数据库系统原理小组调研
OceanBase索引

小组成员：童海博 郑星云 赵克祥 汪清濯 王谦 陈凌锐

学号	姓名	贡献度	分工简述
2151130	童海博	16.67%	数据库索引的基础知识内容
2151610	郑星云	16.67%	OceanBase的索引特点
2152214	赵克祥	16.67%	OB索引的工作过程以及功能或者优化的解析
2154057	王清濯	16.67%	OB索引功能不断发展的新特性
2151140	王谦	16.67%	内容整合+ppt初步制作
2152587	陈凌锐	16.67%	优化ppt+汇报

1 广义的数据库索引基础知识介绍
- 1.1 引言
- 1.2 数据库索引的概念
- 1.3 索引的优缺点点
- 1.4 索引的结构（以MySQL为例）
  - 1.4.1 Hash表
  - 1.4.2 B树(改造二叉树)
  - 1.4.3 B+树(改造B树)
- 1.6 索引的新的趋势和技术

2 OceanBase 数据库的索引及其特点
- 2.1 局部索引与全局索引
- 2.2 唯一索引与非唯一索引
- 2.3 索引存储特点

3 OceanBase 数据库索引实现原理
- 3.1 全局索引实现原理流程
  - 3.1.1 生成全局索引的控制任务
  - 3.1.2 单副本构建
  - 3.1.3 多副本拷贝
  - 3.1.4 唯一性校验
  - 3.1.5 索引状态变更
  - 3.1.6 中间结果清理
- 3.2 局部索引实现原理流程

4 OceanBase 索引功能不断发展的新特性
- 4.1 索引管理：更高的索引管理效率
- 4.2 OceanBase 4.1中的GV$SESSION_LONGOPS视图
- 4.3 OceanBase v4.2函数索引调整特性
  - 4.3.1 OceanBase 4.2在MySQL模式下支持函数索引功能对函数索引支持的总结

1 广义的数据库索引基础知识介绍

1.1 引言

在现代数据库管理系统中，索引是一种至关重要的数据结构，用于提高数据库查询的效率。就像图书馆的索引卡帮助您快速找到特定的书一样，数据库索引允许快速访问数据库表中的数据行。没有索引，数据库必须执行全表扫描，即检查表中的每一行数据，以找到匹配的行，这在大型数据库中是非常耗时的。

1.2 数据库索引的概念

数据库索引是一个指针集合，它们指向存储在磁盘上的数据表中的记录。索引是为了加速对数据表中数据的检索而创建的。它们类似于书籍的目录，可以快速定位到数据，而不必查看整个数据库。

当对数据库进行查询时，如使用 SQL 的 SELECT 语句，数据库管理系统会检查是否存在可用的索引来加速查找过程。如果有，系统会使用索引来快速定位数据，而不是扫描整个表。

数据库的索引通常可分为一下几类：

1. **主键索引**：自动为表的主键列创建。每个表只能有一个主键索引，它保证了表中每一行的唯一性。
2. **唯一索引**：确保某一列或列组合的每个值都是唯一的。
3. **复合索引**：包含两个或多个列的索引，适用于经常一起查询的列。
4. **全文索引**：专门用于全文搜索，常见于搜索文本数据。

1.3 索引的优缺点

优点：

1. **提高查询速度**：这是索引最明显的好处。特别是对于大型数据库，索引可以显著减少数据查找时间。
2. **提高排序和分组的速度**：索引也加速了 ORDER BY 和 GROUP BY 这样的 SQL 语句的执行。

缺点：

1. **增加存储空间**：索引需要额外的空间来存储。
2. **影响写操作的性能**：插入、删除和更新操作可能会因为索引的调整而变慢。

1.4 索引的结构（以MySQL为例）

根据存储引擎的不同，实现方式也不同。MySQL的索引数据结构最常使用的是B树中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。（InnoDB中data阈存储的是行数据，而MyISAM中存储的是磁盘地址）

1.4.1 Hash表

Hash表，在Java中的HashMap，TreeMap就是Hash表结构，以键值对的形式存储数据。我们使用hash表存储表数据结构，Key可以存储索引列，Value可以存储行记录或者行磁盘地址。Hash表在等值查询时效率很高，时间复杂度为O(1)；但是不支持范围快速查找，范围查找时只能通过扫描全表的方式，筛选出符合条件的数据。

