YashanDB 调优手册

v23.2.2.0

2024年5月

深圳计算科学研究院

深圳崖山科技有限公司

数据库性能基础

数据库作为应用系统的数据载体和底层平台，其性能对整个系统的运行质量起决定性作用，而随着技术快速发展，数据量的几何级扩张和并发量的倍速增大，都对数据库性能提出了极高的要求。

决定数据库性能的因素有很多，例如运行在数据库中的应用模型、部署数据库的硬件环境、数据库的配置参数等等。通常情况下，数据库很难直接达到理想的性能要求，而是需要从不同维度来优化其性能表现，这个过程称为数据库性能调优。

从性能调优介入数据库状态的时间点上，可以将调优分为初始化调优和运行时调优。

例如，在设计一个应用系统时，应该同步进行数据库性能的设计，用户需要考虑硬件配置、软件能力、预设数据量和并发量等各种因素，合理地构建软硬件架构，和数据库的初始化配置。

此外，为保证在各种环境不断变化地情况下，应用系统仍能一直高效运行，数据库优化人员需要时刻关注数据库的性能状态，必要时从如下两个层面进行数据库性能调优：

- 实例级：从整体上识别数据库实例的瓶颈，通过对应的优化措施消除这些瓶颈，从而解决性能问题。
- SQL级：针对应用SQL进行优化，性能优秀的SQL语句可以有效降低执行开销，提升应用性能。

上述调优手段并不一定固定在数据库的某个时段，例如某个配置参数在初始化阶段调优后，在数据库运行过程中仍可能被识别出不合理，需要对其进行调优。

本章节将对数据库性能调优涉及的上述概念进行介绍，后续章节将介绍为实现这些调优，YashanDB所提供的具体手段和方法。

应用性能调优

系统架构

桥梁设计中桥梁的强度取决于最薄弱的部分，性能设计与之类似，应用系统自顶向下的每一层组件都需要满足性能要求。用户需要考虑不同的负载场景来设计不同组件的架构，主要包含硬件架构和软件架构两部分。

硬件架构

与数据库性能相关的硬件架构，在设计时应该主要考虑以下几个因素：

- CPU：决定数据库的执行能力和并行处理能力。
- MEMORY：数据库使用大量的内存来缓存数据，以减少I/O访问。
- I/O：不同磁盘性能差异很大，有些磁盘还可以提供容错能力，需要根据数据库的需求来选择。
- NETWORK：决定数据库的通讯效率，要重点关注带宽和迟延。

软件架构

通常情况下，应用系统都包含以下软件组件：

- 用户接口
- 业务逻辑
- 资源管理
- 事务处理

用户在设计这些软件组件时，也应当考虑数据库的性能，原则是尽量降低数据库压力。

应用设计原则

数据模型设计

应用通过创建数据库表，来规划设计业务的数据模型与关系。不合理的数据模型设计，会对数据库的性能产生很恶劣的影响，后期也很难通过SQL来优化。在设计YashanDB的数据模型时，通常需要注意以下几点：

1.选择合适的表类型：

- 对于海量稳态数据的分析业务，建议使用LSC表，通过稀疏索引，条件下推，数据压缩等关键技术，实现极致的查询性能。
- 对于实时分析的业务场景，可以选择TAC表，TAC表支持索引，支持in-place update，拥有向量化计算能力，可实现数据的快速更新和查询。
- 对于交易性的业务场景，可以选择HEAP表，实现高并发在线事务处理的极致性能。

2.建立合理的索引：

- 合理设计索引列，避免表上的列被多个索引反复包含。
- 创建联合索引时要注意字段排列顺序，把过滤最多记录数的字段放在第一位。

3.选择合理的数据类型：

- 选择更“小”的数据类型，例如INT类型会比NUMBER类型的性能更高。
- 选择合适的数据类型，例如对于字符型，如果数据的长度差异比较大，建议使用VARCHAR，节约空间，且对IO压力小；如果数据都是定长或者非常接近，建议使用CHAR，有特殊的性能优化效果。

4.选择合理的表分区属性：

- YashanDB支持哈希分区、范围分区、列表分区。
- 通过分区剪枝的特性，在数据访问时可以快速过滤数据，大大减少访问的数据量，提升性能。
- 通过分区特性，可以更好的提升并行扫描的能力。

5.分布式部署中选择合理的表分布属性：

- 对于数据量不大的表，可以考虑创建为复制表，减少跨节点Join。
- 根据数据分布和访问的均衡度考虑确定分区键，可以充分利用各数据节点资源，提升性能。
- 充分利用分布剪枝的特性，该特性在数据访问时可以快速过滤数据，大大减少访问的数据量，提升性能 。

系统开发架构设计

在任何系统的设计和架构搭建阶段，都应该确保开发人员充分关注SQL的执行效率。

系统开发架构需要遵循以下两点建议：

1.良好的数据库连接管理：

建立数据库连接的代价非常高，而且扩展性很差。因此，应用程序对数据库的并发连接数应该尽可能的少。 对于高并发的系统，应确保数据库连接被池化，通过数据库连接池来复用连接，而不是每次用户请求都重新建立。

2.良好的游标使用和管理：

减少SQL解析和维护数据库连接同样重要。SQL解析通常包含语法解析、语义分析、语法检查、查询重写和查询优化等阶段，解析产生的执行计划将被存储到shared pool中。SQL解析分为硬解析和软解析：

- 硬解析：硬解析表示一个SQL提交后，在shared pool中没有找到该语句的执行计划缓存，此时数据库将完整地执行SQL解析、分析和检查等操作。硬解析非常消耗资源。
- 软解析：软解析表示一个SQL提交后，在shared pool找到了上次的执行计划缓存，此时数据库将直接使用缓存去执行SQL语句，这样就省去了解析、分析和检查等步骤，提升了性能。

正因为解析次数要尽可能的少，所以开发者需要设计出能够一次解析、多次执行的SQL程序。通常做法是使用绑定参数，来替换SQL中每次执行都会变化的部分，而不是使用变化的字符串拼接SQL。否则，SQL语句很可能只解析一次，而不会被其他用户重用。

例如，在程序中应避免使用字符串常量SQL："SELECT * FROM AREA WHERE AREA_NAME = 'Beijing'"，而应该使用绑定变量的SQL："SELECT * FROM AREA WHERE AREA_NAME = :1"。

数据库配置调优

安装部署YashanDB阶段

在安装部署YashanDB过程中，可以通过使用推荐配置、修改建库参数、CPU资源管理提高数据库性能。

初始化配置参数

YashanDB提供了一系列性能参数，可以满足不同硬件环境、不同业务模型下的性能需求。数据库初始化时，应根据应用需求和环境资源，赋予这些参数合适的初始值；同时，在后续的数据库运行过程中，可能需要对这些参数进行调整，以解决某些被识别到的性能瓶颈。

除了在数据库运行过程中，根据识别到的瓶颈进行参数优化之外，数据库在初始化时，也应该根据应用需求，初始化合适的参数值。

下表列示在部署数据库实例时需要重点关注的配置参数，这些参数的初始化值对数据库性能至关重要：

参数名称	参数描述
VM_BUFFER_SIZE	VM用于保存数据运算的中间结果，例如order by，group by。如对其配置过小，将容易导致频繁的VM换入换出（产生IO）。 建议需要根据应用实际需求，配置合理的VM大小。
REDOFILE_IO_MODE	redo文件的IO模式，不同磁盘在不同IO模式下的性能差异可能比较大，设置合理的IO模式，可以充分发挥磁盘性能。 建议一直使用默认值，若出现明显的性能问题，可通过性能测试找到合适的IO模式。
REDO_BUFFER_SIZE	redo缓存区的大小，设置较大的缓存区可以提高redo刷盘效率和redo回放效率。 建议将REDO_BUFFER_SIZE配置为64M。
REDO_BUFFER_PARTS	redo缓存区的分区个数，某个会话产生的redo被哈希分配到唯一的分区，设置合理的分区个数能减少并发冲突。 建议将REDO_BUFFER_PARTS配置为8。
SCOL_DATA_BUFFER_SIZE	可供使用了稳态数据缓冲区的对象（如LSC表）使用的稳态数据缓冲区大小，缓冲区调大可提升读取命中率及LSC表的导入性能。建议将Slice元数据和频繁访问小表都缓存在内存。 HEAP/TAC表情况下，建议将该参数配置为默认值。 LSC表情况下，该参数配置建议参考下述参数大小章节所描述内容。
SCOL_CACHEABLE_SCAN_ROWS	LSC表扫描使用稳态数据缓存区的最大记录数，如果查询计划评估扫描记录数超过该值，就不会使用稳态数据缓冲区。 建议根据稳态数据缓存区大小，当前业务的缓存效果，以及整体查询性能来调整配置。
DBWR_COUNT	Database Writer用于将数据库产生的脏块刷新到磁盘，此参数用于配置Database Writer的数量，一般情况下（尤其是SSD盘），提高Database Writer的数量，可以提升IO的效率。 建议根据实际硬件环境和业务压力，配置合适的Database Writer数量。
DB_BLOCK_SIZE	组织数据的块大小，包括文件中以及Data Buffer中的块大小，YashanDB提供8K，16K，32K三种不同规格的块大小供设置。 小的数据块可以降低高并发下的数据访问冲突，大的数据块可以提升IO效率。因此建议事务处理场景下配置较小的块大小，数仓场景下配置较大的块大小。 需要注意的是，建库指定的块大小不能再被改变，除非重新建库。
DATA_BUFFER_SIZE	数据缓存区的大小，该值对数据库的性能表现至关重要，合理的缓存区配置将有效降低IO的开销。 建议DATA_BUFFER_SIZE的配置参考下述参数大小章节所描述内容。 该参数设置过小可能会出现启动或建库失败，最小值近似公式：(UNDO_SEGMENTS * 99 + 1K)* DB_BLOCK_SIZE，UNDO_SEGMENTS和DB_BLOCK_SIZE参考CREATE DATABASE，计算时UNDO_SEGMENTS最小取64。
COMPRESSION	LSC表默认压缩方式，如建表时不指定压缩方式，则使用该配置。该值可设置为UNCOMPRESSED不压缩，或LZ4压缩，或ZSTD压缩。 建议使用压缩，可减少存储空间占用，降低IO开销。在压缩率ZSTD优于LZ4；在压缩、解压时间上LZ4优于ZSTD。

参数名称	参数描述
CHECKPOINT_TIMEOUT	检查点时间间隔，表示每过一段时间，执行一次增量检查点。 提高检查点频率能减少redo追尾风险，减少宕机重启时间，但是会增加磁盘IO。反之，则会增加redo追尾风险，增加宕机重启时间。 建议该值设置为10到30秒，但如果是持续时间较短的极限性能测试场景，可以设置为较大值，不触发检查点。
CHECKPOINT_INTERVAL	检查点redo大小阈值，表示当检查点覆盖的redo（重启回放所需的redo）的总大小超过该值时，将执行一次增量检查点。 设置较小的阈值可以提高检查点频率和减少redo追尾风险，减少宕机重启时间。 建议该值设置为redo文件的平均大小，但如果是持续时间较短的极限性能测试场景，可以设置为较大值，不触发检查点。

参数关联性

YashanDB的大部分参数是独立的，但是某些参数之间会有关联性，某些文件的大小也依赖于参数设置。其中：

- REDO_BUFFER_PARTS与参数REDO_BUFFER_SIZE有关联，配置不合理可能会导致数据库启动报错，详细信息请参考配置参数。
- 双写文件的最小大小，依赖于DBWR_COUNT，DBWR_BUFFER_SIZE和DB_BLOCK_SIZE。当这些参数调整变大后，数据库启动可能报双写文件太小的错误，此时需要resize双写文件。
- redo文件的最小大小，依赖于REDO_BUFFER_SIZE，MAX_SESSIONS和DB_BLOCK_SIZE。详细信息请参考ALTER_DATABSE和配置参数。

参数大小

资源相关的参数，不能设置太大，否则会因为无法申请资源而导致数据库启动失败。内存相关的参数的更需要关注，虽然这些参数设置较大对性能有提升，但是系统总内存是有限的，合理分配这些内存才能最大化发挥数据库性能。 其中，DATA_BUFFER_SIZE，SCOL_DATA_BUFFER_SIZE，COLUMNAR_VM_BUFFER_SIZE和VM_BUFFER_SIZE所占内存最多，在不同的业务场景下，这些参数的比重也不一样。此处分别给出单机和分布式部署场景下不同表类型的配置建议：

单机部署场景

1. 如果表类型都是HEAP，建议DATA_BUFFER_SIZE占80%左右，VM_BUFFER_SIZE根据业务类型分配，SCOL_DATA_BUFFER_SIZE和COLUMNAR_VM_BUFFER_SIZE使用默认值。
2. 如果表类型都是TAC，建议DATA_BUFFER_SIZE占60%左右，COLUMNAR_VM_BUFFER_SIZE占30%左右，VM_BUFFER_SIZE适当分配，SCOL_DATA_BUFFER_SIZE使用默认值。
3. 如果表类型都是LSC，建议SCOL_DATA_BUFFER_SIZE占50%左右，DATA_BUFFER_SIZE占10%左右，COLUMNAR_VM_BUFFER_SIZE占25%左右，VM_BUFFER_SIZE适当分配。

分布式部署场景

对于CN节点：

1. 如果表类型都是HEAP，建议VM_BUFFER_SIZE根据业务类型分配，DATA_BUFFER_SIZE、SCOL_DATA_BUFFER_SIZE和COLUMNAR_VM_BUFFER_SIZE使用默认值。
2. 如果表类型都是TAC，建议COLUMNAR_VM_BUFFER_SIZE占70%左右，VM_BUFFER_SIZE适当分配，DATA_BUFFER_SIZE和SCOL_DATA_BUFFER_SIZE使用默认值。
3. 如果表类型都是LSC，建议COLUMNAR_VM_BUFFER_SIZE占75%左右，VM_BUFFER_SIZE适当分配，DATA_BUFFER_SIZE和SCOL_DATA_BUFFER_SIZE使用默认值。

