

为什么 MySQL 使用 B+ 树

2019-12-11

为什么这么设计

系统设计

MySQL

B+ 树

B 树

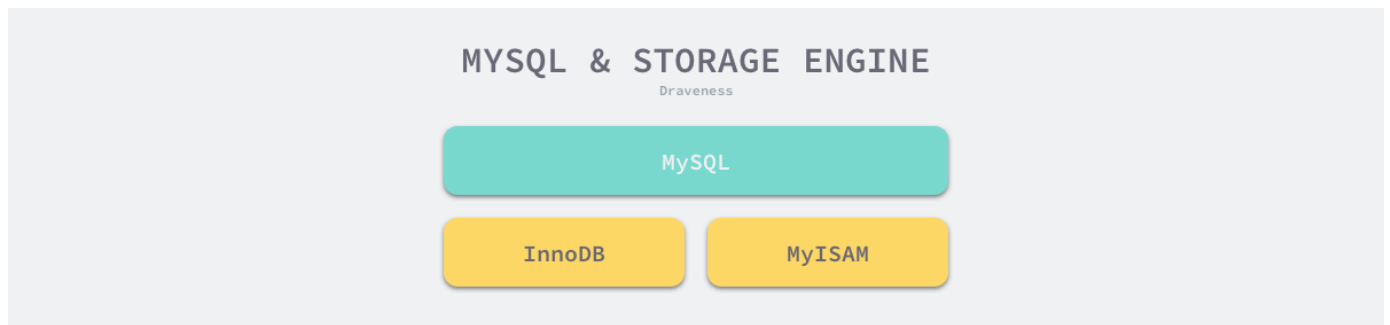
数据结构

为什么这么设计（**Why's THE Design**）是一系列关于计算机领域中程序设计决策的文章，我们在这个系列的每一篇文章中都会提出一个具体的问题并从不同的角度讨论这种设计的优缺点、对具体实现造成的影响。如果你有想要了解的问题，可以在文章下面留言。

为什么 **MySQL** 使用 **B+** 树是面试中经常会出现的问题，很多人对于这个问题可能都有一些自己的理解，但是多数的回答都不够完整和准确，大多数人都只会简单说一下 **B+** 树和 **B** 树的区别，但是都没有真正回答 **MySQL** 为什么选择使用 **B+** 树这个问题，我们在这篇文章中就会深入分析 **MySQL** 选择 **B+** 树背后的一些原因。

概述

首先需要澄清的一点是，**MySQL** 跟 **B+** 树没有直接的关系，真正与 **B+** 树有关系的是 **MySQL** 的默认存储引擎 **InnoDB**，**MySQL** 中存储引擎的主要作用是负责数据的存储和提取，除了 **InnoDB** 之外，**MySQL** 中也支持 **MyISAM** 作为表的底层存储引擎。



我们在使用 **SQL** 语句创建表时就可以为当前表指定使用的存储引擎，你能在 **MySQL** 的文档 [Alternative Storage Engines](#) 中找到它支持的全部存储引擎，例如：**MyISAM**、**CSV**、**MEMORY** 等，然而默认情况下，使用如下所示的 **SQL** 语句来创建表就会得到 **InnoDB** 存储引擎支撑的表：

```
CREATE TABLE t1 (  
  a INT,  
  b CHAR (20  
) , PRIMARY KEY (a)) ENGINE=InnoDB;
```

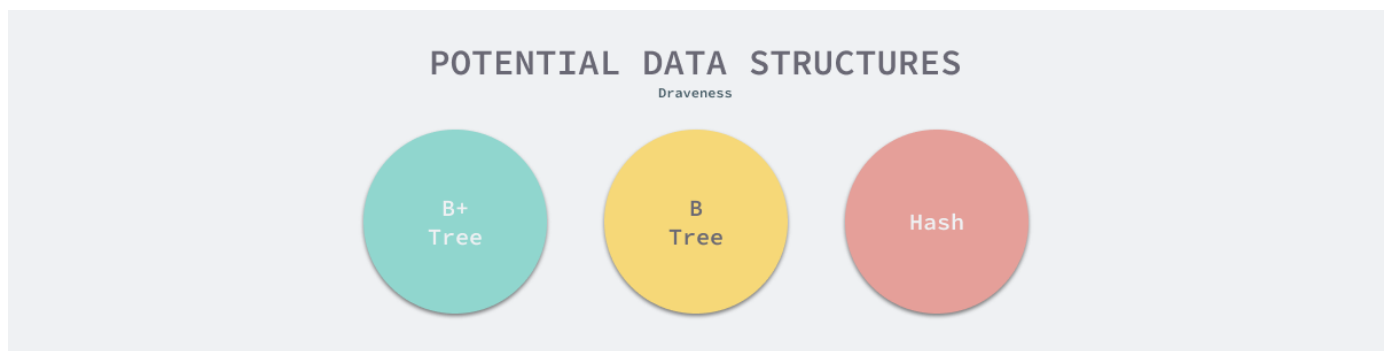


想要详细了解 **MySQL** 默认存储引擎的读者，可以通过之前的文章『[深入浅出 MySQL 和 InnoDB](#)』了解包括 **InnoDB** 存储方式、索引和锁等内容，我们在这里主要不会介绍 **InnoDB** 相关的过多内容。

我们今天最终将要分析的问题其实还是，为什么 **MySQL** 默认的存储引擎 **InnoDB** 会使用 **B+** 树来存储数据，相信对 **MySQL** 稍微有些了解的人都知道，无论是表中的数据（主键索引）还是辅助索引最终都会使用 **B+** 树来存储数据，其中前者在表中会以 $\langle id, row \rangle$ 的方式存储，而后者会以 $\langle index, id \rangle$ 的方式进行存储，这其实也比较好理解：

