Mysql 技术内幕：InnoDB 学习（5）

jask

2024-08-11

Mysql 技术内幕

索引和算法

概述

InnoDB 支持：B+ 树索引，全文索引，哈希索引。

InnoDB 存储引擎支持的哈希索引是自适应的，InnoDB 存储引擎会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引。

B+ 树索引就是传统意义上的索引，这是目前关系型数据库系统中查找最为常用和最为有效的索引。B+ 树索引的构造类似于二叉树，根据键值（Key Value）快速找到数据。

B+ 树索引并不能找到一个给定键值的具体行。B+ 树索引能找到的只是被查找数据行所在的页。然后数据库通过把页读入到内存，再在内存中进行查找，最后得到要查找的数据。

数据结构

二分查找法 二分查找法（binary search）也称为折半查找法，用来查找一组有序的记录数组中的某一记录，其基本思想是：将记录按有序化（递增或递减）排列，在查找过程中采用跳跃式方式查找，即先以有序数列的中点位置为比较对象，如果要找的元素值小于该中点元素，则将待查序列缩小为左半部分，否则为右半部分。通过一次比较，将查找区间缩小一半。

二叉查找树，平衡二叉树 B+ 树是通过二叉查找树，再由平衡二叉树，B 树演化而来。相信在任何一本有关数据结构的书中都可以找到二叉查找树的章节，二叉查找树是一种经典的数据结构。图 5-2 显示了一棵二叉查找树。

平衡二叉树的定义如下：首先符合二叉查找树的定义，其次必须满足任何节点的两个子树的高度最大差为 1。显然，图 5-3 不满足平衡二叉树的定义，而图 5-2 是一棵平衡二叉树。平衡二叉树的查找性能是比较高的，但不是最高的，只是接近最高性能。最好的性能需要建立一棵最优二叉树，但是最优二叉树的建立和维护需要大量的操作，因此，用户一般只需建立一棵平衡二叉树即可。

B+ 树

B+ 树由 B 树和索引顺序访问方法演化而来。

B+ 树是为磁盘或其他直接存取辅助设备设计的一种平衡查找树。在 B+ 树中，所有记录节点都是按键值的大小顺序存放在同一层的叶子节点上，由各叶子节点指针进行连接。

B+ 树的插入操作 B+ 树的插入必须保证插入后叶子节点中的记录依然排序，同时需要考虑插入到 B+ 树的三种情况，每种情况都可能会导致不同的插入算法。

表 5-1 B+ 树插入的 3 种情况

Leaf Page 满	Index Page 满	操作
No	No	直接将记录插入到叶子节点
Yes	No	1）拆分 Leaf Page 2）将中间的节点放入到 Index Page 中 3）小于中间节点的记录放左边 4）大于或等于中间节点的记录放右边
Yes	Yes	1）拆分 Leaf Page 2）小于中间节点的记录放左边 3）大于或等于中间节点的记录放右边 4）拆分 Index Page 5）小于中间节点的记录放左边 6）大于中间节点的记录放右边 7）中间节点放入上一层 Index Page

Figure 1: 三种情况

不管怎么变化，B+ 树总是会保持平衡。但是为了保持平衡对于新插入的键值可能需要做大量的拆分页（split）操作。因为 B+ 树结构主要用于磁盘，页的拆分意味着磁盘的操作，所以应该在可能的情况下尽量减少页的拆分操作。因此，B+ 树同样提供了类似于平衡二叉树的旋转（Rotation）功能。

旋转发生在 Leaf Page 已经满，但是其的左右兄弟节点没有满的情况下。这时 B+ 树并不会急于去做拆分页的操作，而是将记录移到所在页的兄弟节点上。在通常情况下，左兄弟会被首先检查用来做旋转操作，因此再来看图 5-7 的情况，若插入键值 70，其实 B+ 树并不会急于去拆分叶子节点，而是去做旋转操作。

B+ 树的删除操作 B+ 树使用填充因子（fill factor）来控制树的删除变化，50% 是填充因子可设的最小值。B+ 树的删除操作同样必须保证删除后叶子节点中的记录依然排序，同插入一样，B+ 树的删除操作同样需要考虑以下表 5-2 中的三种情况，与插入不同的是，删除根据填充因子的变化来衡量。