对于DN节点：

1. 如果表类型都是HEAP，建议DATA_BUFFER_SIZE占90%左右，VM_BUFFER_SIZE根据业务类型分配，SCOL_DATA_BUFFER_SIZE和COLUMNAR_VM_BUFFER_SIZE使用默认值。
2. 如果表类型都是TAC，建议DATA_BUFFER_SIZE占50%左右，COLUMNAR_VM_BUFFER_SIZE占35%左右，VM_BUFFER_SIZE适当分配，SCOL_DATA_BUFFER_SIZE使用默认值。
3. 如果表类型都是LSC，建议SCOL_DATA_BUFFER_SIZE占50%左右，COLUMNAR_VM_BUFFER_SIZE占35%左右，DATA_BUFFER_SIZE占5%左右，VM_BUFFER_SIZE适当分配。

对于MN节点，建议DATA_BUFFER_SIZE，SCOL_DATA_BUFFER_SIZE，COLUMNAR_VM_BUFFER_SIZE和VM_BUFFER_SIZE都使用默认值。

以上只是建议配置，并不能让所有业务模型都达到性能最优。可参考DBMS_PARAM自适应生成和应用配置推荐参数。

初始化redo文件

出于性能考虑，YashanDB基于WAL（Write-ahead logging）机制来保证数据持久化，数据库性能也因此与redo文件的大小息息相关。通常情况下， redo文件越大，数据库整体性能越好。当redo文件较小时，需要频繁的触发检查点机制来释放redo空间，会对数据库性能造成冲击。

redo文件是循环复用的，并且旧的redo文件被复用前，该redo对应的脏页必须全部刷盘。如果redo被复用前，对应脏页还没有被Checkpoint刷到磁盘，系统将抛出checkpoint not complete的告警日志，且业务执行会被阻塞，即redo追尾现象。

当业务量激增，redo产生的速度超过Checkpoint的速度时，过一段时间后就会发生redo追尾。通常较大的redo文件能提供更长的追尾缓冲时间。

redo文件不宜过小，否则会造成redo频繁切换。

建议初始创建至少6个redo文件，每个redo文件的大小根据磁盘资源设置，范围在100 MB到100 GB的范围，redo块大小设置为系统支持的IO块大小。对redo文件大小设置的一个经验参考是，数据库正常负载下，自动切换redo文件的时间间隔应该至少为5分钟。

初始化表空间

YashanDB存在不同用途的表空间，数据库性能与每种类型的表空间都息息相关。对于内置表空间，如果在建库SQL中不额外指定其信息，系统将创建默认大小的内置表空间，该值一般远小于正常的应用需求。所以用户在创建数据库时，应该根据数据量、并发数、应用场景等不同因素，为所有内置表空间配置合理的初始大小。

Permanent类型表空间

Permanent表空间用于存储用户数据。在配置表空间中，SYSTEM和SYSAUX表空间一般被用于存储数据库的元数据，对于用户数据建议由单独的用户表空间来存储（使用默认USERS表空间，或者创建新的用户表空间）。

用户应该对数据存储空间有预先规划，存储空间过小或者过大都会带来问题：

- 表空间太小：实际使用的空间大于表空间容量时，可能导致应用报错。通过打开自动扩展可避免报错，但可能导致应用性能的波动。
- 表空间太大：空间浪费严重，有可能导致系统资源耗尽的情况。

在规划数据存储空间时，需要考虑以下几个要素：

- 表空间数量：一般情况下只需要一个表空间即可，但有些场景，例如数据分区，可以选择将数据存储在不同表空间以提升性能和可靠性。
- 表空间挂载的数据文件数量：YashanDB默认单个数据文件的最大大小为512G，当数据量超过这个大小时，就需要考虑增加多个数据文件。
- 数据文件的大小和属性：为了避免运行中的自动扩展，应该尽可能通过初始化值满足应用要求的数据文件大小。在配置自动扩展属性时，应当考虑MAXSIZE在规划的容量范围内，避免出现资源耗尽的情况。

需要注意的是，YashanDB在创建数据文件时，采用了空间预占的策略，初始化的时间可能会比较久，建议同时创建多个表空间，以提升初始化效率。

undo类型表空间

undo类型表空间（即内置表空间中的UNDO表空间）用于存储undo数据，在回滚、一致性读等场景使用。

与用户数据存储不同，undo数据的存储采用循环复用的机制，即超过保留期限（由UNDO_RETENTION参数控制）的undo将有可能被覆盖。这要求用户根据实际场景来配置合理的UNDO表空间大小值，如果配置过小，可能导致出现快照过旧的错误；如果配置过大，可能会造成UNDO表空间的空间浪费。

建议结合UNDO_RETENTION参数值进行UNDO表空间的初始化配置：

- 一般情况下， UNDO_RETENTION越大，UNDO表空间的规划就应该越大。
- UNDO表空间的初始值可以相对小一些（例如1G），并对其配置合理的扩展属性。
- 通过参数UNDO_SHRINK_ENABLED和UNDO_SHRINK_INTERVAL打开undo的自维护机制，降低用户维护UNDO表空间的难度。

Temporary类型表空间

Temporary类型表空间用于必要时持久化临时表数据，对其初始大小的配置取决于应用系统中临时表可能被使用的情况。建议初始值不要配置过大，打开自动扩展即可，除非应用系统有明确的临时表需求。

YashanDB中，临时表的存储空间采用预占方式，但只有在BUFFER不足需要换出时才会被使用，且在数据库重启之后被重置。因此用户可以考虑如下两方面的调优：

- Temporary表空间空间不足时，通过RESIZE或者ADD DATAFILE的方式增加容量。
- Temporary表空间空间过剩时，对其执行SHRINK以释放物理空间。

Swap类型表空间

Swap类型表空间被用作VM BUFFER的物理交换空间。在SORT，GROUP BY，HASH JOIN等场景中VM BUFFER将被消耗，当VM BUFFER不足时，就需要通过Swap表空间进行换入换出。

用户应该结合VM_BUFFER_SIZE参数值和自身业务状况对Swap表空间大小配置合适的初始值。当无法准确评估时，可以先初始化为一个较小的值，并打开自动扩展。

Swap表空间可能因为某些SQL临时性地被撑大，发现此种情况时，用户可以在空间空闲后通过SHRINK功能释放物理空间。

实例级调优

通过初始化调优，包括合理的系统架构和数据库初始化配置后，数据库实例在初始状态可能已经拥有了比较好的性能状态，例如：

- 已经配置了合理的数据库内存。
- 已经考虑了数据库不同模块的I/O要求。
- 已经通过优化操作系统提升了数据库性能。

但在实例运行过程中，仍须一直小心可能导致性能问题的瓶颈，持续监控，及时识别性能相关的问题。

实例性能诊断

性能诊断最有效的方式是建立一个基线，当性能问题暴露时，可以将当前信息与基线进行对比，识别出性能问题。基线数据一般包含：

- 应用统计信息：从应用侧监控到的数据，例如每秒事务数、响应时间等信息。
- 数据库统计信息：从数据库监控到的数据，例如等待事件、系统统计等信息。
- 操作系统统计信息：从操作系统监控到的数据，例如IO、CPU、网络等信息。

实例调优

通常每种性能问题都有很多原因，调优最关键的是要诊断出性能问题的根因，从而采取有效的措施，例如：

- 物理IO慢：通常是因为物理磁盘的性能太差导致，但也有可能是由于未优化的SQL导致产生了很多不必要的IO。
- 锁冲突大：大多数都是因为应用设计不合理导致，只有少数是需要对数据库配置进行修改。
- CPU占用过高：可能是由于CPU资源不足、SQL未优化或者应用低效导致。

关于实例性能诊断和实例调优的详细操作指导请参考实例性能诊断与调优章节介绍。

SQL级调优

典型的SQL调优涉及以下步骤：

1.识别高负载的SQL语句。

用户可以从性能视图、性能报告或日志中，找到耗时多、占用系统资源多的语句。

2.收集与性能相关数据。

包括统计信息和元数据：

- 统计信息：优化器统计信息记录了表数据规模和分布等信息，统计信息至关重要，如果统计信息不存在或过时，优化器将无法生成最佳计划。
- 元数据：收集SQL需要访问的表和视图的结构，以及该语句可能用到的索引定义。

3.确定问题的原因。

通常，引起SQL性能问题的原因包括：

- 执行效率低下的SQL

  如果编写的SQL执行了不必要的操作，那么优化器将无法提升其性能，例如：没有连接条件的连接语句（笛卡尔积）、指定大表做驱动表、指定UNION而不是UNION ALL、使用子查询针对外部查询中的每一行执行等。

- 执行计划不是最优

  优化器会在所有可能的执行路径选择理论上最优的计划，但不意味着优化器一定能选出最优计划。有时，优化器会选择一个访问路径不理想的计划，例如，使用低选择率的谓词条件，可能会在大表上使用全表扫描而不是索引。

- 数据访问结构不合理

  缺少索引和视图未物化是导致SQL性能欠佳的典型原因。最佳的访问结构能够对SQL性能进行指数级的提升。

- 过时的优化器统计信息

  当统计信息维护操作（自动或手动）无法跟上DML导致的表数据更改时，收集到的统计信息可能会过时。由于表上的陈旧统计信息无法准确反映表数据分布，因此优化程序可能会基于错误信息做出决策并生成较差的执行计划。

- 不合理的数据库系统参数

  经过数据库初始化配置调优后的参数在大多数情况下是能够满足用户需要的，但随着环境和数据规模的变化，对配置参数进行合理地调整非常重要。例如SGA的大小影响内存的分配，如果过小将影响到缓存的命中率，增大IO负担，从而影响性能。

- 硬件问题

  性能问题同样跟CPU、内存和I/O有关，对系统资源的配置进行优化将可以提升所有SQL的性能。

4.定义问题的范围。

问题解决方式的范围必须与问题的范围相匹配，用户需要判断问题的范围是在实例级别还是SQL语句级别。例如，shared pool太小，这会引起计划缓存被淘汰，进而导致很多硬解析，此时通过调整配置参数增加shared pool的大小可以解决实例级别的问题，并提高所有会话的性能。但是，如果因为某条SQL语句没能使用有效的索引，而更改整个数据库的优化器相关配置参数，可能会损害整体性能。因此，当单个SQL语句有问题时，应该使用属于语句级别的解决方案处理问题。

5.优化执行性能。

如下这些操作均可以有效地提升SQL执行效率，用户可以根据自身情况使用一种或多种：

- 重写SQL以提高效率
- 使用绑定变量减少硬解析
- 使用等值的连接条件
- 过滤条件中不使用函数
- 将复杂SQL分解为多个简单语句和使用并行等。

此外，可以通过改变元数据对象来提升性能，例如增加索引、调整索引顺序、使用分区表甚至调整数据设计等。

6.防止SQL性能下降。

定期查看性能报告或日志，尽可能保证优化器产生的计划一直最优。

关于SQL级的详细调优指导请参考SQL调优章节介绍。

性能调优特性与工具

有效的信息收集和分析对于性能问题的识别和修复至关重要，YashanDB提供如下特性或者工具帮助用户获取这些信息：

- 统计信息：用于收集当前数据库的一系列基础信息，可以为用户判断当前性能瓶颈提供信息支撑，也可以作为优化器选择最优执行路径的依据。
- 性能报告：用于收集性能相关统计数据并生成便于查看的报告，可通过报告内容分析性能瓶颈并进行对应的调优。
- 性能视图：通过视图的方式向用户呈现数据库内部的性能数据。
- 执行计划：SQL优化过程的重要辅助工具。通过执行计划可以确定优化器是否选出了最优的计划，或者自己增加的索引是否生效。
- AUTOTRACE：可以追踪语句级的性能数据。

此外，用户可以通过top、iostat、netstat等操作系统工具，进行数据库进程级别的性能分析。

统计信息

YashanDB提供统计信息功能，主要用于收集表、索引、列的相关信息。

收集内容

表统计信息

包括分区表、临时表（全局）等所有表对象的基础信息，例如：

- row count： 表的row数量
- block count：表的data block数量
- empty block count：block的rows为0时为empty block
- chaint count：行链接/迁移的数量
- average space：block的平均空闲空间，即block的free size
- average row size：平均row size
- sample size：抽样的row数量
- last analyzed time：统计信息收集时间，每次收集之后更新

索引统计信息

索引的基础信息，例如：

- row count：索引的row数量
- leaf block count：叶子block的数量
- first key distinct count：组合索引第一个col的distinct数量，用于skip scan
- all key distinct count：distinct key的数量
- average leaf blocks per key：每个key对应的leaf block数量，唯一索引为1
- average data blocks per key：每个key对应的data block数量
- btree level：BTree的level
- cluster factor：索引的聚集因子，反映key对应的data block的分布情况，row越有序，factor越小，scan的代价越小
- last analyzed time：统计信息收集时间，每次收集之后更新

列统计信息

包括列的基础信息和列的直方图，例如：

- distinct：列value的唯一值的数量
- null count：列values为null的数量
- low/high value：列的最大最小值
- average column length：列的平均长度
- maximum column length：列的最大长度
- last analyzed time：统计信息收集时间，每次收集之后更新
- density: 密度，用于无直方图情况下优化器估计选择率
- number of buckets：直方图桶的个数
- sample size：抽样的row数量
- 频率直方图：通过endpoint number和endpoint value来表示，两个值满足以下要求：
  - endpoint number：小于等于当前bucket value的累计值。
  - endpoint value：当前bucket中的存放的value。
- 等高直方图：通过endpoint number和endpoint value来表示，假设bucket的数量为m（0号bucket不计算在内），两个值满足以下要求：
  - endpoint number：等于bucket number。0号bucket存放的是最小值，其余bucket编号为1~m，用于存放剩余的值。
  - endpoint value：当前bucket中的存放的所有value的最大值。
- TopN直方图：与频率直方图相似，只是只记录前bucket数量的distinct。按distinct排序，前bucket数量distinct的个数至少要占rows的(1 - 1/n) * 100%。
  - endpoint number：小于等于当前bucket value的累计值。
  - endpoint value：当前bucket中的存放的value。
- 混合直方图：混合直方图是频率直方图和等高直方图的结合，其中endpoint value不会跨多个桶，并且记录endpoint value的个数。
  - endpoint number：小于等于当前bucket value的累计值。
  - endpoint value：桶中的最大值。
  - repeat cnt：endpoint value出现个数。

