有误, 需检查更新后再通过v\$ob_plan_cache_plan_explain查看物理执行计划确认。

5. 结语

数据库性能优化是一项对实践经验要求非常高的工作, 不仅考虑SQL层面优化, 还有数据库设计、参数优化等多方面。单SQL优化亦是需不断学习并结合项目实践分析可改进空间, 在实践中持续总结, 将成功经验推广应用, 把踩过的坑控制在最小范围, 利用当前条件达成最佳的性能目标, 辅以个人经验进行全方面的考虑, 尽量实现当前条件下的最佳性能, 故性能优化是一项永远在路上工作。这需我们技术人员保持谦卑的心态, 孜孜不倦的学习, 以永不知足的劲头持续努力, 以助力最小的研发投入达成最佳的业务性能。

参考文献:

[1] 彭煜玮 OceanBase 数据库源码解析: 机械工业出版社

作者简介: 唐勇 (1983 年 3 月), 男, 汉族, 重庆人, 学士学位, 资深软件工程师, 中国民航信息网络股份有限公司重庆分公司