这种方式，不是特别适合我们经常需要查找和范围查找的数据库索引使用。

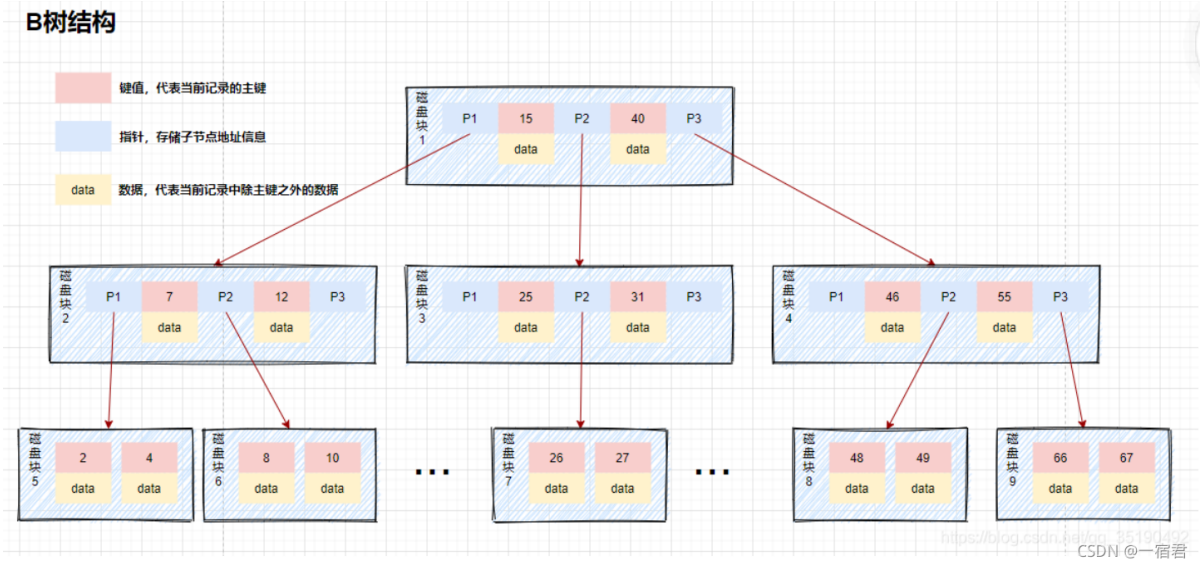
1.4.2 B树(改造二叉树)

MySQL的数据是存储在磁盘文件中的，查询处理数据时，需要先把磁盘中的数据加载到内存中，磁盘IO操作非常耗时，所以我们优化的重点就是尽量减少磁盘的IO操作。访问二叉树的每个节点都会发生一次IO，如果想要减少磁盘IO操作，就需要尽量降低树的高度。

B树是一种多叉平衡查找树，如下图主要特点：

1. B树的节点中存储这多个元素，每个内节点有多个分叉。
2. 节点中的元素包含键值和数据，节点中的键值从大到小排列。也就是说，在所有的节点中都存储数据。
3. 父节点当中的元素不会出现在子节点中。
4. 所有的叶子节点都位于同一层，叶子节点具有相同的深度，叶子节点之间没有指针连接。

B树数据结构大致如下：

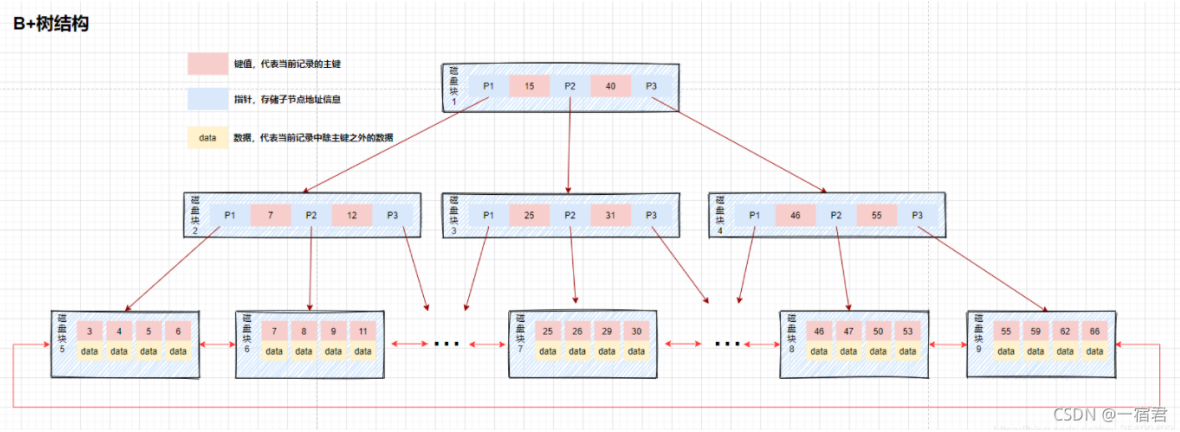


1.4.3 B+树(改造B树)

B+树，作为B树的升级版，MySQL在B树的基础上继续进行改造，使用B+树构建索引。B+树和B树最主要的区别在于**非叶子节点是否存储数据**的问题。

- B树：叶子节点和非叶子节点都会存储数据。
- B+树：只有叶子节点才会存储数据，非叶子节点只存储键值key；叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表。