收集方式

动态收集

优化器在执行计划生成阶段发现统计信息缺失或者无效（missing/stale/insufficient）时，将触发动态收集动作。

通过OPTIMIZER_DYNAMIC_SAMPLING参数可以设置动态收集的级别，每种级别定义了动态收集触发的条件以及采样率。

手工收集

单机部署/共享集群部署中使用DBMS_STATS高级包，分布式部署中使用DBMS_STATS高级包或ANALYZE TABLE语句，让用户对统计信息进行管理和手动触发指定的收集动作。

自动收集

YashanDB内置了定时任务GATHER_STATS_JOB，默认每日凌晨 `2:00` 开始收集全库的统计信息，包括统计信息缺失或者统计信息已经失效的对象的收集。用户可以通过DBMS_SCHEDULER高级包维护该统计信息定时任务。

自动收集统计信息不适用于分布式部署。

通过yasboot进行自动收集

yasboot工具实现了定时收集的功能，单机部署/分布式部署可以借助job命令来实现自动收集的功能。

收集精准化

统计信息自动失效

YashanDB通过统计信息自动失效机制，在表的数据发生大的变化（变化率超过指定阈值），以及表发生一些DDL变化（truncate table/partition、drop partition、add hash partition）时，将该表上已收集的统计信息置为失效，避免优化器使用已过旧的统计信息而生成不合适的执行计划。

数据变化监控

当STATISTICS_LEVEL参数配置为typical或者all时，系统会一直监控表的数据变化，包括表上的DML、truncate表/分区等，通过 DBA_TAB_MODIFICATIONS/USER_TAB_MODIFICATIONS/ALL_TAB_MODIFICATIONS 视图可以查看这些变化。

数据变化持久化

为保证性能，变化监控在内存中进行，DDL变化实时存盘，DML变化通过后台任务定时（15分钟）将变化数据存盘，数据库Shutdown前也会执行变化数据的持久化。本操作通过DBMS_STATS高级包的FLUSH_DATABASE_MONITORING_INFO程序实现，该程序支持用户手动调用执行。

重新收集

通过系统内置job定时重新收集统计信息（全库），或者在需要时手动触发收集，可选择指定对象的收集或者全库收集（在线的全库收集可能会消耗较多的系统资源，运行较长时间）。

重新收集统计信息后，其状态被恢复为未失效，优化器将可以获得真实的统计信息，生成可信的执行计划。

数据变化重置

在以下时间点，上述监控生成的数据变化信息被清空重置：

- 统计信息收集之后
- 表或者分区被删除

统计信息锁定与解锁

大多数情况下，用户均使用自动任务进行批量的统计信息收集，而不是按单个表指定收集，此时存在的一个问题就是，有些如日志类的大表对于调优没有什么价值且很耗时，因此并不希望被一直收集。同时，还可能存在某些表，用户希望在某一时期锁定自己认为最优的统计信息状态，以生成期望的执行计划，并在另一时期解锁让其更新统计信息。为解决这些问题，YashanDB提供统计信息锁定/解锁功能。

通过DBMS_STATS高级包的LOCK/UNLOCK相关存储过程可以实现按用户/表/表分区级别的统计信息锁定/解锁。指定的统计信息被锁定后，所有的统计信息收集相关动作（gather/set/delete等）对其不再生效。

收集结果

用户可以通过下列视图查看统计信息收集的结果数据：

视图名称	说明
DBA_TAB_STATISTICS	包含已收集的所有表的统计信息
DBA_IND_STATISTICS	包含已收集的所有索引的统计信息
DBA_TAB_COL_STATISTICS	包含已收集的所有列的统计信息
DBA_HISTOGRAMS	包含已收集的所有列的直方图
DBA_PART_COL_STATISTICS	包含已收集的所有分区列的统计信息
DBA_PART_HISTOGRAMS	包含已收集的所有分区列的直方图

性能报告

YashanDB在运行过程中，每隔一段时间会将数据库的状态数据保存一份快照到WRM$_SNAPSHOT表中，这些数据包括等待事件、指标数据、空间使用统计、SQL状态信息等数据库关键统计信息。性能报告通过对比两份历史快照来反映整个数据库在一个时间段内的状态，为数据库异常提供事后分析和诊断的依据。

快照

快照默认由系统定时（默认每一个小时，该值可通过DBMS_AWR.MODIFY_SNAPSHOT_SETTINGS修改）创建。

为诊断性能瓶颈，用户可能需要在指定时间点（例如执行某条SQL语句之前和之后）即时地创建快照，用于生成性能报告，此时可以通过调用DBMS_AWR.CREATE_SNAPSHOT程序，手动创建快照。

快照数据超过一定时间（默认七天，该值可通过DBMS_AWR.MODIFY_SNAPSHOT_SETTINGS修改）后，由系统自动进行清理，也可通过DBMS_AWR.DROP_SNAPSHOT_RANGE手动清理。

快照主要信息

即WRM$_SNAPSHOT表内容：

字段名称	含义
SNAP_ID	快照ID，为每次快照的标识，生成报告需以此为关键参数
DBID	数据库ID，同V$DATABASE中的database_id
INSTANCE_NUMBER	实例标识符，同V$INSTANCE中的instance_number
STARTUP_TIME	实例启库时间，数据库连续运行期间的两次快照所生成的报告才是有意义的，此信息有助于在生成报告时对快照ID的有效性进行过滤判断
BEGIN_INTERVAL_TIME	快照开始时间
END_INTERVAL_TIME	快照结束时间
FLUSH_ELAPSED	生成快照时间
SNAP_LEVEL	预留字段
STATUS	预留字段
ERROR_COUNT	预留字段

生成报告

生成报告即通过两个快照的对比，按照不同的维度生成性能报告。报告的运算由存储过程（DBMS_AWR. AWR_REPORT）完成，该存储过程由用户在需要获得报告时主动调用。

```
SET serveroutput ON

--3840305236为数据库ID，1为数据库实例标识符，133起始快照ID,134为结束快照ID
EXEC DBMS_AWR.AWR_REPORT(3840305236,1,133,134);
```

上述程序生成的结果为起始和结束快照区间的性能报告，直接以文本形式输出（需先执行set serveroutput on以打开所在会话端控制输出的开关），可以将其粘贴到html文件进行网页展现和保存。

报告内容

以下为YashanDB所生成性能报告以网页形式展现的示例：

Information

Database Information

当前所在数据库的相关信息。

信息项	含义
DB Name	数据库名称
DB Id	数据库ID
Unique Name	兼容字段，无含义
Role	数据库角色 * PRIMARY：主 * STANDBY：备 * CASCADE STANDBY：级联备
Release	数据库版本号
Instance	兼容字段，无含义
Inst ID	实例标识符
Startup Time	数据库开启时间

Host Infomation

数据库所在服务器的相关信息。

信息项	含义
Host Name	服务器名称
Platform	服务器操作系统平台
CPUs	服务器CPU逻辑个数
Cores	服务器CPU核数
Sockets	服务器CPU物理个数
Memory(GB)	服务器内存大小

Snapshot Infomation

本性能报告所覆盖的快照相关信息。

信息项	含义
Snap Id	起止快照ID
Snap Time	起止快照时间
Sessions	快照时的会话个数
Cursor/Sessions	快照时的平均每个会话游标数
Elapsed	起止快照时间差，即性能报告所覆盖时长
DB Time	起止快照时间内的SQL执行总时间

Load Profile

按每秒、每个事务（DB Time和DB CPU还按每次执行、每个调用）统计的系统平均负载信息。

信息项	含义
DB Time(s)	SQL执行的总时间
DB CPU(s)	SQL执行的CPU耗时
Redo size (bytes)	产生的redo大小
Logical read (blocks)	逻辑读的块数
Block changes	描述数据块的变化（单位块）
Physical read (blocks)	物理读的块数
Physical write (blocks)	物理写的块数
Read IO requests	物理读IO请求次数
Write IO requests	物理写IO请求次数
Read IO (MB)	物理读IO数据量
Write IO (MB)	物理写IO数据量
scan rows	表扫描的行数
User calls	用户级别的调用次数
Parses (SQL)	SQL解析总次数（软解析 + 硬解析）
Hard parses (SQL)	SQL硬解析次数
Logons	当前登录会话数
Executes (SQL)	SQL命令执行次数
Rollbacks	事务回滚次数
Transactions	事务数（回滚数 + 提交数）

Report Summary

Top 10 Foreground Events by Total Wait Time

Top 10（按%DB Time）等待事件信息，对等待事件项的解释见参考手册等待事件。

信息项	含义
Waits	等待次数

信息项	含义
Total Wait Time(sec)	总等待时间
Avg Wait(ms)	平均等待时间
% DB Time	Total Wait Time占DB Time百分比
Wait Class	等待事件所属类别

Wait Classes by Total Wait Time

Top 10等待事件类别信息，与Top 10 Foreground Events by Total Wait Time所列信息项含义一致。Avg Active Sessions项为兼容字段，无含义。

对等待事件类别的解释见数据库性能指标中描述。

SQL Statistics

SQL order by Elapsed Time

按SQL执行所占总时间（Elapsed Time项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by CPU Time

按SQL执行所占CPU耗时（CPU Time项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by User I/O Wait Time

按SQL执行所占User I/O类等待时间（User I/O Time项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by Gets

按SQL执行所占逻辑读（Buffer Gets项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by Reads

按SQL执行所占物理读（Physical Reads项，以字节为单位）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by Executions

按SQL的执行次数（Executions项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by Parse Calls

按SQL的软解析次数（Parse Calls项）排列的TOP SQL信息，信息项的解释见报告内容所述。

SQL order by Sharable Memory

按SQL执行所占共享内存（Sharable Mem项）排列的TOP SQL信息，信息项的解释见报告内容所述。

YAC Statistics

> **Note**：
> YAC Statistics报告仅在共享集群环境下生成。

YAC Summary

共享集群概要信息。

信息项	含义
Number of Instances	共享集群实例数
Number of GCS Tasks	GCS后台线程数量

Global Cache Load Profile

按每秒、每个事务统计的集群全局缓存统计信息。

信息项	含义
Global Cache blocks received	从其他实例接收的页面数量
Global Cache blocks served	发送给其他实例的页面数量
GRC messages received	接收的GRC消息数量
GCS messages received	接收的GCS消息数量
GLS messages received	接收的GLS消息数量
GRC messages sent	发送的GRC消息数量
GCS messages sent	发送的GCS消息数量
GLS messages sent	发送的GLS消息数量
DBWR Fusion writes	融合写入次数

Global Cache Efficiency Percentages

全局缓存访问占比。

信息项	含义
Buffer access - local cache	本地缓存访问占比
Buffer access - remote cache	远端缓存访问占比
Buffer access - disk	磁盘缓存访问占比

Global Cache and Enqueue Services- Workload Characteristics

全局缓存和排队服务的统计信息。

信息项	含义
Avg global cache cr block receive time (us)	从缓存中获取CR页面的平均耗时
Avg global cache current block receive time (us)	从缓存中获取最新页面的平均耗时
Avg global cache current block flush time (us)	数据块平均刷盘耗时

YAC Sys Stats

系统级统计信息中共享集群相关统计项，信息项的解释见统计信息。

性能视图

下表列示YashanDB提供的一系列性能视图，用户可以通过查询这些视图获得数据库内部的一些性能数据：

视图名称	视图说明
V$SYSSTAT	用于查看系统级统计信息，该视图包含了不同组件的信息，是分析数据库实例整体性能瓶颈重要方式。
V$SESSTAT	用于查看会话级的统计信息，可根据此视图分析会话的性能瓶颈。
V$REDOSTAT	用于查看redo刷盘速度，检查点推进速度，redo空闲空间等信息，该视图是分析检查点性能的重要方式。
V$RECOVERY_PROGRESS	用于查看重启回放或备库回放进度，平均速度，剩余时间等信息，该视图是分析备库回放性能的重要方式。
V$SYSTEM_EVENT	用于查看系统各个等待事件的统计信息。
V$SYSTEM_WAIT_CLASS	用于查看系统各个等待类别的统计信息。
V$BUFFER_POOL_STATISTICS	用于分析Data Buffer Pool的使用情况。
V$VMSTAT	用于分析VM Pool的使用情况。
V$SQL	用于查看当前所有的SQL执行统计信息。
V$GLOBAL_MPOOL	用于查看实例级别的appPool，sqlPool，以及dcPool的统计信息。
V$OPEN_CURSOR	用于查看每个statement的相关信息以及使用的appPool情况。
V$PLANCACHE	用于查看检测plan cache的使用情况。

上述所有视图的具体字段含义解释请查阅参考手册动态视图章节。

执行计划

执行计划是SQL优化过程的重要辅助工具，通过执行计划可以确定优化器是否选出了最优的计划，或者自己增加的索引是否生效。

一个执行计划定义了SQL引擎运行一条SQL语句的顺序和算法。可通过EXPLAIN命令或AUTOTRACE命令，在yasql客户端中查询执行计划。

典型的执行计划如下所示：

```
EXPLAIN SELECT branch_no FROM area a, branches b WHERE a.area_no = b.area_no;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3244719443
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|  1 |  NESTED LOOPS INNER            |                      |            |   100000|      690( 0)|
|
|  2 |   TABLE ACCESS FULL            | BRANCHES             | SYS        |   100000|      442( 0)|
|
|  3 |   TABLE ACCESS BY INDEX ROWID  | AREA                 | SYS        |          |             |
|
|* 4 |    INDEX UNIQUE SCAN           | SYS_C_18             | SYS        |         1|      149( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   4 - Predicate : access("A"."AREA_NO" = "B"."AREA_NO")
```

执行计划实际上是一个二叉树结构，执行器以后序遍历的顺序执行计划中定义的算子。上述输出内容则是二叉树后序遍历转换成一维后所显示的样式。例如，本例中算子执行顺序为2->4->3->1->0。