B+ 树索引

前面讨论的都是 B+ 树的数据结构及其一般操作，B+ 树索引的本质就是 B+ 树在数据库中的实现。但是 B+ 索引在数据库中有一个特点是高扇出性，因此在数据库中，B+ 树的高度一般都在 2～4 层，这也就是说查找某一键值的行记录时最多只需要 2 到 4 次 IO，这倒不错。因为当前一般的机械磁盘每秒至少可以做 100 次 IO，2～4 次的 IO 意味着查询时间只需 0.02～0.04 秒。

表 5-2　B+ 树删除操作的三种情况

叶子节点小于填充因子	中间节点小于填充因子	操作
No	No	直接将记录从叶子节点删除，如果该节点还是 Index Page 的节点，用该节点的右节点代替
Yes	No	合并叶子节点和它的兄弟节点，同时更新 Index Page
Yes	Yes	1）合并叶子节点和它的兄弟节点 2）更新 Index Page 3）合并 Index Page 和它的兄弟节点

Figure 2: 删除操作的三种情况

数据库中的 B+ 树索引可以分为聚集索引（clustered inex）和辅助索引（secondary index），但是不管是聚集还是辅助的索引，其内部都是 B+ 树的，即高度平衡的，叶子节点存放着所有的数据。聚集索引与辅助索引不同的是，叶子节点存放的是否是一整行的信息。

聚集索引　InnoDB 存储引擎表是索引组织表，即表中数据按照主键顺序存放。而聚集索引（clustered index）就是按照每张表的主键构造一棵 B+ 树，同时叶子节点中存放的即为整张表的行记录数据，也将聚集索引的叶子节点称为数据页。聚集索引的这个特性决定了索引组织表中数据也是索引的一部分。同 B+ 树数据结构一样，每个数据页都通过一个双向链表来进行链接。

由于实际的数据页只能按照一棵 B+ 树进行排序，因此每张表只能拥有一个聚集索引。在多数情况下，查询优化器倾向于采用聚集索引。因为聚集索引能够在 B+ 树索引的叶子节点上直接找到数据。此外，由于定义了数据的逻辑顺序，聚集索引能够特别快地访问针对范围值的查询。查询优化器能够快速发现某一段范围的数据页需要扫描。

聚集索引的另一个好处是，它对于主键的排序查找和范围查找速度非常快。叶子节点的数据就是用户所要查询的数据。如用户需要查询一张注册用户的表，查询最后注册的 10 位用户，由于 B+ 树索引是双向链表的，用户可以快速找到最后一个数据页，并取出 10 条记录。

另一个是范围查询（range query），即如果要查找主键某一范围内的数据，通过叶子节点的上层中间节点就可以得到页的范围，之后直接读取数据页即可。

辅助索引　对于辅助索引（Secondary Index，也称非聚集索引），叶子节点并不包含行记录的全部数据。叶子节点除了包含键值以外，每个叶子节点中的索引行中还包含了一个书签（bookmark）。该书签用来告诉 InnoDB 存储引擎哪里可以找到与索引相对应的行数据。由于 InnoDB 存储引擎表是索引组织表，因此 InnoDB 存储引擎的辅助索引的书签就是相应行数据的聚集索引键。图 5-15 显示了 InnoDB 存储引擎中辅助索引与聚集索引的关系。

辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。当通过辅助索引来寻找数据时，InnoDB 存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引来找到一个完整的行记录。举例来说，如果在一棵高度为 3 的辅助索引树中查找数据，那需要对这棵辅助索引树遍历 3 次找到指定主键，如果聚集索引树的高度同样为 3，那么还需要对聚集索引树进行 3 次查找，最终找到一个完整的行数据所在的页，因此一共需要 6 次逻辑 IO 访问以得到最终的一个数据页。