B+树的大致数据结构：



B+树可以保证等值和范围查询的快速查找，MySQL的索引采用的就是B+树的结构。

1.6 索引的新的趋势和技术

以下是一些最新的趋势和技术，这些可能仍在当前的数据库索引领域中发挥重要作用：

1. **自适应索引**：自适应索引技术是一种动态创建和调整索引的方法，可以根据查询模式的变化自动优化。这种类型的索引可以实时调整其结构，以适应数据库的使用模式，从而减少了手动维护的需要。
2. **分布式索引**：随着分布式数据库系统的普及，分布式索引技术变得越来越重要。在这种设置中，索引被分布在多个节点上，以提高大数据环境中的查询效率和可扩展性。
3. **全文索引优化**：随着文本数据量的增加，全文索引变得越来越复杂。最新的全文索引技术，如倒排索引，提供了更高效的文本搜索能力。这些技术通常用于大型文档存储和搜索引擎。
4. **机器学习集成**：机器学习算法正在被用来优化索引结构。通过分析查询模式，机器学习模型可以预测哪些索引最有可能被查询，从而优化索引创建和维护过程。
5. **多维索引技术**：对于涉及多个维度或键的查询，例如在地理空间数据库中，多维索引技术如 R 树正在不断改进，以提供更高效的数据访问和查询性能。
6. **内存优化索引**：随着内存价格的下降和容量的增加，内存优化数据库系统变得越来越受欢迎。在这些系统中，索引完全保留在内存中，大大加快了数据访问速度。
7. **索引压缩技术**：为了减少索引所需的存储空间，索引压缩技术正在发展。这对于大型系统来说尤其重要，因为索引可能占用大量空间。
8. **非关系型数据库索引**：随着非关系型数据库，如 NoSQL 数据库的流行，为这些类型的数据结构设计高效的索引成为了一个挑战。非关系型数据库通常需要不同于传统关系型数据库的索引策略。

2 OceanBase 数据库的索引及其特点

2.1 局部索引与全局索引

局部索引

局部索引是一种特殊的索引类型，适用于分区表。在这种索引中，每个主表分区都拥有独立的索引数据结构。局部索引的关键特性在于，它的索引键仅映射到相应分区中的主表数据，不涉及其他分区。这种索引方式使得每个分区维护自己的索引结构，从而提高了特定查询的效率。

例如，考虑一个按 `emp_id` 进行范围分区的 `employee` 表，并在 `emp_name` 上创建局部索引。这意味着，每个 `emp_id` 范围分区都有一个专属的 `emp_name` 索引，确保了索引的高效和相关性。

全局索引

与局部索引不同，全局索引覆盖了分区表的所有主表分区。它们不是与单个分区相对应，而是将所有分区视为一个整体。全局索引有两种主要形式：

1. **全局非分区索引（Global Non-Partitioned Index）**： 这种索引类型不对数据进行分区，维持单一的数据结构。尽管主表已分区，全局非分区索引的一个键可能映射到不同主表分区中的多条数据。这使得全局非分区索引在处理跨分区数据时更为有效。
2. **全局分区索引（Global Partitioned Index）**： 在这种情况下，索引数据根据特定方式（如哈希或范围分区）分散到不同的索引分区中。这种索引独立于主表的分区模式，可能导致索引分区与主表分区之间形成多对多的对应关系。

以 `employee` 表为例，表可能按 `emp_id` 进行范围分区，而在 `emp_name` 上创建全局分区索引。在这种配置下，一个索引分区内的键可能指向多个不同的主表分区。

在决定使用局部索引还是全局索引时，应考虑以下因素：

- 当业务需求包括对非主键列的全局唯一性要求时，推荐使用全局索引。
- 如果查询条件不包括分区键，而且表不面临高并发写入，全局索引可以提高效率。
- 当查询能够利用分区键时，局部索引通常是更优的选择，因为它们在优化查询和写入性能方面更有效。

在 OceanBase 数据库中，创建索引时若未明确指定 `LOCAL` 或 `GLOBAL`，则在分区表上默认创建全局索引（`GLOBAL`）。

2.2 唯一索引与非唯一索引

唯一索引

唯一索引是一种保证表中索引列不会有重复值的索引类型。在 OceanBase 中，唯一索引的一个关键特性是它在存储键中结合了用户指定的索引列和一个可变的主表主键列。这种可变性意味着主键列的值会随索引列值的变化而变化。当索引列值为 NULL 时，可变的主键列值将会是主表中该列的实际值；当索引列值不为 NULL 时，可变的主键列值则为 NULL。

