MySQL的聚簇和非聚簇索引&页分裂&页合并

后厂村村长 2023-06-06 北京 阅读 4 分钟

聚簇(聚集)索引

聚簇索引(InnoDB)是对磁盘上的数据重新组织以按指定的一个或多个列的值排序的算法，聚簇索引的叶子节点就是其**数据节点**，其特点是数据的存储顺序和索引顺序一致。
一般情况下默认以主键为聚簇索引，且一张表只允许存在一个聚簇索引，因为，**数据一旦存储，其顺序只能有一种**。
如果未设置主键，会选择一个符合条件的 Unique-key 做主键；如果找不到符合条件的 Unique-key，InnoDB 会生成一个内部的`rowid`做主键;

非聚簇(聚集)索引
非聚簇索引(MyISAM)的叶子节点仍然是索引文件，只是这个索引文件中包含指向对应数据块的指针。

二者区别（援引自数据库原理一书）：
聚簇索引的叶子节点就是其数据节点，而非聚簇索引的叶子节点仍然是索引节点，该索引节点指向对应数据块的指针。

聚簇索引的优缺点：

优点: 根据主键查询条目比较少时，不用回表(数据就在主键节点下)；
缺点: 如果碰到不规则数据插入时，造成频繁的页分裂；

关于聚簇索引的优点：

InnoDB 的二级索引的叶子节点存放的是索引key值和主键值；因此执行类似 `select *` 查询全量字段时，会先通过二级索引定位到主键值，再根据查到的主键值执行一遍主键索引找到相应的数据块，这个过程叫做`回表`;
可见，回表的本质是两次二叉树查询，所以一般不建议使用 `select *`；
而如果查询的字段就是索引key本身，则称之为`覆盖索引`；
比如学生表里除了`学生ID`，还有`学生姓名、班级ID、专业ID`等，假设以 `学生ID、班级ID` 做了联合索引，那么当我们 `select 学生ID、班级ID from 班级表` 时，通过二级索引，就会找到我们需要的值，无需再使用主键回表，会大大节省时间。
MyISAM 的主键和二级索引的叶子节点，保存的都是其数据的物理地址；
因此，MyISAM，无论是用主键还是二级索引，都要回表查询；
也因这个特性，**MyISAM 可以不设主键**！于 MyISAM 而言，主键索引和二级索引，其实没啥区别，只不过主键因其定义的特殊性，需保证唯一、非空；

关于聚簇索引的缺点：

InnoDB 环境下，页分裂会发生在插入或更新，为满足索引逻辑，可能会产生频繁的页分裂，从而导致更新效率变低；
MyISAM 不存在页分裂的问题：只需在更新数据之后移动索引节点即可；

什么是页分裂?

我们要知道，InnoDB 不是按行来操作数据的，它可操作的最小单位是**页**，页加载进内存后才会通过扫描页来获取行记录。
比如查询`id=111`，是获取`111`所在的数据页，加载进内存后取出`111`这一行。
页的默认大小为16KB，64个连续的数据页称为一个extent（区），64个页组成一个区，所以区的大小为1MB(16*64=1024)，连续的256个数据区称为一组数据区；
两个数据页之间会有指针指向上一个和下一个数据页，形成一个双向链表，数据页中的每个数据行之间会有单向指针连接，组成一个单向链表；
当一个数据页中的数据行太多放不下的下，就会生成一个新的数据页来存储，同时使用双向链表来相连；

> 使用索引时，一个最基础的条件是，后面数据页中的数据行的主键值要大于前一个数据页中数据行的主键值；
> 至于原因，其实索引简单来说，就是一遍一遍过筛子，通过二分法的逻辑不断减少要筛选的数据，而真实数据是按主键顺序存储的，所以主键值就是筛选标准，以便尽快定位我们需要的数据；

我们假设如下的前两行数据已满足凑成一页的条件：
如果我们设置了主键ID自定义，非自增，在已经插入了`1、5、6、7`(已分为两页)等ID的情况下，再插入ID`2`，就会触发页分裂；
因为我们需要保证，**后一个数据页中的所有主键值要比前一个数据页中的主键值大**；
这时我我们需要把`2`插入到`1`的后面，`5、6、7`等ID依次后移；
注意，这里不是单纯的移动ID，而是要带着数据一起搬家！另外，每一行数据所占用的空间是不固定的，有可能移动之后，一页空间存不下`5、6、7`三条数据，需要同时生成第三页存放`7`ID;
如果第三页已经存在了咋办，那就得生成第N页，同时修改第二、三页和第N页的指针，调整到符合要求；
所以如果插入的主键是乱序，为满足索引条件，可能会产生频繁的页分裂，从而导致更新效率变低；
当然了，真正的页分裂要比上面所说的复杂很多，但本质是通过这种逻辑来完成页分裂的。