ID

执行步骤的唯一标识，并不是执行顺序。

Operation type

执行算子，前面的空格标示计划的层次关系。没有下层计划的算子，如 2、4，表示将从数据库中检索数据，这些算子就是访问路径或用于检索数据的技术。

ID为2的算子表示使用全表扫描，从BRACHES表里检索所有行。

ID为4的算子表示在AREA表的索引中检索与BRANCHES.AREA_NO相等的ROWID，例如，AREA_ID为'01'的ROWID值是'AADShUAABAAAnPJAAK'。

ID为3的算子表示根据第四层检索出来的ROWID，访问AREA表获取该ROWID对应的数据行。

ID为1的算子，表示对从BRANCHES和AREA表中取出来的数据做连接处理，并将处理结果返回给ID为0的算子。其中连接算法使用的是NESTED LOOP JOIN。

分布式部署中的执行计划比单机中会多出PX并行执行算子，用于跨节点间的数据发送和接收，例如本条语句在分布式中的执行算子信息如下：

```
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
```

```
|
+----+-------------------------------+--------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT              |                    |            |          |             |
|
|  1 |  DISTRIBUTED COORDINATOR      |                    |            |          |             |
|
|  2 |   COL TO ROW                  |                    |            |          |             |
|
|  3 |    PX N2I REMOTE              | QUEUE_0            |            |   100000|    1943( 0)|
|
|* 4 |     HASH JOIN INNER           |                    |            |   100000|    1604( 0)|
|
|* 5 |      PX N2N REMOTE            | QUEUE_1            |            |   100000|     606( 0)|
|
|  6 |       TABLE ACCESS FULL       | AREA               | SALES      |   100000|     442( 0)|
|
|* 7 |      PX N2N REMOTE            | QUEUE_2            |            |   100000|     606( 0)|
|
|  8 |       TABLE ACCESS FULL       | BRANCHES           | SALES      |   100000|     442( 0)|
|
+----+-------------------------------+--------------------+------------+----------+-------------+-------------------------
-------+
```

其中，'PX N2I REMOTE'表示多个DN上的数据向CN聚集，'PX N2N REMOTE'表示将DN上的数据进行哈希重分布。

Rows

优化器根据统计信息和特定算法计算出来的行数预估值，通常并不能精准体现最终执行计算出来的行数信息。

Cost

优化器根据算子和硬件等信息，算出的一个参考值，Cost值越大，表示该层计划占用的资源越大。

Partition info

算子扫描到的分区范围。

AUTOTRACE

AUTOTRACE的工作方式类似于EXPLAIN，区别在于AUTOTRACE将执行SQL语句，并显示执行结果，每层执行算子的统计信息，以及整条SQL语句所产生的统计信息（后两项可通过AUTOTRACE选项控制是否显示）。通过启动AUTOTRACE，用户能够精准定位到一条SQL执行过程中每层算子消耗的时间，查询/计算出来的行数等信息。如果预估值和实际值有严重偏差，则需要重新收集统计信息。

> **Note**：
> 统计信息显示是否有内容依赖于系统是否已开启统计信息收集开关（STATISTICS_LEVEL参数设置为ALL）。
> 当STATISTICS_LEVEL参数设置为TYPICAL或ALL时会统计表的变化情况，同时也会收集每个计划算子的执行时间、执行次数等信息。
> STATISTICS_LEVEL参数设置为ALL将对系统造成一定性能损失，请确保只在当前SESSION设置该参数。

典型的AUTOTRACE输出如下：

```
ALTER SESSION SET statistics_level=ALL;


SET autotrace ON;
SELECT * FROM area;
AREA_NO AREA_NAME         DHQ
------- ----------------- --------------------
01      华东              Shanghai
02      华西              Chengdu
03      华南              Guangzhou
04      华北              Beijing
05      华中              Wuhan


Execution Plan
----------------------------------------------------------
SQL hash value: 4040703571
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+--------------------------------+
| Id | Operation type                 | Name                 | Owner      | E - Rows | A - Rows | Cost(%CPU)  | A - Time |
Loops    | Memory   | Disk     | Partition info                 |
+----+--------------------------------+----------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+--------------------------------+
|  0 | SELECT STATEMENT               |                      |            |          |        5|             |        29|
5|          |          |                                |
|  1 |  TABLE ACCESS FULL             | AREA                 | SYS        |    100000|        5|      442( 0)|        20|
5|          |          |                                |
+----+--------------------------------+----------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+--------------------------------+


Statistics
----------------------------------------------------------
                  0 physical reads
                  6 db block gets
                  0 consistent gets
                  0 redo size
                  0 recursive calls
                  0 bytes sent via SQL*Net to client
                  0 bytes received via SQL*Net from client
                  0 SQL*Net roundtrips to/from client
                  0 sorts (memory)
                  0 sorts (disk)
                  5 rows processed
                359 bytes sent via PX
```

需注意的是，开启AUTOTRACE时的执行计划输出将比关闭时多一些信息，如下：

- **E - Rows**：CBO预估行数。

- **A - Rows**：实际执行行数。
- **A - Time**：算子实际执行时间。
- **Loops**：被调用次数。
- **Memory**：内存资源占用。
- **Disk**：磁盘资源占用。
- **recursive calls**：递归调用次数。每次执行SQL，为此SQL产生的额外SQL调用次数。
- **db block gets**：本次SQL请求的读取数据块的数量。
- **consistent gets**：一致性读的数量。
- **physical reads**：从硬盘读取的数据块数量。
- **redo size**：redo产生的字节数。
- **rows processed**：SQL查询/影响的行数。
- **bytes sent via PX**：网络上PX发送的数据量。

其中，physical reads和Disk的数值应处在较低水平，如果产生了大量的物理读和硬盘资源消耗，很有可能是内存不足引起，需要调整数据库的内存参数，或调整服务器物理内存。

如果预估的行数和实际行数偏差较大，而且收集统计信息后性能仍未改善，可使用HINT手动调整执行计划。

实例性能诊断与调优

实例级别的调优重点在于通过一些手段实现对数据库整体的性能诊断，和根据诊断结果进行分析，并采取具体的优化措施。

这些手段包括定时或手动地运用工具获得性能相关信息，或者通过视图查询YashanDB提供的一系列性能指标数据，对获得的结果进行经验分析或者与可量化的基线对比判断，对已发生的性能问题进行根因诊断，和对可能发生的性能问题进行预判。

数据库性能指标

数据库的性能需要通过一定的指标来反映。除应用本身的指标外，YashanDB还提供一系列用于衡量性能的指标。这些指标主要包含数据库负载以及各种资源的使用情况，理解这些指标可以帮助诊断性能瓶颈，并可以将其作为调优效果的衡量方法。

YashanDB提供分四种不同层次和维度的统计指标：

- 系统级统计：从数据库实例的角度，提供分析整体性能的指标，包括不同组件和模块的统计信息。
- 会话级统计：由于不同会话的性能瓶颈存在差异，需要从会话的角度，提供分析每个会话性能的指标。
- 语句级统计：从单条SQL语句的角度，提供分析SQL语句性能的指标。
- 等待事件统计：从因等待某些资源而导致阻塞的角度，提供分析性能的指标。

系统级统计

系统级性能指标从时间或者次数的角度统计了YashanDB运行过程中的一些关键内部活动信息，每个统计项在不同程度反映数据库某方面的性能表现，用户往往需要结合多种指标项来确定性能问题的根因，具体每个统计项的含义请查阅参考手册统计信息章节。

用户可以通过V$SYSSTAT视图查询各统计项的值，同时可以看到统计项根据功能被归属的分类：

类别编号	类别名称	说明
1	USER	用户执行相关的统计信息项，例如事务提交数量、SQL请求数量等。
2	REDO	redo日志相关的统计信息项，反映系统redo资源的使用情况。
8	CACHE	数据库内部各类CACHE相关的统计信息项。
64	SQL	SQL执行相关的统计信息项，例如不同DML执行次数、排序次数等。
128	DEBUG	用于内部DEBUG的统计信息项，用户一般不需要关注。

会话级统计

会话级的统计信息可以通过V$SESSTAT视图获取（该视图只提供统计项的ID，具体名称需要结合V$STATNAME视图获取），如果只关注当前会话的统计信息，可以通过V$MYSTAT视图获取。

会话级的统计项和分类与系统级的统计项类似。

语句级统计

语句级统计为与SQL相关的统计信息，可以通过V$SQL视图获取，需关注项如下：

字段	类型	说明
SHARABLE_MEM	INTEGER	在SQL Pool中占用的共享内存大小（单位：Byte）
FETCHES	BIGINT	SQL语句的fetch数
EXECUTIONS	BIGINT	被载入缓存库后的执行次数
ROWS_PROCESSED	BIGINT	SQL语句返回的总列数
PARSE_CALLS	BIGINT	解析调用次数
DISK_READS	BIGINT	读磁盘次数
BUFFER_GETS	BIGINT	读缓存区次数
PHYSICAL_READ_REQUESTS	BIGINT	SQL发出的物理读取I/O请求数
PHYSICAL_READ_BYTES	BIGINT	SQL读磁盘的字节数
PHYSICAL_WRITE_REQUESTS	BIGINT	SQL发出的物理写入I/O请求数

字段	类型	说明
PHYSICAL_WRITE_BYTES	BIGINT	SQL写入磁盘的字节数
USER_IO_WAIT_TIME	BIGINT	用户I/O等待时间（单位：微秒）
PLSQL_EXEC_TIME	BIGINT	PL执行时间（单位：微秒）
CPU_TIME	BIGINT	解析/执行/取得等CPU使用时间（单位：微秒）
ELAPSED_TIME	BIGINT	解析/执行/取得等消耗时间（单位：微秒）

等待事件统计

在数据库的运行过程中，其内部可能因为某些关键活动（例如I/O）产生等待，从而产生一次事件，称为等待事件。对等待事件进行统计得到的信息可以反馈数据库的性能瓶颈，例如锁竞争的等待事件较多时，就需要考虑锁是否成为数据库瓶颈。

YashanDB将等待事件分为以下类别，每种等待类别下包含一种或者多种具体的等待事件：

- Application：由应用产生的等待，例如不同应用层事务产生的行锁等待。
- Concurrency：数据库内部资源产生的等待，例如latch的竞争等待。
- Commit：事务提交时，redo日志同步产生的等待。
- User I/O：用户线程产生的I/O，例如读取数据块产生的I/O等待。
- System I/O：数据库后台线程I/O导致的等待，例如Database Writer线程将脏数据块同步到磁盘导致的I/O等待。
- Other：其他类型的等待。
- Idle：数据库等待客户端消息，此类等待事件表示会话此时处于空闲状态。
- Network：因网络传输产生的等待，例如数据在网络传输期间产生的等待。
- Configuration：因数据库实例配置产生的等待，例如配置redo大小、undo大小、DATA BUFFER大小时产生的等待。
- Cluster：集群等待事件，例如跨节点交互等待应答。
- Distributed：分布式等待事件，例如CN等待各节点回应。

每类等待事件代表了一类问题，并对应特定的优化方式。用户可以通过V$SYSTEM_WAIT_CLASS视图获取不同类别的等待事件统计信息，且该统计信息区分了前台线程和后台线程。一般情况下，性能问题主要反映在前台线程的统计信息中。

V$SYSTEM_EVENT视图提供具体的等待事件的统计信息，包括等待的次数、等待的时间等维度。关于YashanDB中所有等待事件的含义和分析方式描述请查阅参考手册等待事件章节。

数据库性能分析

性能报告分析

YashanDB提供给用户的性能报告主要包含以下内容：

- 数据库信息
- 服务器信息
- 快照信息
- 负载信息
- TOP10等待事件
- wait class信息
- 按照执行时间、CPU时间、I/O等待时间、执行次数、解析次数等排序的SQL信息

用户可以从如下方面对上述内容进行分析：

- 查找慢SQL：当系统执行SQL较慢时，可以查看SQL order by Elapsed Time，查看最耗时SQL。
- 查找两快照之间时间损耗：通过查找TOP10等待时间，可以查看各事件花费时间。
- 查找快照间内存以及IO等的读写大小：通过查找负载信息，可以看到对应信息。
- 查看SQL复用率：查找负载信息"parses(SQL)"跟"Hard parses(SQL)"，差值越大，复用率越高，效率也就越高。

统计项分析

用户可以通过YashanDB提供的各项性能指标统计项来分析系统存在的性能瓶颈，同时，可以在调优之后，通过观察统计项的变化来验证执行的优化是否有效。

在YashanDB中，每个统计项反映的是具体某个组件的一项指标，对指标的细化有助于用户判断性能问题的根因，因为通常情况下，某个性能问题对外的表现极大可能是多个统计指标出现异常。

以常见的IO问题为例，系统IO负载增大通常情况下是由于业务压力升高或者后台线程工作导致，可以通过观察不同统计项的变化来识别相应的问题，从而进行针对性的优化：

- 如果发现DISK READS以及DISK READ TIME增长很快，说明数据库正在频繁地读数据块。导致这种现象的主要原因是业务短时间访问了大量的数据，或者DATA_BUFFER_SIZE太小，导致缓存区发生频繁地淘汰。可以尝试通过增大DATA_BUFFER_SIZE，或者优化不必要的数据扫描（例如将全表扫描优化为索引扫描），来降低IO压力。
- 如果发现VM SWAP OUT增长很快，说明数据库正在产生大量的计算中间结果（例如排序操作），而配置的VM_BUFFER_SIZE较小，导致了频繁地换入换出。可以尝试通过增大VM_BUFFER_SIZE来缓解。
- 如果发现CHECKPOINTS COMPLETED增长很快，说明这是数据库后台清理脏块带来的IO增长，通常情况下不会对业务产生影响。
- 如果发现REDO SYNCH WRITE增长很快，说明业务此时事务提交的频率很高，redo日志刷盘带来了IO的增长，这是正常的现象。但是如果发现此种情况导致IO负载过高而且业务性能出现明显下降时，就需要分析是否存储介质的性能出现了瓶颈。