例如，在创建如下唯一索引时：

```
sqlCopy code
CREATE UNIQUE INDEX i2 ON t1(c3);
```

索引表 `i2` 的存储键是 `c3`，而与之配对的 `c1`（主键列）的值会根据 `c3` 的值（NULL 或非 NULL）而变化。

非唯一索引

与唯一索引不同，非唯一索引允许表内的索引列上存在重复值。在 OceanBase 中，非唯一索引的存储键由用户指定的索引列和主表的主键组成。这种索引类型适用于那些需要支持相同值的情况，例如，当同一列中的多个行可能具有相同的值时。

考虑以下非唯一索引的创建：

```
sqlCopy code
CREATE INDEX i1 ON t1(c2);
```

在这个例子中，索引表 `i1` 的存储键包括 `c2` 和主键列 `c1`。这意味着每个 `c2` 值都与相应的 `c1` 值相关联，即使 `c2` 的值在多个行中重复出现。

2.3 索引存储特点

索引表的存储结构

在 OceanBase 数据库中，索引表的存储结构与普通数据表相似，均使用宏块和微块的结构。由于 OceanBase 采用聚集索引表模型，索引表行中不仅包含用户指定的索引列，还会存储主表的主键列。这种设计有助于优化查询过程，因为它允许系统在进行索引查找后，直接回到主表中获取相关数据。

读写分离架构

OceanBase 采用读写分离架构，将数据分为基线数据和增量数据。其中：

- **增量数据（MemTable）**：存放于内存中，用于记录数据的修改操作。DML（数据操纵语言）操作完全在内存中进行，确保了高性能。
- **基线数据（SSTable）**：存储在 SSD 盘上。当进行数据读取时，系统会将内存中更新过的数据版本与持久化存储中的基线版本合并，以提供最新版本的数据。

此外，OceanBase 还实现了 Block Cache 和 Row Cache，减少对基线数据的随机读取。

宏块和微块

OceanBase 的数据文件以 2MB 的宏块为单位组织数据。每个宏块内部划分为多个 16KB 大小的微块（Micro Block），每个微块包含多个行（Row）。OceanBase 的内部 I/O 操作最小单位是微块。

宏块可以根据数据的删除、插入和更新动态地合并和分裂。

数据编码与压缩

OceanBase 通过数据编码压缩技术实现高效的空间利用。它采用基于列的数据编码，针对不同字段的值域和类型信息，选择合适的编码方式，从而提高压缩效率。这种方法比传统的按行压缩更有效，因为列数据的相似性更高。

冻结和合并

OceanBase 的读写分离架构需要定期将内存中的 MemTable 写入磁盘，这涉及到冻结和合并操作。其中：

- 冻结：指阻止当前 MemTable 的新写入，生成新的活跃 MemTable 的过程。
- 全量合并：涉及到将当前的静态数据和内存中的动态数据合并后写入磁盘作为新的静态数据。
- 增量合并：仅重写发生修改的宏块，大大减少合并工作量。
- 渐进合并：将数据重写分散到多次合并中进行，减轻了合并的负担。

轮转合并

为了减少合并操作对业务的影响，OceanBase 引入了轮转合并机制。在这种机制下，当一个数据副本在进行合并时，会将该副本上的查询流量切换到其他未在合并的副本上，从而保证业务查询不受影响。

转储

OceanBase 还实现了转储机制，即将内存中的增量数据独立存储于磁盘，不与全局静态数据合并。这种设计基于增量数据远小于全局数据的考虑，使得转储操作快速高效。

3 OceanBase 数据库索引实现原理

3.1 全局索引实现原理流程

3.1.1 生成全局索引的控制任务

1. **生成索引表的基础信息：** `ObIndexBuilder::generate_schema` 负责生成索引表的基础信息，包括列信息。索引表的基础信息主要继承自主表，但需要考虑唯一索引的情况，其中唯一索引需要带有索引列，同时需要处理隐藏主键列的情况，以解决索引列值为null时的比较问题。
2. **设置索引表状态：** 在生成索引表的schema时，将索引表的状态置为 `INDEX_STATUS_UNAVAILABLE`，表示索引表还不可用。
3. **将schema写入内部表：** 使用 `ObDDLOperator::create_table` 将生成的索引表的schema写入内部表。
4. **生成索引表的位置信息：** 通过 `ObDDLService::generate_global_index_locality_and_primary_zone` 生成索引表的位置信息，包括分区信息等。
5. **通知目标机器创建索引表的内存结构：** 通过 `ObDDLService::create_table_partitions` 向目标机器发送RPC通知，要求它们创建索引表的各个分区的内存结构，包括memtable、table_store以及partition_key到table_store的映射等。
6. **通知其他机器刷新schema：** 通过 `ObDDLService::publish_schema` 通知其他机器刷新schema，确保它们能够获取最新的索引表信息。
7. **提交全局索引的数据补全控制任务：** 通过 `ObGlobalIndexBuilder::submit_build_global_index_task` 将全局索引的数据补全的控制任务提交到队列中。同时，在提交该控制任务的同时，`submit_build_global_index_task` 在 `__all_index_build_stat` 中创建一条任务记录，用于跟踪任务的状态。
8. **执行全局索引的控制任务：** 由 `ObGlobalIndexBuilder` 负责执行，该线程池只有1个线程，队列长度受限于内存。执行的入口是 `ObGlobalIndexBuilder::run3 -> ObGlobalIndexBuilder::try_drive`。在执行过程中，根据状态机的方式推进全局索引的数据补全流程，包括单副本构建、多副本拷贝、唯一性校验等。