> 文档参考：`https://www.percona.com/blog/innodb-page-merging-and-page-splitting`
> 文档翻译参考：`https://zhuanlan.zhihu.com/p/98818611`

页分裂的大致流程

假设现在有 `9、10、11` 等3页数据，在页#10中插入了乱序的主键ID，或者更新了一条长文本数据导致当没有足够空间存储：

由于页#10没有足够空间去容纳新（或更新）的记录，根据“下一页”逻辑，记录应该由页#11负责。然而：页#11也同样满了，为了保证主键的顺序，也不可能乱序插入。这时候该怎么办呢?
B+树的每一层都是双向链表，页#10有指向页#9和页#11的指针，InnoDB的做法是（简化版）：

- 创建新页
- 判断当前页（页#10）可以从哪里进行分裂（记录行层面）
- 移动记录行
- 重新定义页之间的关系

新的页#12被创建(假设只存在#13页，#12页被合并后删除了)，页#11保持原样，只有页之间的关系发生了改变：

- Page #10 will have Prev=9 and Next=12
- Page #12 Prev=10 and Next=11
- Page #11 Prev=12 and Next=13

B树在水平方向的一致性仍然满足，然而，在物理上，页面的位置却是无序的，而且大概率会落到不同的区。
总结：页面拆分发生在插入或更新时，并会导致页的错位（dislocation，落入不同的区）。

InnoDB用INFORMATION_SCHEMA.INNODB_METRICS表来跟踪页的分裂数。可以查看其中的index_page_splits和index_page_reorg_attempts/successful统计。

一旦产生了分裂的页，MySQL自身(无人为干预)唯一将原先顺序恢复的办法就是新分裂出来的页因为低于合并阈值（merge threshold）被删掉。这时候InnoDB会使用页合并将数据重新合并回来。
人为干预的方式就是用 `OPTIMIZE` 重新整理表。这可能是个很重量级和耗时的过程，但通常是唯一将大量分散在不同区的页理顺的方法。

另一方面，在页的合并和分裂期间，InnoDB 会获取索引树的 x-latch。它可能会导致索引的锁争用（index latch contention）。
如果没有合并和拆分(有写操作)，称为“乐观”更新，只需使用读锁（S）。带有合并和分裂的操作则称为“悲观”更新，需使用写锁（X）。

衍生知识点：页合并

根据 `B-Tree` 的特性，InnoDB 可以自顶向下遍历，也可以在各叶子节点之间水平遍历，因为每个叶子节点都有一个指向下一条（顺序）记录页的指针；
例如，页-5 有指向 页-6 的指针，页-6 有指向前一页（5）的指针和后一页（7）的指针；
如果是基于自增主键进行插入，这种机制可以做到快速顺序扫描（如范围扫描），但如果你不仅插入还进行删除呢?

首先，删除一行记录时，实际上记录并没有被物理删除，只是被暂时标记（flaged）为删除，且它的空间变得允许被其他记录使用；
其次，当页中删除的记录达到 `MERGE_THRESHOLD（默认页体积的50%）` 时，InnoDB 会开始寻找最靠近的页（前或后）看看是否可以将两个页合并以优化空间使用；
如果第`N`页使用了不到一半的空间，第`N-1`页又达到了足够的删除数量，且同样处于50%使用以下，这时 InnoDB 会尝试将这两页合并；
合并后，第`N-1`页保留它之前的数据，并且容纳来自第`N`页的数据，同时，第`N`页变成一个空页，可以接纳新数据。

基于这个逻辑，我们可以得出，**如果 `UPDATE` 操作让页中数据体积达到上面的的阈值点，同样会触发页合并**；
所以，页合并发生在删除或更新操作中，关联到当前页的相邻页；
如果页合并成功，在`INFOMATION_SCHEMA.INNODB_METRICS`中的`index_page_merge_successful`将会增加。

问题延伸-1: 3层B+树的 InnoDB 可存储多少行数据?