对于其他的一些数据库，如 Microsoft SQL Server 数据库，其有一种称为堆表的表类

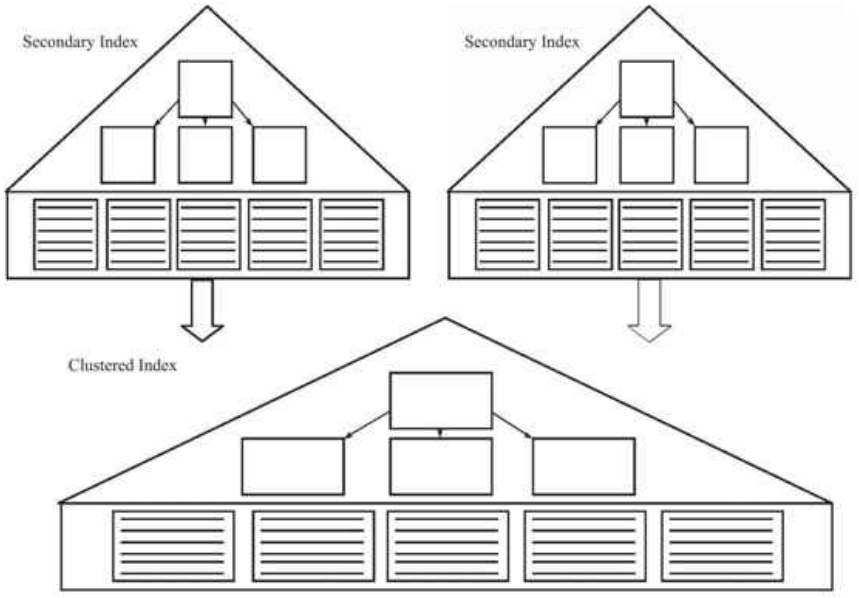

Figure 3: 两种索引的关系

型，即行数据的存储按照插入的顺序存放。这与 MySQL 数据库的 MyISAM 存储引擎有些类似。堆表的特性决定了堆表上的索引都是非聚集的，主键与非主键的区别只是是否唯一且非空（NOT NULL）。因此这时书签是一个行标识符（Row Identifiedr，RID），可以用如“文件号：页号：槽号”的格式来定位实际的行数据。

有的 Microsoft SQL Server 数据库 DBA 问过我这样的问题，为什么在 Microsoft SQL Server 数据库上还要使用索引组织表？堆表的书签使非聚集查找可以比主键书签方式更快，并且非聚集可能在一张表中存在多个，我们需要对多个非聚集索引进行查找。而且对于非聚集索引的离散读取，索引组织表上的非聚集索引会比堆表上的聚集索引慢一些。

B+ 树索引的分裂 B+ 树索引页的分裂并不总是从页的中间记录开始，这样可能会导致页空间的浪费。

例如下面的记录：
1,2,3,4,5,6,7,8,9

插入是根据自增顺序进行的，若这时插入 10 这条记录后需要进行页的分裂操作，会将记录 5 作为分裂点记录 (split record)，分裂后得到：
p1: 1,2,3,4
p2: 5,6,7,8,8,10
然而由于插入是顺序的，P1 这个页中将不会再有记录被插入，从而导致空间的浪费。而 P2 又会再次进行分裂。

InnoDB 存储引擎的 Page Header 中有以下几个部分用来保存插入的顺序信息：

PAGE_LAST_INSERT

PAGE_DIRECTION

PAGE_N_DIRECTION

通过这些信息，InnoDB 存储引擎可以决定是向左还是向右进行分裂，同时决定将分裂点记录为哪一个。若插入是随机的，则取页的中间记录作为分裂点的记录。

B+ 树索引的管理

索引管理 1.ALTER 语句

2.CREATE/DROP INDEX

Fast Index Creation InnoDB 存储引擎从 InnoDB 1.0.x 版本开始支持一种称为 Fast Index Creation（快速索引创建）的索引创建方式——简称 FIC。

对于辅助索引的创建，InnoDB 存储引擎会对创建索引的表加上一个 S 锁。在创建的过程中，不需要重建表，因此速度较之前提高很多，并且数据库的可用性也得到了提高。删除辅助索引操作就更简单了，InnoDB 存储引擎只需更新内部视图，并将辅助索引的空间标记为可用，同时删除 MySQL 数据库内部视图上对该表的索引定义即可。