3.1.2 单副本构建

1. **选取快照点：** 首先需要选择一个快照点，确保在该快照点之后的主表的DML操作（增量数据）都可以被索引表看到。这种"write-only"的行为是实现在线建索引的关键。
2. **等待事务结束：** 为了确定选取的快照点，需要等待所有依赖于schema_version小于等于v1的事务都结束。通过 `do_get_associated_snapshot` 函数发送RPC请求给主表分区的leader，询问这些事务是否已经结束。通过 `ObService::check_schema_version_elapsed` 处理收到的请求，然后通过 `wait_all` 等待所有RPC的返回。注意，这里使用批量同步RPC，可能在分区数较多时会阻塞索引任务推动线程。
3. **Hold住快照点：** 为了确保在单副本构建过程中，选中的快照点不被释放，需要 `hold` 住该快照点。需要注意如果 `hold` 住快照的时间过长，可能会导致 `table_store` 的数量激增。
4. **更新构建快照点信息：** 将选中的构建快照点信息更新到内部表 `__all_index_build_stat` 中，以便后续追踪和管理。
5. **提交索引表基线数据构建任务：** 通过 `ObIndexSSTableBuildTask` 提交索引表基线数据的构建任务。这个任务由 `IdxBuild` 线程池执行，线程数为16，任务队列为4096。

6. **基线补全：** 通过 `drive_this_build_single_replica` 不断检查基线补全任务的状态，如果基线构建完成，则通过 `checksum` 校验来检查主表和索引表数据的一致性。

3.1.3 多副本拷贝

1. **发起多副本拷贝任务：** 在单副本构建流程中，当基线数据构建完成后，需要将这份数据拷贝到其他副本。这个过程由 `ObCopySSTableTask` 完成，该任务被 `rs` 的 `ObRebalanceTaskMgr` 调度执行。
2. **执行多副本拷贝任务：** `ObCopySSTableTask::execute` 是多副本拷贝任务的执行入口。实际上，这个任务会发送 `copy_sstable_batch` 的 RPC 请求，将构建好的基线数据批量拷贝到其他副本。
3. **RPC处理入口：** 收到 `copy_sstable_batch` 的 RPC 请求的是 `obs`，其执行入口是 `ObService::copy_sstable_batch`。在这个阶段，目标 `obs` 接收到了源 `obs` 发送过来的基线数据。
4. **汇报多副本拷贝结果：** 完成基线数据的拷贝后，源 `obs` 向 `rs` 汇报多副本拷贝任务的结果。这个结果包括拷贝是否成功、失败的详细信息等。
5. **回调更新状态：** `rs` 在接收到多副本拷贝任务的结果后，执行回调函数 `ObGlobalIndexBuilder::on_copy_multi_replica_reply`，用于更新多副本拷贝任务的状态。这个回调会反映拷贝任务的执行情况，成功或失败等信息。

3.1.4 唯一性校验

1. **发起唯一性校验任务：** 对于唯一索引，需要校验索引列数据的唯一性。此任务由 `ObGlobalIndexBuilder::try_unique_index_calc_checksum` 触发。
2. **选取校验快照点：** 为了校验唯一性，需要选取一个快照点。在此快照点之后，对主表的 DML 操作都能看到索引表的基线数据。这个过程涉及等待所有副本的新事务的时间戳推过快照点。函数 `get_checksum_calculation_snapshot` 负责完成这一操作。
3. **发送RPC请求：** 获取到快照点后，发送RPC请求给主表和索引表的 leader，请求计算在该快照点之前的主表和索引表列的校验和。这个 RPC 请求在 `ObService::calc_column_checksum_request` 处理。
4. **计算校验和：** 主表和索引表的 leader 在收到请求后计算该快照点的列校验和，计算完成后将结果记录在内部表 `__all_index_checksum` 中，并通过RPC通知 `rs`。RPC 的回调函数为 `ObGlobalIndexBuilder::on_col_checksum_calculation_reply`，用于更新校验和计算任务的状态。
5. **检查任务状态：** 通过 `drive_this_unique_index_calc_checksum` 持续检查校验和计算任务的状态。如果所有的校验和计算任务都完成，则进入下一步。
6. **执行校验：** 通过 `ObGlobalIndexBuilder::try_unique_index_check -> ObIndexChecksumOperator::check_column_checksum` 执行校验和的比对。这一步是通过比对主表和索引表列的校验和来校验唯一性，确保在快照点之前的数据的唯一性。

