# TXSQL 并行查询技术思考和实践

2023-08-06 17:08

# 目录

# 引言

对于商业数据库[5][6][7]，开源数据库[8]，云原生数据库[9][10]，或者大数据系统[32]，并行计算[33]都是多核处理环境下提高性能的基本技术手段。本文以 MySQL 数据库内核为例，分析了如何在庞大代码库上构建并行计算能力，并通过关键抽象来划分层次和管理复杂性。

# 摘要

腾讯云托管数据库 TencentDB for MySQL [1]（本机存储，Binlog 复制集群）和云原生数据库 TDSQL-C for MySQL [2]（共享存储， Redolog 复制集群）都采用基于 MySQL 增强的 TXSQL 内核。这两个产品广泛应用于事务处理 (TP) 场景，在分析型处理 (AP) 场景曾经存在明显短板。针对这个问题，两款产品先后发布了并行查询功能，支持调动多核资源加速 AP 查询，在 TPC-H [3]多条查询上都有明显提速，而且在 sysbench [4]场景下几乎没有衰退。TXSQL 内核遵循抽象、复用和扩展的基本原则，以尽可能少的改动，实现比较复杂的业务需求，积极跟进社区新版本特性，追求业务价值最大化。具体地讲， TXSQL 内核在 8.0 上构建了并行查询基础框架，结合 MySQL 代码特点，采用计划复现方案，解决并行任务（执行计划片段）的分发难题，常规计算逻辑可以快速适配到并行计算。

# 背景介绍

原始版本的 MySQL 只支持查询间并行，一条查询由一个线程处理（单线程），这在高并发的 TP 场景表现良好，但是对于数据量比较大的 AP 查询就无法充分利用系统资源，因此需要引入查询内并行（即并行查询）。另外， MySQL 8.0 计算进行了大规模重构，执行层已经统一到经典的火山迭代器模型[11]，为并行查询的实现提供了良好基础。

并行计算的本质是**任务拆分**和**任务调度**，将总计算任务拆成更小的子任务（计算并行），将需要处理的数据集拆成更小的分区（数据并行），使得不同 CPU 可以独立处理任务，实现多核并行加速。但根据 Amdahl 定律[14]，并行加速效果受限于总任务量中可并行子任务的占比 p ，p 越大理论加速比越大 ，因此，并行查询的基本目标是在各种场景下尽可能提高 p 值。

注：MySQL 已经在 InnoDB 层实现并行数据扫描[12]和并行 DDL [13]，虽然也是“并行执行”，基本理论相通，但其实现局限于存储层内部，而且形态比较简单，不在本文讨论范围内。此外，并行查询默认支持行迭代模型，但也可以和列式计算模型[17]对接，实现多重加速效果的叠加。

# 基本概念和关键抽象

关系数据库的基本运算是集合迭代运算[34]。TXSQL 并行查询在基本运算之上构建并行任务，支持算子内并行和算子间并行[18]。每个并行查询都由一个**协调者** (Coordinator) 和多个**执行者** (Executor) 配合完成*分解任务*的调度执行。协调者是一个*角色*，它可能是一个专用线程支持，也可能是特定线程的代理角色[19]。这些角色线程是一个进程内的，将来也可以扩展到不同进程，甚至不同主机上。线程间交互按相对位置采用匹配的通信机制，例如进程内为无锁队列，跨进程采用 IPC 机制，跨主机采用 RPC 机制，但对于调度而言，只需要关注*交互内容*（命令传达和状态反馈）而非通信介质。

并行执行计划是在原始执行计划上构建的粗粒度表示，基于四个关键抽象：并行任务，数据交换，分区队列和任务依赖图。**并行任务**通常称为执行计划片段 (Fragment) 、切片 (Slice) [31]、步骤 (Stage) [32]或者数据流算子 (Data Flow Operation, DFO) [6]，由一个或多个迭代算子 (Iterator) 构成的树。显然，在输入和输出的表示上，执行计划片段和行迭代算子是统一的，其输出由根算子产生，而输入就是叶算子的输入。**数据交换** (Exchange) 封装了并行任务之间的数据分发机制。**分区队列**里是数据横向切分的区间边界信息。**任务图**表示执行计划片段的调度顺序，是一个有向无环图。