关于YashanDB中所有统计项的具体描述信息请查阅参考手册统计信息章节。

系统资源调优

实例调优的目的是减少执行所需的资源消耗以及运行时间，提高特定资源的利用率。通常情况下，实例级性能问题都是因为某些资源的过度使用而导致这些资源成了系统中的瓶颈。

经过分析诊断后，用户可能发现很多时候性能问题的根因来自于系统的资源能力，或者通过提升系统资源能力就可以很好地解决某些性能问题。数据库性能与操作系统的资源配置息息相关，例如内存、I/O、网络等。针对这些资源的调优，对提升数据库性能有很大的帮助。

I/O性能调优

I/O性能对数据库性能至关重要，例如redo数据刷盘、脏块刷盘都直接依赖底层磁盘的I/O性能。在部署数据库时，应当选择满足I/O性能要求的介质作为数据库底层的存储。

通常情况下，磁盘的I/O性能评价指标主要包括以下几个方面：

- 吞吐量：每秒的I/O请求大小
- IOPS：每秒的I/O请求数
- 响应时间：I/O请求从发出到收到响应的时间间隔
- 使用率：磁盘处理I/O的时间百分比

> **Note**：
> 上述I/O性能评价指标均可以通过I/O工具（例如IOSTAT）获取。

由于数据库的I/O模型有自身明确的特征，用户需要进一步依据这些特征来判断磁盘是否满足数据库应用要求，主要考虑以下几个方面：

- 离散I/O的能力：数据块的读取和刷盘都是离散I/O，如果磁盘离散I/O性能差将严重影响数据库的性能。
- 不同I/O大小的能力：YashanDB的默认数据块大小为8K，redo的I/O大小一般为1M，其它如归档、备份等均有不同的I/O大小设置。
- I/O并行能力：数据库I/O模型为读写混合模型，因此对磁盘的混合I/O能力要求较高。

建议用户在部署数据库之前，通过I/O测试工具（例如FIO）对这些场景的I/O进行验证，以保证数据库运行过程中不会出现明显的I/O瓶颈。

当数据库在运行过程中出现明显的I/O瓶颈，且并无提升资源能力的计划时，需要考虑进行I/O性能调优，一些典型的措施列举如下：

- redo文件和DATA文件分盘部署，以减少I/O竞争。
- 通过DATAFILE_IO_MODE和REDOFILE_IO_MODE改变数据库I/O行为，以满足运行的环境。
- 调整DBWR系列参数，以提升I/O效率。
- 增大VM BUFFER，以减少SWAP。

CPU性能调优

CPU的性能同样会影响数据库的性能表现。在部署数据库时，一般情况下用户需要关注以下可能影响数据库性能的CPU指标：

- CPU核数：决定了系统的并发处理能力，同样也决定了数据库的并发处理能力，同时还需要考虑系统中其他应用的并行。
- CPU主频：决定了CPU的执行能力，同样也将影响数据库的执行能力。
- CACHE：YashanDB内部有很多针对CACHE的优化机制，因此较大的CACHE可以带来很好的性能收益。
- NUMA：跨NUMA节点访问内存的延迟很高，对数据库这种访存模型很不友好，因此建议尽量避免出现较多的NUMA节点。

> **Note**：
> 1.上述CPU性能评价指标均可以通过lscpu等操作系统工具获取。
> 2.建议数据库并行处理数一般不超过CPU核数的2倍以上，否则频繁的上下文切换将严重影响数据库的性能。

当数据库在运行过程中出现CPU瓶颈（例如系统LOAD过高）时，通常需要考虑以下几个方面的优化：

- 数据库应用优化：对SQL进行调优，选择更好的执行计划，降低资源开销。
- 数据库限流：通过改变线程池的配置，对数据库进行流程控制。
- 其他应用优化：CPU属于公共资源，因此需要考虑对除了数据库之外的应用进行优化。
- 硬件升级：如果当前硬件满足不了应用诉求，需要考虑对硬件进行升级。

内存性能调优

内存是数据库重要的运行资源之一，需要考虑以下几个方面的优化：

- 内存大小：为数据库配置合理的内存大小，例如DATA BUFFER POOL，这些内存对数据库整体性能至关重要。
- 内存SWAP：建议关闭SWAP区，防止内存SWAP造成的性能抖动，YashanDB提供了使用操作系统大页内存的方式，可以通过大页消除SWAP的带来的性能抖动。
- 防止OOM：建议将系统内存申请的over_commit关闭，防止数据库运行过程中出现意外的OOM。

在诊断内存问题时，用户还可以借助一些系统工具，例如虚拟内存管理带来的Page Fault会严重影响数据库的访存效率，如果发现数据库有过多的Page Fault，那么就应该及时对内存进行优化。

SQL调优

SQL级别的调优重点在于通过各种手段提高SQL语句的性能，以达到特定、可衡量的性能标准。

通常情况下，用户希望通过SQL调优能达到如下两个目标：

- 减少SQL执行的响应时间，即减少从用户发送执行命令到收到数据回复之间的时间。
- 提升吞吐量，即在正常负载情况下，可以使用更少的资源处理语句的数据需求。

这需要从开发人员编写SQL和数据库运行SQL两方面都进行分析和必要时地调优，尽可能达到理想的性能目标。

本手册将SQL调优入门知识、SQL调优原理与规则章节对SQL调优进行从浅至深地阐述，以期让用户了解YashanDB的优化器工作原理和执行流程。同时，进行SQL调优还需要用户对自身业务逻辑和数据库环境非常了解，多方面结合的分析才能保证达到更好和更合适的调优效果。

SQL调优入门知识

SQL调优入门知识包括以下几部分：

1. SQL的执行流程。
2. YashanDB的优化器。
3. 执行计划。
4. YashanDB所有的执行算子。

上述知识将为继续了解YashanDB的SQL调优原理打下基础。

SQL执行流程

在实际了解调优之前，需要大致了解SQL语句的执行流程，知晓SQL语句如何从输入的字符串一步步转换成最终的执行结果。

从客户端输入SQL语句到向客户端输出结果，SQL引擎将依次进行解析、优化、执行等多个步骤处理。

Parse&Verify 解析与校验

SQL执行的第一阶段解析与校验。其中解析分为软解析和硬解析两种方式：

1. 根据客户端输入SQL语句的HASH值，在shared pool的计划缓存中查看是否有副本，有副本则重用查询计划，直接跳到执行阶段，即“软解析”。
2. 否则进行硬解析，对SQL进行分词，生成系统能够识别的抽象语法树，并对语法树进行校验：
    1. 校验Table，查看Table是否能正常访问到，以及校验权限等问题。
    2. 校验Filter过滤条件，包括过滤条件里面的column信息是否能在表中找到等。
    3. 校验Columns查看每个column的节点，查看函数是否是内置函数或者外部声明函数，以及每个节点里面带着的filter情况。

> **Note**：
> SQL引擎始终对DDL执行硬解析。

Rewrite 语句重写

在解析与校验完成之后，SQL引擎会根据特定关系代数规则，将抽象语法树改写为另外一种等价的逻辑描述树，使之更容易被优化。

重写对于非常复杂的查询（包括那些包含许多子查询或许多连接的查询）而言特别重要，查询生成器工具通常会创建这些很复杂的查询。 YashanDB内部存在多种规则的语句重写，大体包括如下几类：

- 子查询改写
- 谓词下推
- 谓词组合优化

关于改写和谓词相关的优化的详细介绍见查询改写与谓词优化。

Optimize 优化

在Optimize阶段会生成逻辑计划，并根据查询中的引用对象和条件，在考虑多方面因素后确定最有效的执行计划，如确定Join Orders，选择率计算等，之后再根据相应规则转化成物理计划。

生成执行计划是处理任何SQL的必要步骤，将在很大程度上影响执行时间。

Execute 执行

SQL引擎器使用一系列的算法组织和处理数据，比较典型的有排序、连接、表扫描、汇聚等，这些算法成为执行算子。优化器在Optimize阶段搭积木般组合各种算子，最后生成执行计划，执行器则根据计划逐步执行算子，最后生成数据。

优化器

优化器是SQL引擎的核心，在了解SQL调优之前，应该对优化器进行了解。

优化器的目的是尝试为SQL生成最佳执行计划，YashanDB默认的优化器为CBO，其核心规则为在所有可能的备选执行路径中选择成本最低的计划。

SQL是一种非过程式语言，因此优化器可以自由的以任何顺序进行合并、重组和处理SQL语句，并通过检查多种访问方法（例如全表扫描或索引扫描），不同的联接方法（例如嵌套循环或哈希联接），不同的联接顺序以及可能的转换，来确定SQL语句的最佳计划。

对于给定的SQL查询和硬件环境，优化器为每个可能的计划步骤分配一个相对数值的成本，然后将这些值一起分解以生成该计划的总体成本估算。在计算出替代计划的成本之后，优化器将选择成本估算最低的计划。成本的计算基于统计信息和关联条件，同时，在特定环境中，成本计算同时会考量CPU、IO和网络通讯。

统计信息对于CBO是否能做出正确的选择至关重要，因此，当优化器发现统计信息不合理/不完整/不可用时，将会触发动态的统计信息收集，覆盖旧的统计信息。而在某些特殊情况下，用户通过性能诊断发现优化器未进行统计信息更新，可以进行统计信息的手工收集。

作为影响性能的关键组件，优化器永远致力于生成最优的执行计划，但在各项因素（例如元数据变化、数据量变化等）交织的复杂环境中，优化器的判断是可能出错的。当发现数据库环境正常，而优化器生成的执行计划并不是最优和想要的计划时，就可能需要人工介入进行调优。

执行计划

SQL与计划

执行计划是整个SQL语句执行过程的描述。一个执行计划的优秀与否，很大程度的决定了SQL执行的效率。

也正是因为一条SQL语句可以由不同的计划来进行执行，SQL调优才有了意义。

下面将结合例子来讲解，计划与调优具体之间的相关联系，阐明计划如何影响性能。

计划与算子

示例

```
EXPLAIN SELECT * FROM area WHERE area_no = 1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3728302104
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  TABLE ACCESS FULL             | AREA                 | SALES      |         1|       13( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : filter("AREA"."AREA_NO" = 1)
```

对于需要查询的table来说，能够有多种不同的方式去进行扫描（获取表中数据）。这些不同的扫描方式就是不同的扫描算子。 因为这些不同的算子存在，才能使得一条SQL存在不同的执行计划。对于不同的SQL来说，最适合、效率最高的算子可能都是不相同的。 SQL调优的终极目标，就是在这些不同的执行计划中找到效率最高的一个。

回到上述例子，table表在简单进行扫描时。选择何种算子最优，也是不能一概而论的。 一种很流行的错误观点是无论任何扫描场景都选择索引扫描，这是因为没有深层理解算子的实际执行过程所导致的。 例如在需要回表时，采用的就是table full scan，而不需要回表时，采用index scan则效率能够更高。 因此在进行调优之前，有必要对不同算子进行深入了解。

算子效率评估

不同算子的效率既然不同，那就需要一套标准来评估各个算子性能的高低。这一套标准就是执行计划中展示出来的cost（执行代价）。

示例

```
EXPLAIN SELECT area_no FROM area WHERE area_no = 1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 1386453710
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
```

```
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT             |                     |            |          |             |
|
|* 1 |  INDEX FAST FULL SCAN        | SYS_C_33            | SALES      |        1|        6( 0)|
|
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : filter("AREA"."AREA_NO" = 1)
```

可以看到，正如上小节所说，在不需要回表时，走index scan的代价小于table full scan。 一套能够反应算子真实代价的model，是生成高效计划的关键。但cost model也有失效时，在一些较为复杂的语句上。cost model可能会表现的不够准确。 这时就需要人工介入去判断是否有某些算子的cost评估不准确。从而进行相应调整，规避不优秀的计划。 例如使用hint的方式强行指定算子。

算子组合

计划往往不只是上述例子中所示，仅有少少的一层。当SQL变得复杂时，不仅需要考虑不同的算子，还要考虑如何选择多个算子的组合。

示例

```
EXPLAIN SELECT * FROM area a, branches b WHERE a.area_no = b.area_no;

PLAN_DESCRIPTION
------------------------------------------------------------
SQL hash value: 1557742453
Optimizer: ADOPT_C

+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type               | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT             |                     |            |          |             |
|
|  1 |  NESTED LOOPS INNER          |                     |            |    100000|      211( 0)|
|
|  2 |   TABLE ACCESS FULL          | BRANCHES            | SALES      |    100000|      132( 0)|
|
|  3 |   TABLE ACCESS BY INDEX ROWID  | AREA                | SALES      |          |             |
|
|* 4 |    INDEX UNIQUE SCAN         | SYS_C_33            | SALES      |         1|       13( 0)|
|
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   4 - Predicate : access("A"."AREA_NO" = "B"."AREA_NO")
```

一个最好的例子就是Join。对于Join来说，仅仅考虑下层的scan算子是不够的。还需要考虑不同表是放在Join算子的左侧还是右侧。 对于hash join来说还需要将小表放在右边（build表），使得建表的成本降低。这就是一种算子的组合方式。 如何在不同算子组合中选择最优的一种和选择最优的算子一样至关重要。

选出最优的组合也是调优中最困难的部分，具体的调优规则可见：SQL调优原理与规则

执行算子

数据查询算子

算子名称	含义	适用行模式	适用列模式
SUBQUERY	子查询标识，表示存在行子查询执行计划，列子查询会转为result计划。	√	√
VIEW	视图标识，当行表同层查询存在from子查询或者用户视图时需要增加。	√	×
CONNECT BY	层次递归查询，对于某一条记录，以深度优先返回与当前记录通过connect by条件关联的所有记录，且返回的记录集依据prior id=father_id这个条件，形成的是一个包含当前的记录和以当前记录id为根节点的树状结构数据。	√	×
SELECT STATEMENT	查询语句标识。	√	√
INSERT STATEMENT	插入语句标识	√	√
UPDATE STATEMENT	数据更新语句标识。	√	√
DELETE STATEMENT	数据删除语句标识。	√	√
MERGE STATEMENT	数据批量插入语句标识，满足条件时插入语句，在一次全表扫描中可以完成所有INSERT。	√	√
UNION ALL	将多个SELECT的结果集合并返回。	√	√
AGGREGATE	对未分组的数据执行聚集操作。	√	√
FOR UPDATE	查询某行数据的同时加行级锁，防止其他用户更新此条数据。	√	√
RESULT	谓词重组后，计划树上需要承载FILTER和投影的独立计划节点	√	√
WINDOW	limit数据筛选。用于限定结果集返回行数，具有略过offset的能力。	√	√
WINDOW SORT	窗口函数标识，可以同时进行分组和排序，基于排序做分组，且会返回原始信息而非分组后（一组只有一条）的信息。	√	√
WINDOW HASH	窗口函数标识，可以同时进行分组和排序，基于哈希做分组，且会返回原始信息而非分组后（一组只有一条）的信息。	×	√
WINDOW NOSORT	窗口函数标识，该算子对有序数据进行分组并计算窗口函数返回值，且会返回原始信息而非分组后（一组只有一条）的信息。如果下层算子提供的数据顺序已经满足窗口函数所需的排序要求，则可以选用该算子。	√	√
FIRST ROW	获取下层扫描算子得到的第一行记录。	√	√
COUNT	返回表中选取的行数。	√	√
COUNT STOPKEY	使用ROWNUM作为限定条件返回表中选取的行数。	√	√