3.1.5 索引状态变更

如果以上步骤全部成功，通过 `ObGlobalIndexBuilder::try_handle_index_build_take_effect` 函数使索引生效。具体操作是修改索引表的 schema 状态为 `INDEX_STATUS_AVAILABLE`。中控的 OBS 在检测到该状态后，向客户端的会话（session）返回成功的信息，表示索引已成功生效。

3.1.6 中间结果清理

1. **触发索引构建结束处理：** 在索引构建完成后，无论是成功还是失败，通过 `ObGlobalIndexBuilder::try_handle_index_build_finish` 触发索引构建结束的处理流程。
2. **清理中间结果：** 调用 `clear_intermediate_result` 函数，该函数位于 `ObIndexSSTableBuilder` 中，负责清理 SQL 执行的中间结果，包括释放索引构建过程中产生的临时数据等。
3. **释放快照：** 在清理过程中，执行 `release_snapshot` 操作，该操作用于释放索引构建时所占用的快照。释放快照是为了确保在索引构建完成后不再占用额外资源。
4. **清理内部表：** `ObIndexSSTableBuilder::clear_interm_result` 中包含对内部表的清理操作，确保在索引构建结束后没有残留的临时数据或状态。

3.2 局部索引实现原理流程

局部索引构建的整体流程跟全局索引类似，也是先等事务结束，拿到快照点，然后选择一个副本做单副本构建，等单副本构建完成后，拷贝基线数据到其他副本，然后（对唯一索引）做唯一性检查，之后索引生效。其中基线数据的构建通过ObBuildIndexDag完成，唯一性的检查通过ObUniqueCheckingDag完成。

生成全局索引的控制任务：

1. **生成索引表的schema：** 通过 `ObIndexBuilder::do_create_local_index` 调用 `ObIndexBuilder::generate_schema` 函数，生成局部索引表的schema。这个过程与全局索引生成 schema 的逻辑基本相同，主要区别在于局部索引不需要生成索引表的位置信息。
2. **创建内存对象：** 在生成索引表的 schema 后，调用相应的DDL服务函数，如 `ObDDLService::create_user_table`、`ObDDLService::create_table_in_trans`、`ObDDLService::create_table` 完成内存对象的创建。这一过程确保索引表的内存结构在数据库中得以建立。
3. **发布 schema：** 通过 `ObDDLService::publish_schema` 将生成的索引表 schema 发布，通知其他机器刷新 schema。这是为了确保集群中所有节点都能感知到新的索引表的存在。
4. **提交局部索引的控制任务：** 通过 `ObIndexBuilder::submit_build_local_index_task` 调用 `ObRSBuildIndexScheduler::push_task` 将局部索引的控制任务 `ObRSBuildIndexTask` 放入队列中。同时，更新内部表 `__all_index_build_stat`，记录相关索引构建的状态。
5. **执行局部索引的控制任务：** 由 `ObDDLTaskExecutor` 负责执行局部索引的控制任务。该 executor 只有一个线程，队列长度受限于内存。任务执行的入口在 `ObDDLTaskExecutor::run1` 中，调用 `ObRSBuildIndexTask::process` 函数进行任务处理。

4 OceanBase 索引功能不断发展的新特性

MySQL 在业内是最受欢迎的关系数据库之一，不少用户将 MySQL 作为刚开始使用数据库的首选。OceanBase 的一大重要特性即是与 MySQL 完全兼容，用户无需修改代码即可完成数据库的升级迁移，也大幅降低了开发者的学习成本。

OceanBase 4.1 对 MySQL 的兼容策略是完全兼容 5.7，同时支持 8.0 功能。相较于 MySQL 5.7，8.0 版本在性能、安全性、可用性等方面都有显著提升，同时在索引部分也新增了许多功能特性。

4.1 索引管理：更高的索引管理效率

MySQL 8.0的引入标志着数据库管理的一大进步，尤其是在索引可见性方面。这个新功能允许用户在保持索引结构的同时，可以将索引设置为不可见（invisible）状态。在这种状态下，查询优化器不会考虑这些索引来制定查询计划，但是索引数据仍然会被正常维护。这一特性的好处在于它允许用户在不删除索引的情况下测试该索引对查询性能的影响。如果需要，用户可以随时将索引设置回可见（visible）状态。