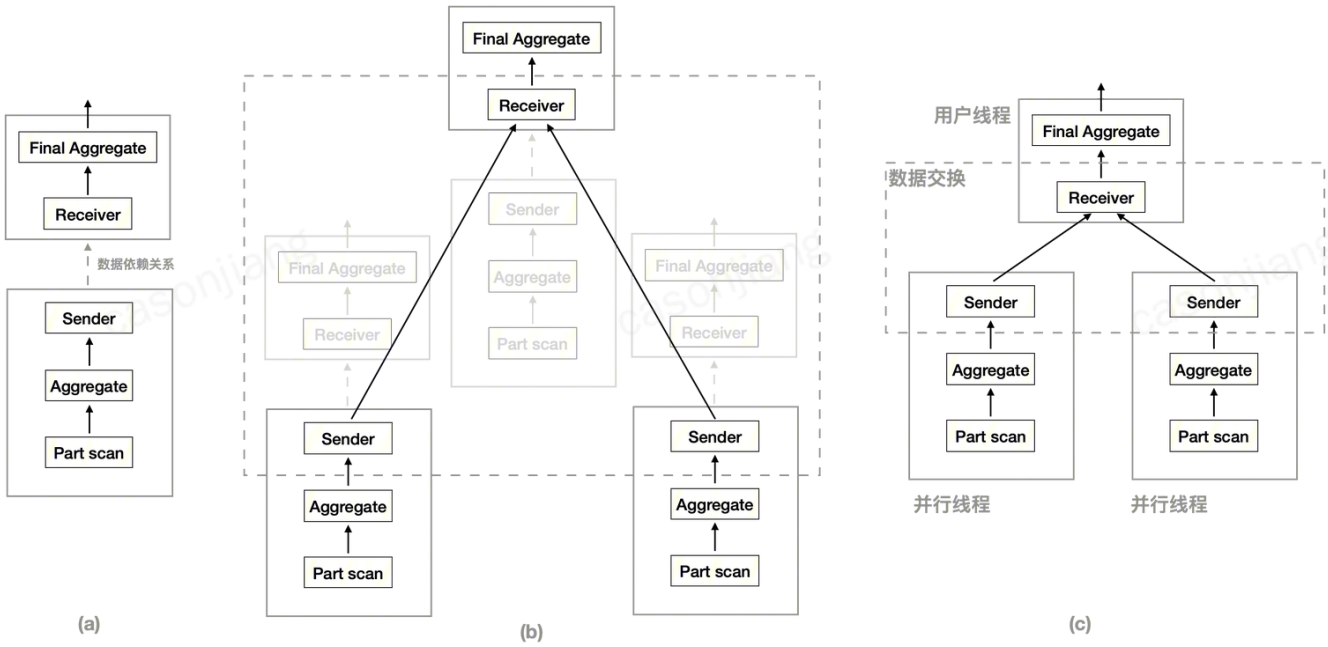
# 并行执行模型

TXSQL 并行执行模型中，不同线程上的执行计划片段形成数据流连接，共同构成一个逻辑上的全局执行计划（数据流图），执行时按照数据流动顺序调度执行。每个执行计划片段内的执行模型仍然是经典的火山迭代执行模型。

## 构造并行执行计划

TXSQL 优化器采用**两段式优化** (Two-Pass Optimization) [21] [6]，第一阶段确定 JOIN 表序和单表的访问方法，第二阶段确定执行计划片段和数据交换算法。然后对执行计划进行修改，插入数据交换的对接算子 (图 1 (a) 中的 Sender 和 Receiver ) 。 Coordinator 会先生成执行计划， Executor 也会生成相同的执行计划，然后这些线程上的执行计划片段之间形成连接 (图 1 (b) ) ，构成一个全局执行计划 (图 1 (c) ) 。