表扫描算子

算子名称	含义	适用行模式	适用列模式
TABLE ACCESS FULL	全表扫描。	√	√

算子名称	含义	适用行模式	适用列模式
TABLE ACCESS BY INDEX ROWID	根据索引查找到对应数据的block的rowid（根据rowid回表）。	√	√
TABLE ACCESS BY USER ROWID	指定rowid进行数据block的查找。	√	√

索引扫描算子

算子名称	含义	适用行模式	适用列模式
INDEX UNIQUE SCAN	唯一索引扫描，仅仅适用于where条件是等值查询的SQL。	√	√
INDEX RANGE SCAN DESCENDING	索引范围降序扫描 。与索引范围扫描区别在与，索引查找方向为降序（从大到小）。当扫描对象是唯一索引是，谓词条件必须是范围查询（between、<、>）；扫描对象是非唯一性索引，则没有限制。索引范围降序扫描可能返回多条记录。	√	√
INDEX RANGE SCAN	索引范围扫描。当扫描对象是唯一索引是，谓词条件必须是范围查询（between、<、>）；扫描对象是非唯一性索引是，则没有限制。索引范围扫描可能返回多条记录。	√	√
INDEX FULL SCAN DESCENDING	按降序方式进行索引全扫描。	√	√
INDEX FULL SCAN	索引全扫描。扫描目标索引所有叶子块的所有索引行。	√	√
INDEX FAST FULL SCAN	索引快速全扫描。类似索引全扫描，但扫描结果不是有序的。因为是物理读，不是逻辑读索引，同时可以并行读索引。	√	√
INDEX FULL SCAN (MIN/MAX)	索引全扫描取最小/最大。不分组情形下，针对索引字段的min、max函数优化，只返回一条记录。	√	√
INDEX RANGE SCAN (MIN/MAX)	索引范围扫描取最小、最大。不分组情形下，针对索引字段的min、max函数优化，只返回一条记录。	√	√
SPATIAL INDEX SCAN	RTREE索引扫描，仅适用于索引字段的ST_Contains、ST_Intersects等空间关系函数的优化。	√	×

分区扫描算子

算子名称	含义	适用行模式	适用列模式
PART SCAN ITERATOR	一组分区扫描。	√	√
PART SCAN ALL	所有分区扫描。	√	√
PART SCAN SINGLE	单个分区扫描。	√	√
PART SCAN COMBINED ITERATOR	指定二级分区扫描。	√	√

AC扫描算子

算子名称	含义	适用行模式	适用列模式

算子名称	含义	适用行模式	适用列模式
AC SCAN	AC扫描。	×	√
EXPAND	展开AC扫描的数据。	×	√

分组/排序算子

算子名称	含义	适用行模式	适用列模式
SORT	据实际情况生成的排序计划。	√	√
TOP SORT	据实际情况生成前TOP个数据的排序计划。	√	√
SORT ORDER BY	order by语句产生的排序计划。	√	√
HASH DISTINCT	使用hash算法对数据进行除重。	√	√
SORT DISTINCT	使用排序算法对数据进行除重。	√	√
SORTED DISTINCT	使用排序算法对已排序的数据进行除重。	√	√
TOP SORT DISTINCT	使用排序算法对已排序的数据进行除重，并返回前TOP个。	√	√
HASH GROUP	使用hash算法对数据进行分组。	√	√
SORT GROUP	使用排序算法对数据进行分组。	√	×
TOP SORT GROUP	使用排序算法对数据进行分组，并返回前TOP个。	√	×
GROUP	对已排好序或者分组的数据进行分组。	√	√
SORT GROUPING SETS	使用排序算法，按照多个列进行分组，并对结果进行union操作。	×	√
HASH GROUPING SETS	使用hash算法，按照多个列进行分组，并对结果进行union操作。	×	√

HASH JOIN算子

算子名称	含义	适用行模式	适用列模式
HASH JOIN OUTER	HASH JOIN左外连接。以左表为准，当左右表满足join指定条件则同时返回左右表数据，如果满足不了，则将左表数据+右表补空返回。	√	√
HASH JOIN FULL OUTER	HASH JOIN全外连接。当左右表满足join指定条件则同时返回左右表数据，如果满足不了，则将左表数据+右表补空、左表补空+右表数据返回。	√	√
HASH JOIN SEMI	HASH JOIN左半连接。如果左表中满足指定条件的某行数据在右表中出现过，则此行保留在结果集中。Filter in / exists常常改造成SEMI JOIN。	√	√
HASH JOIN ANTI	HASH JOIN左反半连接。如果左表中满足指定条件的某行数据没有在右表中出现过，则此行数据保留在结果集中。Filter not in / not exists常常改造成ANTI JOIN。	√	√
HASH JOIN	HASH JOIN内连接。将左表物化，右表通过HASH算法从左表查找满足join条件的数据。仅支持等值查询。	√	√
HASH RIGHT OUTER	HASH JOIN右外连接。以右表为准，当左右表满足join指定条件则同时返回左右表数据，如果满足不了，则将右表数据+左表补空返回。	√	√
HASH RIGHT SEMI	HASH JOIN右半连接。如果右表中满足指定条件的某行数据在左表中出现过，则此行保留在结果集中。	√	√
HASH RIGHT ANTI	HASH JOIN右反半连接。如果右表中满足指定条件的某行数据没有在左表中出现过，则此行数据保留在结果集中。	√	√

MERGE JOIN算子

算子名称	含义	适用行模式	适用列模式
MERGE SORT	MERGE JOIN的子PLAN，用于参与MERGE JOIN的KEY排序。	√	√
MERGE JOIN	MERGE JOIN内连接。先将左右表进行排序物化，然后通过JOIN条件的进行merge排序，可以支持等值、大于、大于等于、小于、小于等于的排序。	√	√

NEST LOOP算子

算子名称	含义	适用行模式	适用列模式
NEST LOOPS OUTER	NEST LOOP JOIN左外连接。以左表为准，当左右表满足join指定条件则同时返回左右表数据，如果满足不了，则将左表数据+右表补空返回。	√	√
NEST LOOPS FULL OUTER	NEST LOOP JOIN全外连接。当左右表满足join指定条件则同时返回左右表数据，如果满足不了，则将左表数据+右表补空、左表补空+右表数据返回。	√	√
NEST LOOPS SEMI	NEST LOOP JOIN左半连接。如果左表中满足指定条件的某行数据在右表中出现过，则此行保留在结果集中。Filter in / exists常常改造成SEMI JOIN。	√	√
NEST LOOPS ANTI	NEST LOOP JOIN左反半连接。如果左表中满足指定条件的某行数据没有在右表中出现过，则此行保留在结果集中。Filter not in / not exists常常改造成ANTI JOIN。	√	√
NEST LOOPS	NEST LOOP 内连接。两表通过循环遍历方式查找满足join条件的数据。	√	√

辅助功能算子

算子名称	含义	适用行模式	适用列模式
LOAD TABLE CONVENTIONAL	在刚才插入数据时，数据库执行了实时统计值收集的动作。	√	√
PARALLEL	并行执行的标识，表示此标识以下的算子都为并行执行。	×	√
MERGE	数据合并，将多路有序的数据源进行归并成一路数据的算子。	√	√
ROW TO COL	行计算转为列计算。	√	√
COL TO ROW	列计算转为行计算。	√	√
MATERIAL	将输入的数据源物化。	√	√

分布式优化

与单机相比，分布式SQL计算会带来额外的节点间的通信开销，为了提高性能，减少数据的跨网络传输带来的影响，YashanDB做了以下优化：

1.增加了PX算子，负责跨网络数据交换，PX有以下类型：

根据运行环境划分：

- REMOTE：分布式环境。
- LOCAL：单机并行环境。

根据数据交互节点划分：

- I2I：一个节点向另一个节点发送数据。
- I2N：一个节点向多个节点发送数据。
- N2I：多个节点向一个节点发送数据。
- N2N：多个节点向多个节点发送数据。

根据数据走向划分：

- PX send type：random、hash、broadcast。
- PX receive type：random、sort。

2.为了充分利用每个节点的计算资源，YashanDB优化器会尽量将计算下推到DN（两阶段计算），可以进行两阶段计算的算子包括：

- 两阶段ORDER BY
- 两阶段GROUP BY
- 两阶段DISTINCT
- 两阶段聚集
- 两阶段LIMIT
- COUNT STOPKEY

3.通过将小表创建为复制表，减少跨节点Join。

SQL调优原理与规则

YashanDB优化器内部存在较多的优化细节，其中，以下优化原理和规则是在进行SQL调优之前应该了解的：

- 查询改写与谓词优化
- 选择率与统计信息

通常情况下，优化器能做出最优的选择，但由于语句本身等一些原因没有生成最优计划时，需要进行SQL级的调优，通用的SQL调优方法包括：

- **改写SQL**

对于初学者，SQL可以被简单地理解为一种消息传递的语言——文本查询、数据返回。但如果不去深入理解和运用这门语言，可能会导致客户端经常会产生一些低效的SQL。

开发人员编写SQL语句用于向数据库发出请求，获得结果集，数据库端响应SQL并执行，对于产生相同结果的两条不同写法的SQL，其响应时间可能相差几个数量级。

当SQL运行出现性能问题时，调优人员应该首先排查该语句编写的是否合理，例如是否未使用绑定变量而直接使用常量（这将导致SQL执行计划无法被重用，而出现耗费资源的硬解析）。通过改写SQL实现性能提升是数据库性能调优中的重要一环。

- **hint调优**

在SQL本身已改写达到要求，而SQL执行效率仍不理想时，可能需要对优化器进行干预，让其生成符合预期的执行计划。

YashanDB中， 用户可以通过指定hint的方式来改变优化器对计划的选择。hint使用相关介绍见Hint。

需说明的是，在CBO模式下，优化器有时会忽略它认为不合理的hint，此时需要在知晓优化器工作原理的基础上进行分析，调整hint。

统计信息是优化器工作的基准，对于用户调优而言，了解SQL语句所涉及表的数据信息、业务逻辑、关联关系等情况则是做出合理hint选择的基本要求。

- **计划记录**

在使用完hint对计划进行调优之后，可以通过outline的方式对计划进行记录。在下次遇到相同的执行语句时，可以直接拿到稳定的执行计划。

outline使用相关介绍见Outline。

查询改写与谓词优化

冗余查询改写

考虑如下语句：

示例

```
SELECT AREA_NO FROM (SELECT * FROM AREA);
```

从语句内容可以看出，该SQL语句的目标是获取AERA表的AREA_NO列，但多出一个冗余的子查询，这将导致SQL引擎执行SQL时查询出多余的投影列。

这种冗余的写法增加了执行器的负担，因此YashanDB会将上述语句改写成如下类似结构：

示例

```
SELECT AREA_NO FROM AREA
```

"IN / NOT IN + 子查询"改写

对于 "IN + 子查询" 的语句，当查询中不包含AGGR, GROUP BY, CONNECT BY, WINDOW FUNCTION时，YashanDB则会改写成半连接或反连接。因为如果按照原SQL的语义去执行，则每次从BRANCHES表里取出一次数据时，都要完整执行一遍AREA的子查询。而改写后的语句，将会大幅减少AREA表的扫描次数，从而提升性能。

示例

```
EXPLAIN SELECT * FROM branches b WHERE b.area_no IN (SELECT area_no FROM area);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 2158759181
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+--------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+--------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|  1 |  NESTED LOOPS SEMI             |                      |            |      101|      179( 0)|
|
|  2 |   TABLE ACCESS FULL            | BRANCHES             | REGRESS    |   100000|      132( 0)|
|
|* 3 |   INDEX UNIQUE SCAN            | SYS_C_17             | REGRESS    |        1|        6( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+--------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   3 - Predicate : access("AREA"."AREA_NO" = "B"."AREA_NO")


EXPLAIN SELECT * FROM branches b WHERE b.area_no NOT IN (SELECT area_no FROM area);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 708164047
Optimizer: ADOPT_C
```

```
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  NESTED LOOPS ANTI             |                      |            |        1|  1463780( 0)|
|
|  2 |   TABLE ACCESS FULL            | BRANCHES             | REGRESS    |   100000|      132( 0)|
|
|  3 |   INDEX FAST FULL SCAN         | SYS_C_17             | REGRESS    |   100000|       92( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : filter(INVERT("B"."AREA_NO" <> "AREA"."AREA_NO"))
```

"EXISTS / NOT EXISTS + 子查询" 改写

与 "IN / NOT IN + 子查询" 类似，当查询中不包含AGGR, GROUP BY, CONNECT BY, WINDOW FUNCTION时，"EXISTS / NOT EXISTS + 子查询"的组合也可能被改写成半连接或反连接。