例如，在考虑删除某个索引时，我们可以先将其设置为invisible。这样，我们可以在实际业务运行一段时间后，确认这个索引是否真的不再需要。如果发现某个业务SQL查询依赖于这个索引，我们可以简单地将其改回visible。由于索引数据一直在维护，这种设置的速度非常快，并且可以避免因误删除索引而需要重新创建它所带来的开销。

在MySQL 8.0中，设置索引可见性的语法如下：

- 将索引设置为不可见:

```
ALTER TABLE table_name ALTER INDEX index_name INVISIBLE;
```

- 将索引设置为可见:

```
ALTER TABLE table_name ALTER INDEX index_name VISIBLE;
```

OceanBase，从1.x版本开始，也支持这种索引可见性的设置。这使得用户可以在不删除或禁用索引的情况下，测试和调整索引策略对查询性能的影响。同时，由于索引本身需要维护并消耗资源，遗留的不必要索引可能会影响性能。通过设置索引可见性，用户可以安全地验证索引对查询性能的影响，避免资源浪费。此外，这一功能还能在不重建索引的情况下快速恢复索引，避免因误删关键索引而导致的性能下降风险。总的来说，索引可见性设置为用户提供了一种更灵活的方式来管理索引策略，优化查询性能，同时降低资源浪费，从而提高数据库管理的效率。

此外，MySQL 8.0还引入了对逆序索引的支持。逆序索引允许索引列被指定为降序排列，从而支持某些按降序排序的查询。例如，`ORDER BY c1 DESC, c2` 的查询可以利用以下索引：

```
CREATE TABLE t1 (c1 INT, c2 INT, INDEX i1 (c1 DESC, c2 ASC));
```

目前，OceanBase还不支持逆序索引功能。对于需要逆序排序的场景，OceanBase的优化器会利用正序索引进行逆序扫描，并支持前缀排序和通过并行执行（PX）加速排序。在未来的版本中，OceanBase计划支持逆序索引。对于混合正逆序的多列排序，例如索引为 `index1 (c1, c2, c3)` 且排序为 `ORDER BY c1 DESC, c2 ASC, c3 ASC` 的场景，OceanBase会选择逆序扫描index1索引，然后只对c2和c3进行排序。此外，如果建立索引后排序性能仍不满意，可以通过并行执行（PX）来加速排序过程。

4.2 OceanBase 4.1中的GV$SESSION_LONGOPS视图

OceanBase 4.1 版本开始，您可以通过 `GV$SESSION_LONGOPS` 视图查看数据库内部耗时任务。索引创建过程是一个可能耗时较久的过程，索引创建所处的阶段、进度信息会维护在此视图。

视图 `GV$SESSION_LONGOPS` 字段含义如下。

列名	类型	说明
SID	bigint(20)	Session ID
OPNAME	varchar(128)	具体 DDL 操作名
TARGET	varchar(128)	DDL操作对象
SVR_IP	varchar(46)	机器 IP
SVR_PORT	bigint(20)	机器 Port
START_TIME	bigint(20)	开始时间
ELAPSED_SECONDS	decimal(24,4)	已消耗时间
TIME_REMAINING	bigint(20)	预估剩余时间
LAST_UPDATE_TIME	bigint(20)	记录更新时间
MESSAGE	varchar(500)	补充信息
TRACE_ID	varchar(64)	Trace ID

观测索引创建过程

建索引的主要时间一般发生在索引数据补全，在某些环境，可能因为建索引的时候，还有并发的事务未执行完成，也有可能在等待事务结束部分，我们的进度观测例子中主要展示下这两个部分的进度。

等待事务结束阶段

索引创建任务处于等待事务结束阶段样例

```
obclient> select * from oceanbase.gv$session_longops \G
*************************** 1. row ***************************
              SID: -1
         TRACE_ID: YE5186458A15C-0005EF63AE10FBBB-0-0
           OPNAME: create index
           TARGET: __idx_500005_i1
           SVR_IP:
         SVR_PORT: 58648
       START_TIME: 2023-06-09
 ELAPSED_SECONDS: 7
  TIME_REMAINING: 0
LAST_UPDATE_TIME: 2023-06-09
          MESSAGE: TENANT_ID: 1004, TASK_ID: 2, STATUS: WAIT TRANS END,
PENDING_TX_ID: 76
```

处于等事务结束阶段，我们在 MESSAGE 字段中会展示 `WAIT TRANS END`，并且会将获取到的第一个未结束的事务 ID 展示出来（即上述样例 `PENDING_TX_ID: 76`），我们可以进一步通过 `__all_virtual_trans_stat` 表的 `trans_id` 字段来查询对应的事务信息。