图1 并行执行计划

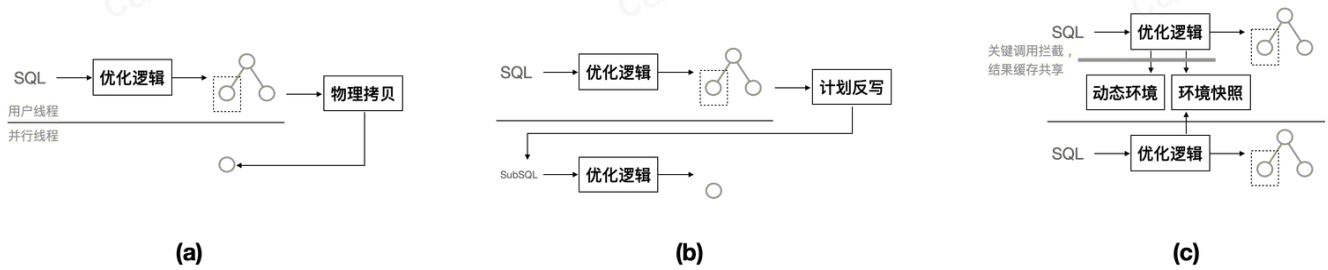



相对于**一段式优化** (One-Pass Optimization) [22]，两段式优化搜索空间小，对社区原生优化器侵入少，但有可能受第一阶段限制而产生局部最优计划。采用一段式还是两段式，更多地是一种现实权衡，这两种方式各有优缺点，也都实践有效并有商业数据库采用。考虑到 MySQL 社区还在持续地重构代码推进新优化器 [15]，两段式优化几乎是 MySQL 基础上实现并行查询的唯一合理选择。

## 任务形态的选择

Coordinator 将执行计划片段分发到 Executor 中，通常需要基于一种*中间形态*： (1) 对片段的物理结构进行完备的 (self-contained) 描述，这些信息可以拷贝到 Executor 中重新构造出相同的物理结构，称为**计划拷贝** (图 2 (a) ) ，(2) 将执行计划片段的物理结构表示成相应的 SQL ，再在 Executor 上经优化器构造出相应的物理结构，称为**反写 SQL** (图 2 (b) )。但是，也可以 (3) 控制优化器基于原始 SQL 产生相同的执行计划，并对执行计划片段进行标识，Executor 根据标识定位到片段并执行，称为**计划复现** (图 2 (c) )。