这里需要特别注意的是，临时表的创建路径是通过参数 tmpdir 进行设置的。用户必须保证 tmpdir 有足够的空间可以存放临时表，否则会导致创建索引失败。

由于 FIC 在索引的创建的过程中对表加上了 S 锁，因此在创建的过程中只能对该表进行读操作，若有大量的事务需要对目标表进行写操作，那么数据库的服务同样不可用。此外，FIC 方式只限定于辅助索引，对于主键的创建和删除同样需要重建一张表。

Online Schema Change Online Schema Change（在线架构改变，简称 OSC）最早是由 Facebook 实现的一种在线执行 DDL 的方式，并广泛地应用于 Facebook 的 MySQL 数据库。所谓“在线”是指在事务的创建过程中，可以有读写事务对表进行操作，这提高了原有 MySQL 数据库在 DDL 操作时的并发性。

Facebook 采用 PHP 脚本来现实 OSC，而并不是通过修改 InnoDB 存储引擎源码的方式。

实现 OSC 步骤如下：

init，即初始化阶段，会对创建的表做一些验证工作，如检查表是否有主键，是否存在触发器或者外键等。

createCopyTable，创建和原始表结构一样的新表。

alterCopyTable：对创建的新表进行 ALTER TABLE 操作，如添加索引或列等。createDeltasTable，创建 deltas 表，该表的作用是为下一步创建的触发器所使用。之后对原表的所有 DML 操作会被记录到 createDeltasTable 中。

createTriggers，对原表创建 INSERT、UPDATE、DELETE 操作的触发器。触发操作产生的记录被写入到 deltas 表。

startSnpshotXact，开始 OSC 操作的事务。

selectTableIntoOutfile，将原表中的数据写入到新表。为了减少对原表的锁定时间，这里通过分片（chunked）将数据输出到多个外部文件，然后将外部文件的数据导入到 copy 表中。分片的大小可以指定，默认值是 500 000。

dropNCIndexs，在导入到新表前，删除新表中所有的辅助索引。

loadCopyTable，将导出的分片文件导入到新表。

replayChanges，将 OSC 过程中原表 DML 操作的记录应用到新表中，这些记录被保存在 deltas 表中。

recreateNCIndexes，重新创建辅助索引。

replayChanges，再次进行 DML 日志的回放操作，这些日志是在上述创建辅助索引中过程中新产生的日志。

swapTables，将原表和新表交换名字，整个操作需要锁定 2 张表，不允许新的数据产生。由于改名是一个很快的操作，因此阻塞的时间非常短。

Online DDL 虽然 FIC 可以让 InnoDB 存储引擎避免创建临时表，从而提高索引创建的效率。但正如前面小节所说的，索引创建时会阻塞表上的 DML 操作。OSC 虽然解决了上述的部分问题，但是还是有很大的局限性。MySQL 5.6 版本开始支持 Online DDL（在线数据定义）操作，其允许辅助索引创建的同时，还允许其他诸如 INSERT、UPDATE、DELETE 这类 DML 操作，这极大地提高了 MySQL 数据库在生产环境中的可用性。

Cardinality 值

What is Cardinality 怎样查看索引是否是高选择性的呢？可以通过 SHOW INDEX 结果中的列 Cardinality 来观察。Cardinality 值非常关键，表示索引中不重复记录数量的预估值。同时需要注意的是，Cardinality 是一个预估值，而不是一个准确值，基本上用户也不可能得到一个准确的值。在实际应用中，Cardinality/n_rows_in_table 应尽可能地接近 1。如果非常小，那么用户需要考虑是否还有必要创建这个索引。故在访问高选择性属性的字段并从表中取出很少一部分数据时，对这个字段添加 B+ 树索引是非常有必要的。

InnoDB 的 Cardinality 统计 建立索引的前提是列中的数据是高选择性的，这对数据库来说才具有实际意义。然而数据库是怎样来统计 Cardinality 信息的呢？因为 MySQL 数据库中有各种不同的存储引擎，而每种存储引擎对于 B+ 树索引的实现又各不相同，所以对 Cardinality 的统计是放在存储引擎层进行的。

在 InnoDB 存储引擎中，Cardinality 统计信息的更新发生在两个操作中：INSERT 和 UPDATE。根据前面的叙述，不可能在每次发生 INSERT 和 UPDATE 时就去更新 Cardinality 信息，这样会增加数据库系统的负荷，同时对于大表的统计，时间上也不允许数据库这样去操作。因此，InnoDB 存储引擎内部对更新 Cardinality 信息的策略为：