数据补全阶段

您可以通过行数统计指标判断索引创建任务的整体进度，本章通过一个查询样例解释字段和进度指标。

索引创建任务处于数据补全阶段样例

```
obclient> select * from oceanbase.gv$session_longops \G
*************************** 1. row ***************************
              SID: -1
         TRACE_ID: YFDE80BA2DA8D-0005FDA8116C1F4E-0-0
          OPNAME: create index
           TARGET: 500097
           SVR_IP: 127.1
         SVR_PORT: 65000
       START_TIME: 2023-06-09 17:10:42
  ELAPSED_SECONDS: 5
   TIME_REMAINING: 0
LAST_UPDATE_TIME: 2023-06-09 17:10:48
          MESSAGE: TENANT_ID: 1004, TASK_ID: 541, STATUS: REPLICA BUILD,
ROW_SCANNED: 2000000, ROW_SORTED: 4000000, ROW_INSERTED: 503316
1 row in set (0.03 sec)
```

`GV$SESSION_LONGOPS` 字段含义

对于索引创建任务而言， 各字段含义如下

- TRACE_ID: observer程序日志的ID，可以用该ID来搜索相关的日志文件
- OPNAME：建索引时，会展示 `create index`
- TARGET：建索引时，展示正在创建的索引名
- SVR_IP: 调度任务在哪个 OBServer 执行
- SVR_PORT：调度任务在哪个 OBServer 执行
- START_TIME：索引构建开始时间，这里只精确到日期，跟Oracle是兼容的
- ELAPSED_SECONDS: 索引构建执行的时间，单位为秒
- TIME_REMAINING: 剩余时间预测，兼容 Oracle，暂时还没有实现
- LAST_UPDATE_TIME: 统计信息收集的时间，精确到日期，兼容 Oracle
- **MESSAGE**：里面包含了索引任务的具体信息，最重要的字段！

MESSAGE 字段信息详细说明

索引创建任务 MESSAGE 内部字段说明

- TENANT_ID 为租户ID
- TASK_ID 为 DDL 的任务ID
- STATUS 为 DDL 执行到的状态，如 `REPLICA BUILD` 指的是数据补全阶段

4.3 OceanBase v4.2函数索引调整特性

4.3.1 OceanBase 4.2在MySQL模式下支持函数索引功能

OceanBase 4.1 以及之前的版本中，已在Oracle模式下支持了函数索引功能。OceanBase 4.2 在MySQL模式支持函数索引功能，兼容MySQL 8.0。

示例1：使用create index语句创建函数索引

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
    ON tbl_name (expr,...)
    [index_option] ...

key_part: (expr) [ASC | DESC]
```

expr是一个合法的函数索引表达式，且允许是布尔表达式，例如"c1=c1"。与MySQL不同的是，OceanBase禁止在函数索引的定义中引用生成列。

例如以下语句在t1_func表上创建了一个索引定义是c1+c2 < 1的函数索引i1

```
create table t1_func(c1 int, c2 int);
create index i1 on t1_func ((c1+c2 < 1));
```

示例2：使用alter table语句创建函数索引

```
ALTER TABLE tbl_name
    [alter_option [, alter_option] ...]
    [partition_options]

alter_option: {
    table_options
  | ADD {INDEX | KEY} [index_name]
        [index_type] (key_part,...) [index_option] ...
  | ADD SPATIAL [INDEX | KEY] [index_name]
        (key_part,...) [index_option] ...
  | ...

key_part: (expr) [ASC | DESC]
```

例如以下语句在t1_func上添加了3个函数索引，其中一个名字是i2，另外两个由系统自动生成的名称，格式为'functional_index'前缀加编号。

```
alter table t1_func add index ((concat(c1,'a')));
alter table t1_func add index ((c1+1));
alter table t1_func add index i2 ((concat(c1,'a')));
```

示例3：使用create table语句在建表时创建函数索引

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]
```

```
create_definition: {
    col_name column_definition
  | {INDEX | KEY} [index_name] [index_type] (key_part,...)
      [index_option] ...
  | SPATIAL [INDEX | KEY] [index_name] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
      [index_name] [index_type] (key_part,...)
      [index_option] ...
  ...
}

key_part: (expr) [ASC | DESC]
```

对函数索引支持的总结

Oceanbase 4.2版本支持了在MySQL模式下创建和使用函数索引，并且禁止了部分非确定性函数用于创建函数索引和生成列以提升稳定性。但是其中部分函数在MySQL和Oracle中是允许用于函数索引的，例如：

```
(Mysql 8.0.31)
mysql> create table time_func(c1 timestamp, c2 date);
Query OK, 0 rows affected (0.04 sec)

mysql> create index i1 on time_func((date(c1)));
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Oceanbase后续版本将会放开对这些函数的限制，进一步提升兼容性。