图 2 并行任务形态

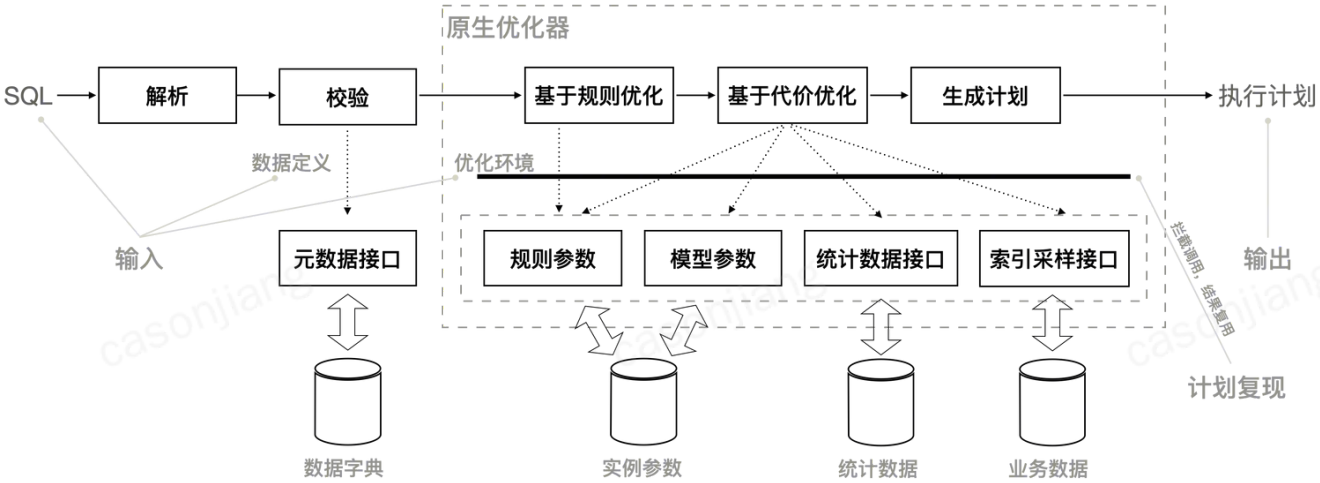



考虑到 MySQL 复杂的代码历史，社区并行查询试探性工作基本放弃 [14]，但仍然推动了持续重构和引入新功能。随着 TXSQL 并行功能的不断深入，前两种方式的代码维护会越来越困难。毕竟在没有原始设计的支持时，凭空构造中间形态过于复杂而且容易出错 [6]。对于计划拷贝，物理结构的完备描述本身就是一个难题，而且随着新特性的并行化支持，代码维护也是个问题；如果将物理结构表示成相应的 SQL ，底层的迭代运算结构和 SQL 的声明式语言并不直接匹配；而计划复现则可以充分利用优化器现有能力，通过控制优化过程来生成相同的物理结构。

### 计划复现机制

从 SQL 产生执行计划的操作，可以用抽象函数表示为`执行计划 = 优化函数(SQL，元数据，优化环境)`[20][24]。这是一组确定性逻辑，只要输入参数相同，就可以产生相同的输出。元数据变更需要经过 DDL ，在 SQL 执行期间不变。优化环境由启发式规则参数、代价模型参数、常规统计信息和索引实时采样结果组成。与其他数据库优化器不同，MySQL 优化器还支持优化阶段读数据求值（优化期执行[23]）并在余下优化过程中使用该值，这个值也算是优化环境的一部分。

图 3 优化环境




计划复现是商业数据库优化器的基本功能，但 MySQL 并不具备这个功能，不过 MySQL 优化器的数据接口还是比较清晰的。因为 SQL 优化前并不能确定是否需要并行执行，而优化环境的记录会造成额外开销，所以，需要有极致的优化，避免在短查询场景性能回退。