B+ 索引树的使用

不同应用中 B+ 索引树的使用 在 OLTP 应用中，查询操作只从数据库中取得一小部分数据，一般可能都在 10 条记录以下，甚至在很多时候只取 1 条记录，如根据主键值来取得用户信息，根据订单号取得订单的详细信息，这都是典型 OLTP 应用的查询语句。在这种情况下，B+ 树索引建立后，对该索引的使用应该只是通过该索引取得表中少部分的数据。这时建立 B+ 树索引才是有意义的，否则即使建立了，优化器也可能选择不使用索引。

对于 OLAP 应用，情况可能就稍显复杂了。不过概括来说，在 OLAP 应用中，都需要访问表中大量的数据，根据这些数据来产生查询的结果，这些查询多是面向分析的查询，目的是为决策者提供支持。如这个月每个用户的消费情况，销售额同比、环比增长的情况。因此在 OLAP 中索引的添加根据的应该是宏观的信息，而不是微观，因为最终要得到的结果是提供给决策者的。例如不需要在 OLAP 中对姓名字段进行索引，因为很少需要对单个用户进行查询。但是对于 OLAP 中的复杂查询，要涉及多张表之间的联接操作，因此索引的添加依然是有意义的。但是，如果联接操作使用的是 Hash Join，那么索引可能又变得不是非常重要了。

联合索引 本质上来说，联合索引也是一棵 B+ 树，不同的是联合索引的键值的数量不是 1，而是大于等于 2。

覆盖索引 InnoDB 存储引擎支持覆盖索引（covering index，或称索引覆盖），即从辅助索引中就可以得到查询的记录，而不需要查询聚集索引中的记录。使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的 IO 操作。

覆盖索引的另一个好处是对某些统计问题而言的。

InnoDB 存储引擎并不会选择通过查询聚集索引来进行统计。由于 buy_log 表上还有辅助索引，而辅助索引远小于聚集索引，选择辅助索引可以减少 IO 操作。

优化器选择不使用索引的情况 因此对于不能进行索引覆盖的情况，优化器选择辅助索引的情况是，通过辅助索引查找的数据是少量的。这是由当前传统机械硬盘的特性所决定的，即利用顺序读来替换随机读的查找。若用户使用的磁盘是固态硬盘，随机读操作非常快，同时有足够的自信来确认使用辅助索引可以带来更好的性能，那么可以使用关键字 FORCE INDEX 来强制使用某个索引。

索引提示 以下两种情况可能需要用到 INDEX HINT：

MySQL 数据库的优化器错误地选择了某个索引，导致 SQL 语句运行的很慢。这种情况在最新的 MySQL 数据库版本中非常非常的少见。优化器在绝大部分情况下工作得都非常有效和正确。这时有经验的 DBA 或开发人员可以强制优化器使用某个索引，以此来提高 SQL 运行的速度。

某 SQL 语句可以选择的索引非常多，这时优化器选择执行计划时间的开销可能会大于 SQL 语句本身。例如，优化器分析 Range 查询本身就是比较耗时的操作。这时 DBA 或开发人员分析最优的索引选择，通过 Index Hint 来强制使优化器不进行各个执行路径的成本分析，直接选择指定的索引来完成查询。

Multi-Range Read 优化 MySQL5.6 版本开始支持 Multi-Range Read（MRR）优化。Multi-Range Read 优化的目的就是为了减少磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问，这对于 IO-bound 类型的 SQL 查询语句可带来性能极大的提升。Multi-Range Read 优化可适用于 range，ref，eq_ref 类型的查询。