InnoDB 构建`B+树`是以页为最小单位，参考上面的解释，所以我们可得出如下结论：

- 1、如果树的高度3，那么前2层存储的为 索引键值和页指针，第三层叶子节点才会存储数据；
- 2、根节点页如果为为默认值`16kb`，则为可存储的 索引键值和页指针 的上限；
  可执行 `show variables like 'innodb_page_size'` 查看当前库设置的 页大小；
- 3、指针占6字节(官方资料)，bigint类型的主键占8字节，加和共14字节，则根节点可存储的索引条数为：`16kb * 1024 / 14字节 = 1170`；
  我们假设一行数据占用`1kb`的空间，也就是一页可存放16条记录，这个设定是为了方便计算，实际的行大小可能远小于1kb；
  如果当前只有2层`B+树`，则可存储的行数为：`1170*16=18720`;
  现在拓展为3层`B+树`，根节点不再直接指向数据页，而是指向第二层索引页，即根节点的每一条索引指向一个16kb的索引页，则可存储的索引条数演变为`1170*1170`，即 1170 的2次方；
  由此可得出，3层`B+树`可存储：`1170*1170*16=21902400` ，约两千万条记录；
  继续延伸，按以上设定，N 层`B+树`可存储的记录数为: `1170的(N-1)次方 * 16`;

附，根节点 变动逻辑：

- 当为某个表创建一个B+树索引时，都会为这个索引创建一个 根节点 页；
- 最开始表中没数据时，每个B+树索引对应的 根节点 中既没数据记录，也没索引记录；
- 开始插入数据后，会先存储到根节点，当根节点空间被用完后，会将 根节点中所有记录复制到一个新的分配页，原本的 根节点 会变为存储记录页的索引目录页，整个过程 根节点 不会移动；
- 根节点 并不是向上创建目录页，而是将根节点自身变为目录页，原本根节点的数据会被复制一份到根节点下方，然后再在根节点上创建新的子节点存储新的记录；新插入的记录根据键值（聚簇索引中的主键值，二级索引中对应的索引列的值）的大小就会被分配到对应页中，而根节点便升级为存储目录项的记录页。

这就是 `B+树` 的演变过程。这个过程特别注意的是：一个B+树索引的根节点自诞生之日起，便不会再移动。这样只要我们对某个表建立一个索引，那么它的根节点的页号便会被记录到某个地方，然后凡是InnoDB存储引擎需要用到这个索引的时候，都会从那个固定的地方取出根节点的页号，从而来访问这个索引。

面试-Summary：

- InnoDB 的主索引文件中直接存放该行数据，称为聚簇索引，次索引(二级索引)指向对主键的引用；
- MyISAM 的主索引和次索引，都指向物理行地址(磁盘位置)；
- InnoDB(聚簇索引)的主键值最好是有序的，不仅能充分使用到索引，还尽可能避免了页分裂；否则就必须进行页分裂来保证索引的逻辑正确性；
- InnoDB 的主键，尽量使用连续增长的值，而不是随机值(比如随机字符串或UUID), 否则可能产生大量的页分裂；
- InnoDB的B+树索引注意事项：根页面的位置万年不动，一个页面最少存储2条记录。
- 聚簇索引的叶子节点存储的是行数据；而非聚簇索引叶子节点存储的是聚簇索引（通常是主键-ID）。
- 聚簇索引查询效率更高，而非聚簇索引需要进行回表查询，因此性能不如聚簇索引。
- 聚簇索引一个表中只能有一个，而非聚簇索引则没有数量上的限制。

mysql

阅读 1.6k • 发布于 2023-06-06

后厂村村长
4 声望 • 0 粉丝
Hello, Debug World

« 上一篇
为啥MySQL的InnoDB在一页(page)中最少要存储...

下一篇 »
HTTP状态码499，502，503，504原理及复现

引用和评论

推荐阅读

go-lang匿名函数与闭包
后厂村村长 • 阅读 150

Go 语言操作 MySQL 之 预处理
Meng小羽 • 赞 7 • 阅读 9k • 评论 1

什么是MVCC？看看它的实现原理
归思君 • 赞 5 • 阅读 503

MySQL 事务的 ACID 特性
归思君 • 赞 4 • 阅读 288

怎么优雅的选择 MySQL 存储引擎
Meng小羽 • 赞 2 • 阅读 2.8k

Go 语言操作 MySQL 之 SQLX 包
Meng小羽 • 赞 1 • 阅读 10.7k • 评论 4

Go 语言操作 MySQL 之 事务操作
Meng小羽 • 赞 1 • 阅读 9.6k

1 条评论　得票　最新

撰写评论 ...

提交评论

评论支持部分 Markdown 语法：`**粗体**` `_斜体_` `[链接](http://example.com)` `` `代码` `` `- 列表` `> 引用`。你还可以使用 `@` 来通知其他用户。

itemcf：不好意思打扰博主了，我发现您写的文章都很不错，可以转载您主页里的文章到openSnn吗？我会在转载的文章下标记出处和作者。请博主放心，在还没有博主您的同意前我不会进行转载。

• 回复 • 2023-09-20 • 来自广东