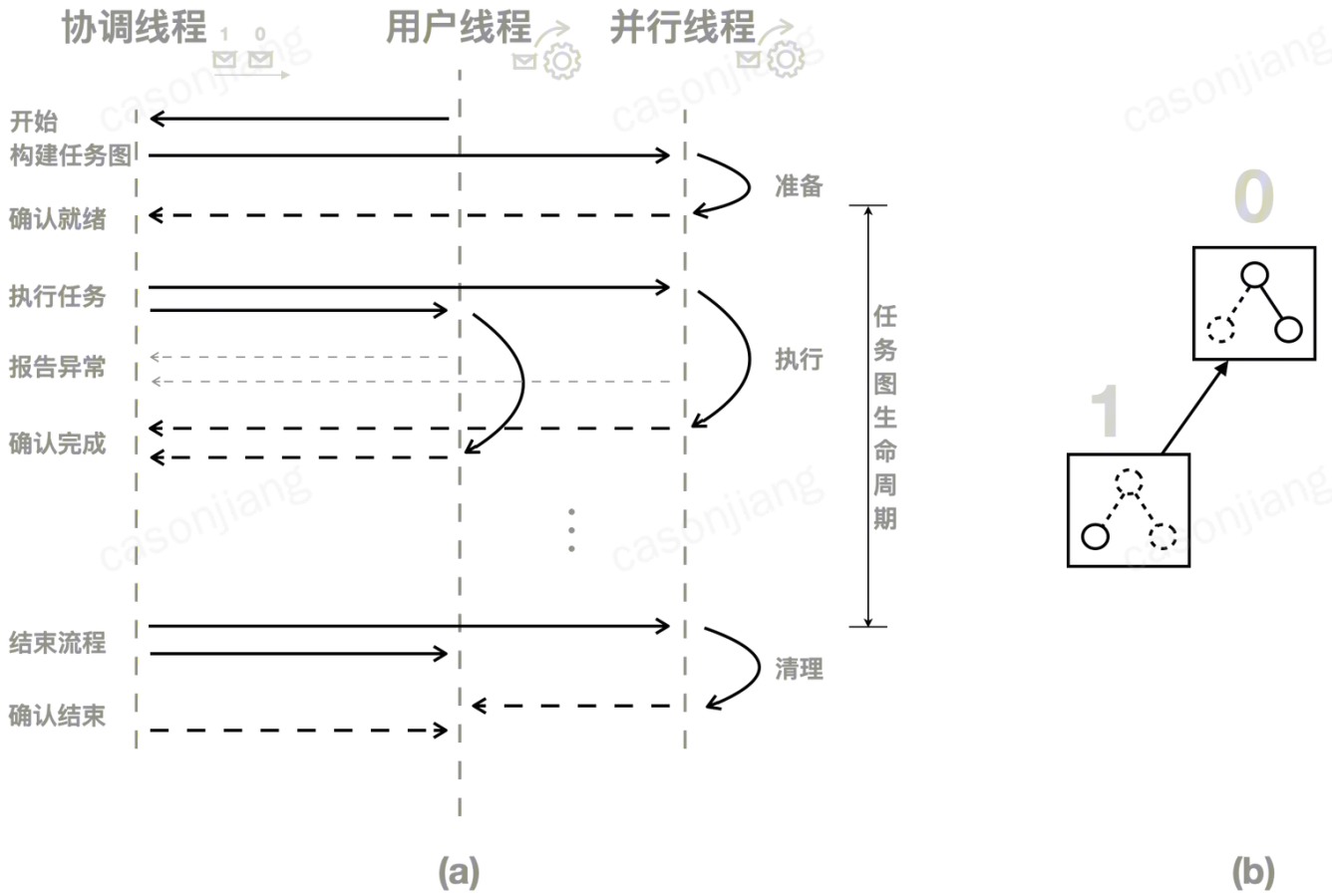
除此之外，所有线程上的执行计划都会进行二次校验，确保物理结构语义相同。

## 任务调度执行

如上所述，所有线程都需要构建相同的执行计划，这些执行计划的片段之间形成连接，共同构成一个全局数据流图，然后开始调度执行。从调度视角看，只需要关注交互命令[27]，通过交互来控制任务的依赖和同步。这里调度有*两层含义*，第一层是任务之间按一定顺序执行（**有向图调度**），第二层是同样的任务逻辑被分发给多个线程，这些线程同时独立处理各自的数据分区（**并行调度**）。

例如图 2 (b) 产生两个片段，片段 0 是数据接收端，片段 1 是数据生产端，按拓扑序可以得到调度序列 (1, 0) 。那么，协调线程在第一步就把生产端片段标识 1 和接受端标识 0 分别发给相应的线程组 (图 2 (a)) 。每组线程个数由并行度 (Degree of Parallelism, DoP) 决定。但这里 0 要向客户端返结果，也称为用户服务线程，通常是一个线程。显然，0 已经被动调度，第二步中不需要再调度。

图 4 任务调度




常规调度逻辑只需要按依赖顺序执行有向图的所有节点，但有一种比较特殊的情况是接收端任务提前结束，那么，发送端任务感知接收端关闭而结束后，调度系统就不再需要执行发送端任务的兄弟节点。

此外，任务调度执行还需要和 MySQL 异常框架集成。MySQL 通过 `Internal_error_handler` 和 `my_error()` 模拟了异常处理机制。异常可能在数据处理过程中抛出，或者响应其他线程的中断信号（`THD::killed_state`）时抛出。这些异常都是不可恢复的，如果发生，就需要中止调度，通知所有线程结束，并向客户端返回异常信息。