示例

```
EXPLAIN SELECT * FROM branches b WHERE EXISTS (SELECT area_no FROM area a WHERE b.area_no = a.area_no);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3585099968
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|  1 |  NESTED LOOPS SEMI             |                      |            |       101|      179( 0)|
|
|  2 |   TABLE ACCESS FULL            | BRANCHES             | REGRESS    |   100000|      132( 0)|
|
|* 3 |   INDEX UNIQUE SCAN            | SYS_C_23             | REGRESS    |         1|        6( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   3 - Predicate : access("A"."AREA_NO" = "B"."AREA_NO")


EXPLAIN SELECT * FROM branches b WHERE NOT EXISTS (SELECT area_no FROM area a WHERE b.area_no = a.area_no);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3371762311
Optimizer: ADOPT_C
```

```
+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT              |                     |            |          |             |
|
|  1 |  NESTED LOOPS ANTI            |                     |            |    99900|      179( 0)|
|
|  2 |   TABLE ACCESS FULL           | BRANCHES            | REGRESS    |   100000|      132( 0)|
|
|* 3 |   INDEX UNIQUE SCAN           | SYS_C_23            | REGRESS    |        1|        6( 0)|
|
+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   3 - Predicate : access("A"."AREA_NO" = "B"."AREA_NO")
```

"ANY/All + 子查询" 改写

当比较运算符(>、>=、<、<=) 右边为 any/all时，any 子查询会被改写成半连接，all子查询改写成反连接，例如：

示例

```
EXPLAIN SELECT product FROM sales_info_hash s WHERE amount > ANY(SELECT revenue_total FROM finance_info f WHERE s.branch =
f.branch);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 777020315
Optimizer: ADOPT_C

+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT              |                     |            |          |             |
|
|* 1 |  HASH JOIN SEMI               |                     |            |     2660|     2056( 0)|
|
|  2 |   PART SCAN ALL               |                     |            |   100000|      442( 0)| [0,1]
|
|  3 |    TABLE ACCESS FULL          | SALES_INFO_HASH     | SYS        |   100000|      442( 0)|
|
|  4 |   TABLE ACCESS FULL           | FINANCE_INFO        | SYS        |   100000|      442( 0)|
|
+----+-------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("S"."BRANCH" = "F"."BRANCH")
                   filter("S"."AMOUNT" > "F"."REVENUE_TOTAL")


EXPLAIN SELECT product FROM sales_info_hash WHERE amount > ALL(SELECT revenue_total FROM finance_info);

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3243986469
```

```
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                     |            |          |             |
|
|* 1 |  NESTED LOOPS ANTI             |                     |            |        1|   664389( 0)|
|
|  2 |   PART SCAN ALL                |                     |            |   100000|      442( 0)| [0,1]
|
|  3 |    TABLE ACCESS FULL           | SALES_INFO_HASH     | SYS        |   100000|      442( 0)|
|
|  4 |   VIEW                         |                     |            |   100000|      447( 0)|
|
|  5 |    TABLE ACCESS FULL           | FINANCE_INFO        | SYS        |   100000|      442( 0)|
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : filter(INVERT("SALES_INFO_HASH"."AMOUNT" > "VSQ$1@SEL$0"."REVENUE_TOTAL"))
```

谓词相关处理

谓词组合优化

谓词重组包括合并和扩展两类：

- 谓词合并，去除冗余的谓词条件

  多个冗余的谓词，合并成一个。例如： A = B AND A != B 谓词，合并为 恒FALSE。

- 谓词扩展，根据关系代数公式构造成等价高效的谓词条件

  根据已有条件，构造出新的谓词。例如：A > B AND B > C，可以推导出 A > C条件，在某些场景下，新构造出来的A > C能够将笛卡尔连接转换成内连接。

谓词下推

- 谓词下推就是在不改变谓词语义的情况下将谓词放置到效率最高的地方。很显然谓词所启的作用越早，效率越高。
- 如索引相关的放置到索引上，存储引擎可以根据该谓词减小返回的结果集大小，这比让存储引擎返回全部索引再由SQL引擎过滤结果集更优。

选择率与统计信息

本小节将描述选择率原理、统计信息与选择率的关系以及相关调优判断。在复杂Join相应的优化中可以通过例子具体了解选择率对计划的影响。

选择率

选择率是基于filter的概念，它指的是filter的选择比例。filter主要给算子做过滤作用，最简单的例子就是在一个select * from t1 where col = 1的语句中，filter能够把scan算子扫出来的全部数据进行过滤，筛选出只满足col = 1的结果。

一般filter都是依附于算子上。基于是否是join的依据，filter可以分为连接条件和过滤条件（一些地方也叫join condition和filter来区分）。

连接条件指的是on上的条件或者隐式内连接时where上的条件，比如下述两条查询语句。

示例

```
SELECT * FROM area INNER JOIN branches ON area.area_no = branches.area_no;
SELECT * FROM area, branches WHERE area.area_no = branches.area_no;
```

这两个filter则是连接条件，此外的可以认为是过滤条件。

而选择率就是filter在原来的数据基础上能够过滤出多少条数据的比率。假设t1一共有100条数据，而filter一共过滤出了10条。那么最终的选择率就是1/10。

选择率会直接影响估计出来的行数。并且在Cost Model里，行数是一个比较重要的评估要素。因此选择率会在很大程度上间接影响最终的cost评估。

统计信息

选择率计算的依据是收集上来的统计信息。因此统计信息的收集与否以及统计信息的准确与否很大程度上会影响选择率计算的准确性。 YashanDB的统计信息相关介绍见统计信息所述。

对于SQL调优来说，需要确保统计信息的实时性。通过查询视图和执行语句的方式，可以判断统计信息是否收集准确。 下面给出判断列最大值最小值的统计信息是否准确的例子， 其余的统计信息也可以通过类似的方式进行判断。

示例

```
SELECT HIGH_VALUE FROM DBA_TAB_COL_STATISTICS WHERE table_name = 'AREA' AND column_name = 'AREA_NO';
HIGH_VALUE
----------------------------
3035


SELECT MAX(area_no) FROM area;
MAX(AREA_NO)
------------
05


SELECT LOW_VALUE FROM DBA_TAB_COL_STATISTICS WHERE table_name = 'AREA' AND column_name = 'AREA_NO';

LOW_VALUE
--------------------------
3031


SELECT MIN(area_no) FROM area;
MIN(AREA_NO)
------------
01
```

Join调优介绍

Join Orders介绍

Join是SQL语法中最常见的一个特性，只要稍微复杂一点的业务场景都会用到Join。在入门知识中，已经大体介绍了计划是各个算子排列组合成的。对于Join来说，常常会有非常多的组合方式。这无疑使得选出一个最优的Join组合变得异常困难。

假设有t1, t2, t3, t4四张表，对它们执行如下join查询：

```
EXPLAIN SELECT * FROM t1, t2, t3, t4  WHERE t1.c1 = t2.c1 AND t2.c1 = t3.c1 AND t4.c1 = t2.c1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 745528657
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                     |            |          |             |
|
|* 1 |  HASH JOIN INNER               |                     |            |        1|     2137( 0)|
|
|  2 |   JOIN FILTER USE              |                     |            |   100000|      442( 0)|
|
|* 3 |    TABLE ACCESS FULL           | T4                  | REGRESS    |   100000|      442( 0)|
|
|* 4 |   JOIN FILTER CREATE           |                     |            |        1|     1499( 0)|
|
|* 5 |    MERGE JOIN INNER            |                     |            |        1|     1499( 0)|
|
|  6 |     MERGE SORT                 |                     |            |          |             |
|
|* 7 |      HASH JOIN INNER           |                     |            |        3|     1068( 0)|
|
|  8 |       TABLE ACCESS FULL        | T1                  | REGRESS    |   100000|      442( 0)|
|
|  9 |       TABLE ACCESS FULL        | T2                  | REGRESS    |        3|      431( 0)|
|
| 10 |     MERGE SORT                 |                     |            |          |             |
|
| 11 |      TABLE ACCESS FULL         | T3                  | REGRESS    |        3|      431( 0)|
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("T4"."C1" = "T2"."C1")
   3 - Predicate : RUNTIME FILTER(RUNTIME USE(0): "T4"."C1")
   4 - Predicate : RUNTIME FILTER(RUNTIME CREATE(0): "T2"."C1")
   5 - Predicate : access( "T1"."C1" = "T3"."C1" )
   7 - Predicate : access("T1"."C1" = "T2"."C1")
```

对于上述SQL语句来说，即使仅仅只有四层Join，在考虑Join算子种类、Join内Scan表的左右顺序的和Scan算子的种类的情况后，可能一共会几十种组合。

实际上，在数据库领域中的Join Orders本质就是一个NP hard难题。因此对于调优复杂Join来说，需要一些特定的技巧来规避如此大的解空间。

但在实际讲解如何进行Join调优之前，有必要对Join Orders相应的算法进行了解。这能有助于理解Join的生成方式，从而做到知其然，也知其所以然。

目前YashanDB中主要有两种Join算法：

1. Dynamic Programming - 动态规划：

- 在小于等于八张表连接时，目前会采用DP算法确定Join Orders。DP算法本质上是一种穷举法。基本思路为n张表join时，可以给定一个数字k，1<k<n，best(1..n) =best(1..k) + best(k..n) + cost(join(1..k), join(k..n))。 通过循环或者递归实现最佳join order的生成。因此在DP算法和正确的cost model下，一定能够找最优的Join计划。

2. Greedy Operator Ordering - 贪心算法：

- 在超过八张表连接时，YashanDB会采用GOO算法。这是因为随着表的数量增加，DP算法的复杂度已经无法支撑高效的Join Orders生成了。 因此需要另外一种算法进行Join Orders的选择。GOO算法可以描述如下：对于n个节点所组成的集合，两两组合从中找到join cost最小的两个节点。 将其join作为新的节点代替原来的两个节点；重复这一过程直到得到一个Join Tree。可以看到此算法的时间复杂度是O(n^3)，空间复杂度为O(n)。

复杂Join的调优

复杂Join的调优需要遵循一个大体原则：能够低成本过滤掉大量数据的Join优先执行。

```
EXPLAIN SELECT * FROM t1, t2, t3, t4  WHERE t1.c1 = t2.c1 AND t2.c1 = t3.c1 AND t4.c1 = t2.c1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 3320065567
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  MERGE JOIN INNER              |                      |            |        4|     1724( 0)|
|
|  2 |   MERGE SORT                   |                      |            |          |             |
|
|* 3 |    MERGE JOIN INNER            |                      |            |        1|     1293( 0)|
|
|  4 |     MERGE SORT                 |                      |            |          |             |
|
|* 5 |      HASH JOIN INNER           |                      |            |        1|      862( 0)|
|
|  6 |       TABLE ACCESS FULL        | T1                   | REGRESS    |       79|      431( 0)|
|
|  7 |       TABLE ACCESS FULL        | T2                   | REGRESS    |        3|      431( 0)|
|
|  8 |     MERGE SORT                 |                      |            |          |             |
|
|  9 |      TABLE ACCESS FULL         | T3                   | REGRESS    |        3|      431( 0)|
|
| 10 |   MERGE SORT                   |                      |            |          |             |
|
| 11 |    TABLE ACCESS FULL           | T4                   | REGRESS    |       13|      431( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access( "T2"."C1" = "T4"."C1" )
   3 - Predicate : access( "T1"."C1" = "T3"."C1" )
   5 - Predicate : access("T1"."C1" = "T2"."C1")
```

对于T1、T2这种能够提前过滤掉大部分数据的Join，在复杂Join调优时就应当让其提前。由于先执行了这类过滤性强的Join，使得后续需要执行的条数大大降低，从而降低了整体SQL的执行成本。

正常来说如果连接表数量低于8，SQL引擎本身的DP算法已经足够优秀去选出较好的Join Orders。而在连接表数量超过8时，由于GOO算法本身的不足，Join Orders可能会出现不够优秀的问题，导致SQL执行效率不理想，此时需要人工介入对这种复杂Join进行调优。

对于复杂Join的SQL来说，穷举所有Join Orders的情况进行调优是比较困难的。因此需要基于某些规则进行调优的尝试。大体的规则如下：

1. 选择率判断：判断各个算子的Rows评估是否准确。
2. 算子选择判断：判断各个算子的选择是否合理。

选择率判断

在算子上如果存在filter，则会对算子生成的rows进行一层过滤。而过滤多少的预估则是由选择率来实现的。如果选择率的估计不够准确，则会导致rows评估偏差较大。进而导致cost model产生误判，使得Join Orders不够优秀。

因此实际调优Join Orders的第一步往往都是查看选择率判断是否准确。

```
EXPLAIN SELECT * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 SELECTIVITY 0.0001 AND t2.c1 = t3.c1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 234816895
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  MERGE JOIN INNER              |                      |            |        1|     1293( 0)|
|
|  2 |   MERGE SORT                   |                      |            |          |             |
|
|* 3 |    HASH JOIN INNER             |                      |            |        1|      862( 0)|
|
|  4 |     TABLE ACCESS FULL          | T1                   | REGRESS    |       79|      431( 0)|
|
|  5 |     TABLE ACCESS FULL          | T2                   | REGRESS    |        3|      431( 0)|
|
|  6 |   MERGE SORT                   |                      |            |          |             |
|
|  7 |    TABLE ACCESS FULL           | T3                   | REGRESS    |        3|      431( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access( "T1"."C1" = "T3"."C1" )
   3 - Predicate : access("T1"."C1" = "T2"."C1")

SELECT COUNT(*) FROM t1, t2 WHERE t1.c1 = t2.c1 SELECTIVITY 0.0001;

COUNT(*)
--------------------
                  79
```