MRR 优化有以下几个好处：

MRR 使数据访问变得较为顺序。在查询辅助索引时，首先根据得到的查询结果，按照主键进行排序，并按照主键排序的顺序进行书签查找。

减少缓冲池中页被替换的次数。

批量处理对键值的查询操作。

于 InnoDB 和 MyISAM 存储引擎的范围查询和 JOIN 查询操作，MRR 的工作方式如下：

将查询得到的辅助索引键值存放于一个缓存中，这时缓存中的数据是根据辅助索引键值排序的。

将缓存中的键值根据 RowID 进行排序。

根据 RowID 的排序顺序来访问实际的数据文件。

Index Condition Pushdown（ICP）优化 在支持 Index Condition Pushdown 后，MySQL 数据库会在取出索引的同时，判断是否可以进行 WHERE 条件的过滤，也就是将 WHERE 的部分过滤操作放在了存储引擎层。在某些查询下，可以大大减少上层 SQL 层对记录的索取（fetch），从而提高数据库的整体性能。

若支持 Index Condition Pushdown 优化，则在索引取出时，就会进行 WHERE 条件的过滤，然后再去获取记录。这将极大地提高查询的效率。Where 可以过滤的条件是要该索引可以覆盖到的范围。

哈希算法

哈希表 哈希表技术很好地解决了直接寻址遇到的问题，但是这样做有一个小问题，两个关键字可能映射到同一个槽上。一般将这种情况称之为发生了碰撞（collision）。在数据库中一般采用最简单的碰撞解决技术，这种技术被称为链接法（chaining）。

InnoDB 中的哈希算法

自适应哈希索引 自适应哈希索引经哈希函数映射到一个哈希表中，因此对于字典类型的查找非常快速，如 SELECT*FROM TABLE WHERE index_col= 'xxx'。但是对于范围查找就无能为力了。

全文检索

全文检索（Full-Text Search）是将存储于数据库中的整本书或整篇文章中的任意内容信息查找出来的技术。

倒排索引 全文检索通常使用倒排索引（inverted index）来实现。倒排索引同 B+ 树索引一样，也是一种索引结构。它在辅助表（auxiliary table）中存储了单词与单词自身在一个或多个文档中所在位置之间的映射。这通常利用关联数组实现，其拥有两种表现形式：

inverted file index，其表现形式为 {单词，单词所在文档的 ID}

full invertedindex，其表现形式为 {单词，(单词所在文档的 ID，在具体文档中的位置)}

InnoDB 全文索引 在 InnoDB 存储引擎中，将 (DocumentId，Position) 视为一个 “ilist”。因此在全文检索的表中，有两个列，一个是 word 字段，另一个是 ilist 字段，并且在 word 字段上有设有索引。此外，由于 InnoDB 存储引擎在 ilist 字段中存放了 Position 信息，故可以进行 Proximity Search。

倒排索引需要将 word 存放到一张表中，这个表称为 Auxiliary Table（辅助表）。在 InnoDB 存储引擎中，为了提高全文检索的并行性能，共有 6 张 Auxiliary Table，目前每张表根据 word 的 Latin 编码进行分区。

Auxiliary Table 是持久的表，存放于磁盘上。然而在 InnoDB 存储引擎的全文索引中，还有另外一个重要的概念 FTS Index Cache（全文检索索引缓存），其用来提高全文检索的性能。

FTS Index Cache 是一个红黑树结构，其根据（word，ilist）进行排序。这意味着插入的数据已经更新了对应的表，但是对全文索引的更新可能在分词操作后还在 FTS Index Cache 中，Auxiliary Table 可能还没有更新。InnoDB 存储引擎会批量对 Auxiliary Table 进行更新，而不是每次插入后更新一次 Auxiliary Table。当对全文检索进行查询时，Auxiliary Table 首先会将在 FTS Index Cache 中对应的 word 字段合并到 Auxiliary Table 中，然后再进行查询。这种 merge 操作非常类似之前介绍的 Insert Buffer 的功能，不同的是 Insert Buffer 是一个持久的对象，并且其是 B+ 树的结构。然而 FTS Index Cache 的作用又和 Insert Buffer 是类似的，它提高了 InnoDB 存储引擎的性能，并且由于其根据红黑树排序后进行批量插入，其产生的 Auxiliary Table 相对较小。

InnoDB 存储引擎允许用户查看指定倒排索引的 Auxiliary Table 中分词的信息，可以通过设置参数 innodb_ft_aux_table 来观察倒排索引的 Auxiliary Table。

全文检索 MATCH() …AGAINST() 语法支持全文检索的查询，MATCH 指定了需要被查询的列，AGAINST 指定了使用何种方法进行查询。