# 任务拆分与对接

## 可分解计算

并行查询的本质是任务拆分和调度，将总计算任务拆成更小的子任务，让不同 CPU 核心独立处理子任务。将原始执行计划拆解成片段，要看计算逻辑是否支持拆解。

可分解运算包含关系运算和统计函数[LZOJ]两类。显然，`Filter`，`Join`，`Sort` 和 `Limit` 都支持在数据分区上进行局部计算，并将局部结果进行轻量化合并而得到最终结果。`count()`，`sum()`，`min()`，`max()`，`avg()`，`stddev()` 和 `rollup` 等统计操作，也可以基于局部结果合并而成。

## 流水线断点

虽然关系代数定义的是集合运算，但是这个集合是一个概念，不一定要有一个物理上的集合。例如，`Filter(Scan(t1))` 可以基于一个行缓冲区完成所有计算，而 `Sort(Scan(t1))` 则需要在 `Sort` 中暂存所有记录并排序。统计函数虽然表现形式上为函数，但实际上需要两次迭代，第一次迭代输入更新中间状态，第二次迭代中间状态输出结果。这些临时存储点称为**流水线断点**。那么，只需要更换数据临时存储，就可以完成数据交换的对接。

## 数据交换站点

每一对“生产端–消费端”执行计划片段之间就是一个跨线程数据交换站点 (Exchange [18] )。片段可以分发到一个或者多个线程中执行。那么，每个线程上的发送端和接受端就构成了 N:1, 1:N 或者 M:N 数据网络。这些收发路由策略在第二阶段优化中决定。收发端算子只需要将网络中的消息流进行编解码，并放到迭代算子输入/输出内存中，并且及时处理空等、拥塞和异常情况。

## 数据动态分区

MySQL/InnoDB 存储是 B+ 树[29]，这是一颗平衡树，同一层级上的元素可以近似理解为代表相同大小的分区。从这个列表可以获得均衡的分区列表。但是，分区数量需要比并行度更大才能获得比较好的实际均衡效果，这就需要对列表节点进行更深层次的划分。

实际上 InnoDB 已经支持简单的动态分区读[12]，但功能场景比较受限，所以，TXSQL 内核进行了必要的扩展，使其支持多种扫描方式。

相对于静态分区，动态分区更有利于调动资源[30]。

# 项目迭代与质量保证

TXSQL 并行查询遵循迭代开发过程，先构建基础框架，然后逐步支持更多计算结构，同时进行性能优化。 SQL 可以从部分支持并行，到完全支持并行，这种渐进支持由并行优化前置的兼容性检测环节实现。兼容性检测模块扫描整个执行计划，识别 (1) 理论上不支持并行、(2) 理论上可以并行但是当前暂不支持和 (3) 可以并行三种操作，只在可并行范围内拆出并行任务，其他片段都标记为串行任务。

虽然 TXSQL 内核在设计上尽量和社区保持清晰的代码边界，但是仍然引入了较多的代码。在质量方面，除了新写测试用例进行覆盖外，还挖掘了社区原有回归测试集，这些测试集数据量都比较小，功能覆盖有限，为了丰富测试手段，在测试模式下支持行级并行分区，验证各种回归场景。另外，通过混沌测试、对比测试和压力测试等进行充分的功能检验。