在上述例子中，可以看到对于T1和T2由于错误的估计了Join后的剩余条数，使得T1和T2先进行了Join。这将会导致Join Orders的判断错误。使得最终计划不佳。在实际调优时，YashanDB提供了两个能力判断Rows的评估是否准确。

explain

如上述例子所示，通过实际select出来的值和explain出来的值可以判断rows评估是否准确。具体使用可见EXPLAIN文档。

autotrace

直接开启autotrace可以替代上述过程。具体使用可见AUTOTRACE文档。

```
SELECT * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 SELECTIVITY 0.0001 AND t2.c1 = t3.c1;

Execution Plan
----------------------------------------------------------------
SQL hash value: 234816895
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+-------------------------------+
| Id | Operation type                 | Name                | Owner      | E - Rows | A - Rows | Cost(%CPU)  | A - Time |
Loops    | Memory   | Disk     | Partition info                |
+----+--------------------------------+---------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+-------------------------------+
|  0 | SELECT STATEMENT               |                     |            |          |       79|             |       354|
79|          |          |                               |
|* 1 |  MERGE JOIN INNER              |                     |            |         1|       79|    1293( 0)|       341|
79|          |          |                               |
|  2 |   MERGE SORT                   |                     |            |          |       79|             |       242|
79|          |          |                               |
|* 3 |    HASH JOIN INNER             |                     |            |         1|       79|     862( 0)|       188|
79|          |          |                               |
|  4 |     TABLE ACCESS FULL          | T1                  | REGRESS    |        79|       79|     431( 0)|        24|
79|          |          |                               |
|  5 |     TABLE ACCESS FULL          | T2                  | REGRESS    |         3|        3|     431( 0)|        53|
3|          |          |                               |
|  6 |   MERGE SORT                   |                     |            |          |       59|             |        42|
59|          |          |                               |
|  7 |    TABLE ACCESS FULL           | T3                  | REGRESS    |         3|        3|     431( 0)|        26|
3|          |          |                               |
+----+--------------------------------+---------------------+------------+----------+----------+-------------+----------+---
-------+----------+----------+-------------------------------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access( "T1"."C1" = "T3"."C1" )
   3 - Execution : [BUILD] Memory : 327680  [BUILD] Time : 133  [HDT] RehashTimes : 0  [HDT] OriginPrime : 131071
       Predicate : access("T1"."C1" = "T2"."C1")
```

算子选择判断

除了选择率判断之外，选择的算子是否准确也至关重要。一个错误的算子选择常常会导致非常低的效率。下面也将结合例子展开描述。

```
EXPLAIN SELECT /*+LEADING(t1,t2) USE_HASH(t1,t2)*/ * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 AND t2.c1 = t3.c1;

PLAN_DESCRIPTION
----------------------------------------------------------------
SQL hash value: 4196032415
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                     |            |          |             |
|
|* 1 |  HASH JOIN INNER               |                     |            |   1651424|     4545( 0)|
|
|* 2 |   HASH JOIN INNER              |                     |            |   1651424|      885( 0)|
|
|  3 |    TABLE ACCESS FULL           | T1                  | REGRESS    |      2528|      431( 0)|
|
|  4 |    TABLE ACCESS FULL           | T2                  | REGRESS    |      1899|      431( 0)|
|
|  5 |   TABLE ACCESS FULL            | T3                  | REGRESS    |         3|      431( 0)|
|
```

```
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("T1"."C1" = "T3"."C1")
   2 - Predicate : access("T1"."C1" = "T2"."C1")


EXPLAIN SELECT /*+LEADING(t1,t2) USE_NL(t1)*/ * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 AND t2.c1 = t3.c1;

PLAN_DESCRIPTION
-----------------------------------------------------------------
SQL hash value: 462255180
Optimizer: ADOPT_C

+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type               | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT             |                     |            |          |             |
|
|* 1 |  HASH JOIN INNER             |                     |            |   1651424|     4848( 0)|
|
|* 2 |   NESTED LOOPS INNER         |                     |            |   1651424|     1189( 0)|
|
|  3 |    TABLE ACCESS FULL         | T1                  | REGRESS    |      2528|      431( 0)|
|
|  4 |    TABLE ACCESS FULL         | T2                  | REGRESS    |      1899|      431( 0)|
|
|  5 |   TABLE ACCESS FULL          | T3                  | REGRESS    |         3|      431( 0)|
|
+----+------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("T1"."C1" = "T3"."C1")
   2 - Predicate : filter("T1"."C1" = "T2"."C1")
```

在上例中，可以看到选择Nested Loops Inner时比选择Hash Join Inner时，代价要大一些。这是因为Join相关算子有各自的适用场景，对于大表之间Join来说，采用Hash Join可能更为合适。 此外，如果两个表的Join列都是有序的，可能选择Merge Join比较合适。 在一些复杂的场景中，很容易出现类似Join相关算子选择错误的问题。比如在大表Join时没有索引的情况下选择了Nest Loop Join而不是选择Hash Join，以及选择Hash Join时采用了大表进行Hash表的创建。因此如果要对算子进行判断，需要了解各个算子相关的适用场景，做到对各个算子心中有数。

hint调优

结合上述的两类问题，主要有三种hint可以对Join Orders进行调节。

1. Join Hint.
2. Index Hint.
3. Selectivity Hint.

选择率调整

上述出现选择率问题时，可以通过Selectivity Hint来进行调整。

```
EXPLAIN SELECT * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 SELECTIVITY 0.0001 AND t2.c1 = t3.c1;

PLAN_DESCRIPTION
-----------------------------------------------------------------
SQL hash value: 234816895
Optimizer: ADOPT_C
```

```
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  HASH JOIN INNER               |                      |            |      480|     1317( 0)|
|
|* 2 |   HASH JOIN INNER              |                      |            |      480|      885( 0)|
|
|  3 |    TABLE ACCESS FULL           | T1                   | REGRESS    |     2528|      431( 0)|
|
|  4 |    TABLE ACCESS FULL           | T2                   | REGRESS    |     1899|      431( 0)|
|
|  5 |   TABLE ACCESS FULL            | T3                   | REGRESS    |        3|      431( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("T1"."C1" = "T3"."C1")
   2 - Predicate : access("T1"."C1" = "T2"."C1")


EXPLAIN SELECT * FROM t1, t2, t3  WHERE t1.c1 = t2.c1 AND t2.c1 = t3.c1 SELECTIVITY 1;

PLAN_DESCRIPTION
--------------------------------------------------------------------
SQL hash value: 2568121636
Optimizer: ADOPT_C

+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                 | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                      |            |          |             |
|
|* 1 |  HASH JOIN INNER               |                      |            |   1651424|     1321( 0)|
|
|* 2 |   HASH JOIN INNER              |                      |            |      2528|      866( 0)|
|
|  3 |    TABLE ACCESS FULL           | T1                   | REGRESS    |      2528|      431( 0)|
|
|  4 |    TABLE ACCESS FULL           | T3                   | REGRESS    |         3|      431( 0)|
|
|  5 |   TABLE ACCESS FULL            | T2                   | REGRESS    |      1899|      431( 0)|
|
+----+--------------------------------+----------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("T1"."C1" = "T2"."C1")
   2 - Predicate : access("T1"."C1" = "T3"."C1")
```

可以看到，YashanDB通过Selectivity Hint的方式可以立竿见影的完成对Join Orders的调整。但这类的调整由于粒度较小，需要大体了解选择率相关的处理逻辑。 具体选择率生成相关原理可见：选择率与统计信息命令。

算子调整

算子的调整则粒度更大一些，直接通过hint的方式将执行的算子进行改变。在进行这类操作之前，需要对各个算子的原理以及优劣了解清楚。

```
EXPLAIN SELECT /*+INDEX(t1)*/ c1 FROM t1;

PLAN_DESCRIPTION
------------------------------------------------------------------
SQL hash value: 1102139983
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                     |            |          |             |
|
|  1 |  COL TO ROW                    |                     |            |          |             |
|
|  2 |   INDEX FAST FULL SCAN         | C1_T1               | REGRESS    |         8|      142( 0)|
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Projection: RemoteTable[1][INTEGER]
   2 - Projection: Tuple[0, 0][INTEGER]
```

上述例子中，由于只需要拿c1列不需要回表，所以我们可以使用hint指定走index scan。从而手动使得计划最优。对于其他使用错的算子也是同理，比如在选错成nested loop join时，使用Hint指定使用hash join来进行规避。

SQL调优工具

与SQL调优相关的工具主要有explain、autotrace、hint和outline。其中，explain和autotrace用来分析计划，hint和outline则可以直接作用于SQL语句上。

YashanDB相关hint的使用可见：Hint

YashanDB相关explain的使用可见：执行计划

YashanDB相关autotrace的使用可见：AUTOTRACE

YashanDB相关outline的使用可见：Outline

Hint

Hint是一种特殊的注释，它以固定的格式和位置出现在SQL语句的文本中，用于影响优化器对于执行计划的选择。但这种影响不是强制的，优化器在某些情况下可能忽略SQL的HINT指令。

改变Join连接顺序的Hint

LEADING 语法能够改变Join两边表的顺序，常见做法是通过LEADING调整大小表，减少扫描数量。或通过LEADING调整表连接顺序，增加选择率，减少执行负担。

LEADING有一定的限制，例如只能调整具体表的顺序，而不能调整表与表连接后产生的结果集与其他结果集之间的顺序。

改变Join类型的Hint

USE_ 和 NO_USE_ 语法能够指定两张表的连接类型：

模式	说明	备注
USE_NL	优先选择Nesed Loop Join	任何Join类型一定能转成Nested Loop Join
NO_USE_NL	优先选择非Nesed Loop Join类型	无法生成其他Join类型时候，不生效
USE_HASH	优先使用Hash Join	无法使用Hash Join时候，不生效
NO_USE_HASH	不使用 Hash Join	根据代价评估，会使用Nested Loop Join或者Merge Join
USE_MERGE	优先使用Merge Join	无法生成Merge Join时候，不生效
NO_USE_MERGE	不使用Merge Join	根据评估，使用Nested Loop Join或者Hash Join

改变访问路径的Hint

Hint指令也可以改变访问路径，比如强制走索引或强制走全表扫描。

模式	说明	备注
INDEX	优先使用Hint指定的索引	一定会生效，选Fast Full Scan和Index Scan之间Cost最小的一个
NO_INDEX	不使用Hint指定的索引	一定会生效，选择Table Full Scan
INDEX_FFS	优先使用Fast Full Scan	一定会生效，只选Fast Full Scan
NO_INDEX_FFS	不使用Fast Full Scan	一定会生效，可选Index Range Scan等
FULL	使用全表扫描	一定会生效，选择Table Full Scan

Outline

Outline通过固定SQL语句执行计划确保执行计划稳定性。使用CREATE OUTLINE来创建指定SQL对应的存储纲要，Outline会根据SQL语句对应的当前执行计划生成对应的Hint信息，当存储纲要可用时，数据库将自动根据这些Hint信息生成对应SQL语句的执行计划。

Outline开关

系统通过USE_STORED_OUTLINES（默认值`FALSE`）控制Outline开关，对该参数的值说明如下：

- true：使用Category为DEFAULT的Outline。
- false：不使用Outline，为参数默认值。
- category_name ：使用指定Category的Outline。

Category

在创建OUTLINE时，需要为其指定一个类别，用于将Outline分类管理和控制。

Outline管理

Outline由用户来定义和维护，包括创建、修改和删除。具体操作请参考CREATE OUTLINE、ALTER OUTLINE、DROP OUTLINE。

成功创建的Outline可通过DBA_OUTLINES/ALL_OUTLINES/USER_OUTLINES视图查看其定义信息，通过DBA_OUTLINE_HINTS/ALL_OUTLINE_HINTS/USER_OUTLINE_HINTS视图查看其包含的Hint。

Outline使用

1.创建一个Outline：

示例

```
-- 对给定的SQL创建Outline。
CREATE OUTLINE yashan_outline FOR CATEGORY ctgy_yashan ON
SELECT /*+ FULL(a) */ a.area_name, b.branch_name
FROM area a, branches b
WHERE a.area_no = b.area_no
AND b.branch_no LIKE '01%'
AND a.area_no LIKE '01';

CREATE OUTLINE yashan_outline1 FOR CATEGORY ctgy_yashan ON
SELECT /*+ FULL(a) */ a.area_name
FROM area a
WHERE a.area_no LIKE '01';
```

2.开启ctgy_yashan类别的Outline，该类别下所有的Outline都将会生效并被优化器识别：

示例

```
ALTER SESSION SET USE_STORED_OUTLINES=ctgy_yashan;
```

3.查看Outline是否生效，请注意系统将首先对此SQL语句和ctgy_yashan类别下的SQL语句进行大小写不敏感匹配，匹配成功的情况下，优化器才会采纳该Outline并在执行计划中进行列示，如下：

示例

```
EXPLAIN
SELECT /*+ FULL(a) */ a.area_name, b.branch_name
FROM area a, branches b
WHERE a.area_no = b.area_no
AND b.branch_no LIKE '01%'
AND a.area_no LIKE '01';

PLAN_DESCRIPTION
```

```
-------------------------------------------------------------
SQL hash value: 1136128624
Optimizer: ADOPT_C

+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
| Id | Operation type                 | Name                | Owner      | Rows     | Cost(%CPU)  | Partition info
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+
|  0 | SELECT STATEMENT               |                     |            |          |             |
|
|* 1 |  HASH JOIN INNER               |                     |            |     8000|      722( 0)|
|
|* 2 |   TABLE ACCESS FULL            | AREA                | SALES      |    10000|      445( 0)|
|
|* 3 |   TABLE ACCESS BY INDEX ROWID  | BRANCHES            | SALES      |          |             |
|
|* 4 |    INDEX RANGE SCAN            | SYS_C_36            | SALES      |    10000|      159( 0)|
|
+----+--------------------------------+---------------------+------------+----------+-------------+-------------------------
-------+

Operation Information (identified by operation id):
---------------------------------------------------

   1 - Predicate : access("A"."AREA_NO" = "B"."AREA_NO")
   2 - Predicate : filter("A"."AREA_NO" LIKE '01')
   3 - Predicate : filter("B"."BRANCH_NO" LIKE '01%')
   4 - Predicate : access("B"."BRANCH_NO" LIKE '01%')

Hint Information :
---------------------------------------------------

full(a) / rejected by IGNORE_OPTIM_EMBEDDED_HINTS

Outline Information :
---------------------------------------------------

   - outline OL_AB used for this statement
```

4.查看视图，确认Outline包含的hint是否与执行计划中所列算子和顺序一致：

示例

```
SELECT join_pos, hint
FROM USER_OUTLINE_HINTS
WHERE name = 'OL_AB';
JOIN_POS HINT
-------- ------------------------
       0 LEADING(A B)
       0 USE_HASH(A B)
       1 FULL(A)
       1 INDEX(B SYS_C_36)
```