# 性能

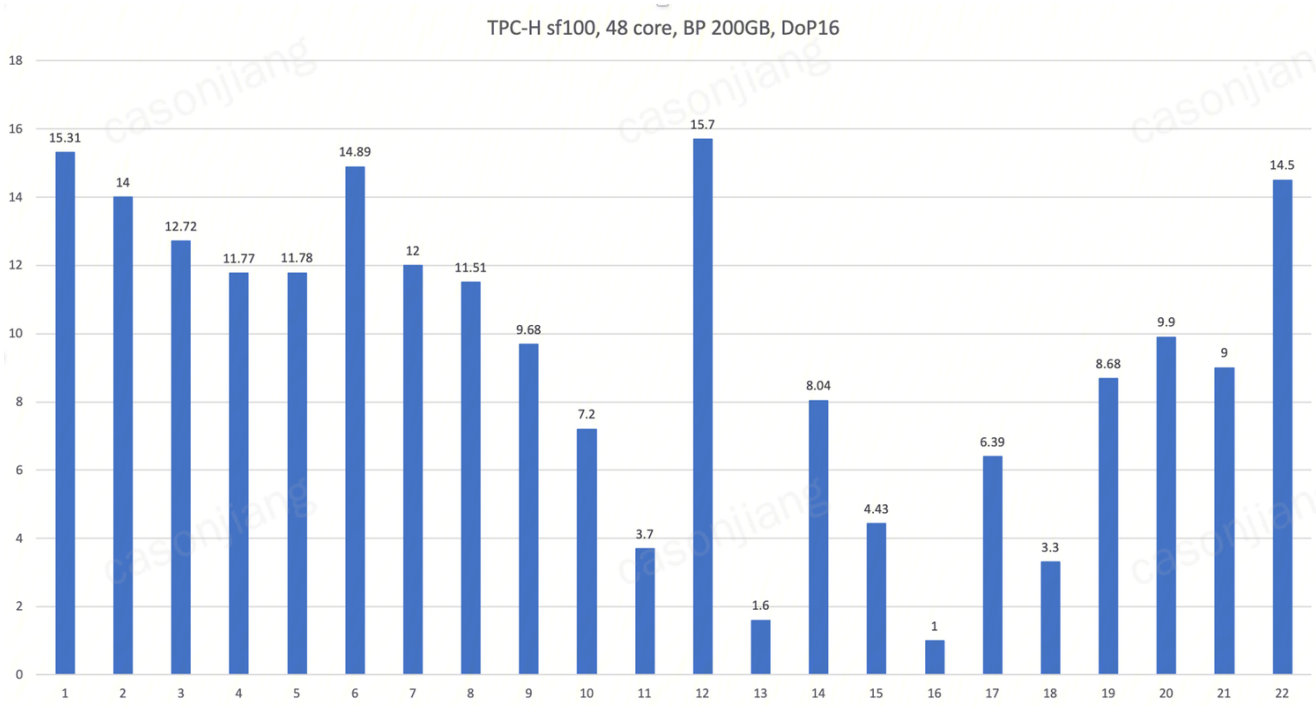
## 并行准备开销

TXSQL 内核采用计划复现方案，需要记录优化环境以备并行之需。因为决定采用并行执行是在第一阶段优化后，这些准备工作会对短查询也可能会产生一定的性能影响。经深入优化，功能开启时 sysbench read_only 性能损失不到 1% ，功能关闭时无性能损失。

## 并行加速效果

TPC–H [3]是行业基准测试，里面有 22 条查询模拟一个分析型业务，这里面有单表、多表、分组、聚合，排序，独立子查询和相关子查询。在 48 核 256GB 内存 200GB Buffer Pool 配置下，进行 scale factor 100 （其中 lineitem 单表约 6 亿行）的 TPC–H 测试，并行度设置为 16 ，大部分查询都有明显提升：

图 5 TPC-H 性能




已发版本支持 q1 q3 q4 q6 q9 q12 q14 q19 ，包含单表聚合和多表连接场景。内测版本支持了 q11 q15 q18 q22 为独立子查询并行，q17 q20 q21 为相关子查询并行，q2 为 in-memory hash join 并行，q5 q7 q8 q10 q21 为内表并行。未来版本会继续优化 q13 q16 并行加速效果。

## 结论

TXSQL 内核遵守抽象、复用和扩展的基本原则，以相对较小的工程代价实现了比较丰富的并行查询能力，并且及时跟进社区版本。这也表明虽然 MySQL 遗留代码复杂，但依然是有规律可循的。计划重现方案看上去似乎不太灵巧，实际效果却相当不错。目前实现还有不少改进空间，随着业务更多反馈，产品能力会持续打磨。

## 参考资料


[1] cdb: https://cloud.tencent.com/product/cdb
[2] cynosdb: https://cloud.tencent.com/product/cynosdb
[3] TPC-H: https://www.tpc.org/tpch/default5.asp
[4] sysbench: https://github.com/akopytov/sysbench
[5] Baru, Chaitanya K., et al. "DB2 parallel edition." *IBM Systems journal* 34.2 (1995): 292-322.
[6] Cruanes, Thierry, Benoit Dageville, and Bhaskar Ghosh. "Parallel SQL execution in Oracle 10g." Proceedings of the 2004 ACM SIGMOD international conference on Management of data. 2004.
[7] Shankar, Srinath, et al. "Query optimization in microsoft SQL server PDW." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012.
[8] PostgreSQL parallel query: https://www.postgresql.org/docs/9.6/parallel-query.html
[9] Aurora parallel query: https://docs.aws.amazon.com/zh_cn/AmazonRDS/latest/AuroraUserGuide/aurora-mysql-parallel-query.html
[10] PolarDB parallel query: https://help.aliyun.com/document_detail/128615.html
[11] WL#11785: Volcano iterator design: https://dev.mysql.com/worklog/task/?id=11785
[12] WL#11720: InnoDB: Parallel read of index
[13] MySQL Parallel DDL: https://blogs.oracle.com/mysql/post/mysql80-innodb-parallel-threads-ddl
[14] WL#2006: Parallel Query Execution in MySQL
[15] WL#14071 Hypergraph join optimizer: https://dev.mysql.com/worklog/task/?id=14071
[16] Amdahl's law: https://en.wikipedia.org/wiki/Amdahl%27s_law

[17] Abadi, D. J., Madden, S. R., & Hachem, N. (2008, June). Column-stores vs. row-stores: how different are they really?
[18] Graefe, G. (1990). Encapsulation of parallelism in the volcano query processing system.
[19] Teeuw, W. B., & Blanken, H. M. (1993). Control versus data flow in parallel database machines
[20] Chaudhuri, Surajit. “An overview of query optimization in relational systems.” Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. 1998.
[21] Pirahesh, Hamid, Joseph M. Hellerstein, and Waqar Hasan. "Extensible/rule based query rewrite optimization in Starburst." *ACM Sigmod Record* 21.2 (1992): 39-48.
[22] Graefe G. The cascades framework for query optimization[J]. IEEE Data Eng. Bull., 1995, 18(3): 19-29
[23] Refactoring Query Processing in MySQL: https://db.cs.cmu.edu/events/quarantine-db-talk-2020-mysql/
[24] MySQL 统计信息的现状与发展 http://mysql.taobao.org/monthly/2020/12/05/
[25] discomposable aggregation: https://en.wikipedia.org/wiki/Aggregate_function
[26] Astrahan, M. M., & Chamberlin, D. D. (1975). Implementation of a structured English query language.
[27] csp: https://en.wikipedia.org/wiki/Communicating_sequential_processes
[28] Selinger P G, Astrahan M M, Chamberlin D D, et al. Access path selection in a relational database management system[C]//Proceedings of the 1979 ACM SIGMOD international conference on Management of data. 1979: 23-34.
[29] Comer, Douglas. "Ubiquitous B-tree." *ACM Computing Surveys (CSUR)* 11.2 (1979): 121-137.
[30] Rahm, Erhard. "Parallel query processing in shared disk database systems." ACM SIGMOD Record 22.4 (1993): 32-37.
[31] GreenPlum: https://gpdb.docs.pivotal.io/6-17/admin_guide/query/topics/parallel-proc.html
[32] Spark: https://spark.apache.org/docs/latest/cluster-overview.html
[33] Parallel Computing: https://en.wikipedia.org/wiki/Parallel_computing
[34] Relational Database: https://en.wikipedia.org/wiki/Relational_database