- 在主键索引中， id 是主键，我们能够通过 id 找到该行的全部列；
- 在辅助索引中，索引中的几个列构成了键，我们能够通过索引中的列找到 id ，如果有需要的话，可以再通过 id 找到当前数据行的全部内容；

对于 **InnoDB** 来说，所有的数据都是以键值对的方式存储的，主键索引和辅助索引在存储数据时会将 id 和 $index$ 作为键，将所有列和 id 作为键对应的值。



在具体分析 **InnoDB** 使用 **B+** 树背后的原因之前，我们需要为 **B+** 树找几个『假想敌』，因为如果我们只有一个选择，那么选择 **B+** 树也并不值得讨论，找到的两个假想敌就是 **B** 树和哈希，相信这也是很多人会在面试中真实遇到的问题，我们就以这两种数据结构为例，分析比较 **B+** 树的优点。

设计

到了这里我们已经明确了今天待讨论的问题，也就是为什么 **MySQL** 的 **InnoDB** 存储引擎会选择 **B+** 树作为底层的数据结构，而不选择 **B** 树或者哈希？在这一节中，我们将通过以下的两个方面介绍 **InnoDB** 这样选择的原因。

- **InnoDB** 需要支持的场景和功能需要在特定查询上拥有较强的性能；
- **CPU** 将磁盘上的数据加载到内存中需要花费大量的时间，这使得 **B+** 树成为了非常好的选择；



数据的持久化以及持久化数据的查询其实是一个常见的需求，而数据的持久化就需要我们与磁盘、内存和 CPU 打交道；MySQL 作为 OLTP 的数据库不仅需要具备事务的处理能力，而且要保证数据的持久化并且能够有一定的实时数据查询能力，这些需求共同决定了 B+ 树的选择，接下来我们会详细分析上述两个原因背后的逻辑。

读写性能

很多人对 OLTP 这个词可能不是特别了解，我们帮助各位读者快速理解一下，与 OLTP 相比的还有 OLAP，它们分别是 Online Transaction Processing 和 Online Analytical Processing，从这两个名字中我们就可以看出，前者指的就是传统的关系型数据库，主要用于处理基本的、日常的事务处理，而后者主要在数据仓库中使用，用于支持一些复杂的分析和决策。



作为支撑 OLTP 数据库的存储引擎，我们经常会使用 InnoDB 完成以下的一些工作：

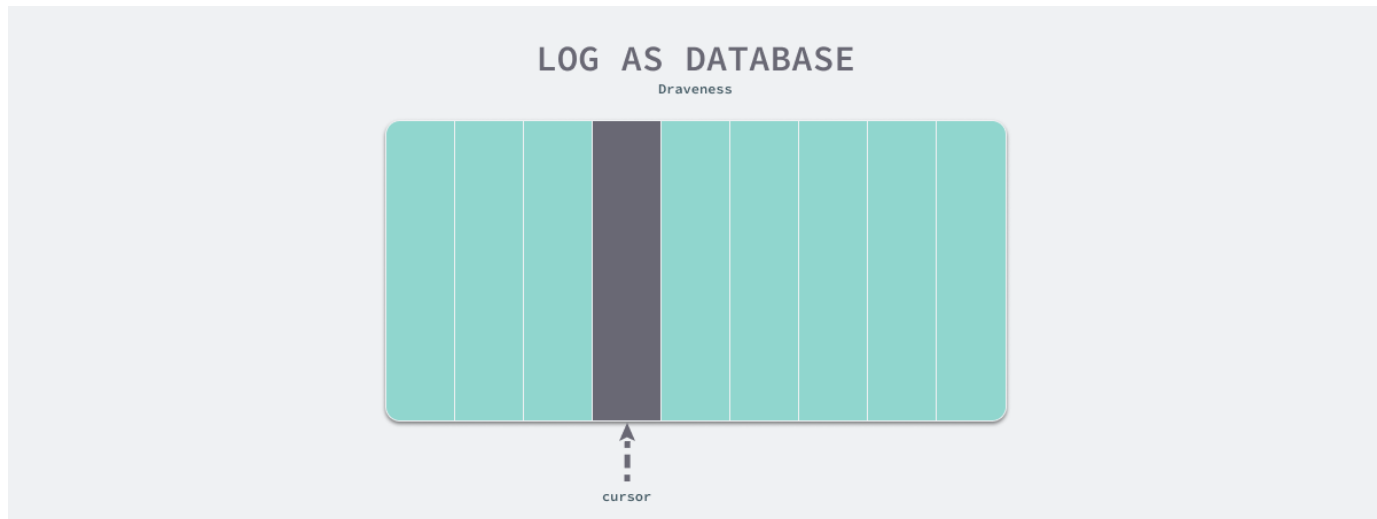
- 通过 INSERT、UPDATE 和 DELETE 语句对表中的数据进行增加、修改和删除；
- 通过 UPDATE 和 DELETE 语句对符合条件的数据进行批量的删除；
- 通过 SELECT 语句和主键查询某条记录的全部列；
- 通过 SELECT 语句在表中查询符合某些条件的记录并根据某些字段排序；
- 通过 SELECT 语句查询表中数据的行数；
- 通过唯一索引保证表中某个字段或者某几个字段的唯一性；

如果我们使用 B+ 树作为底层的数据结构，那么所有只会访问或者修改一条数据的 SQL 的时间复杂度都是 $O(\log n)$ ，也就是树的高度，但是使用哈希却有可能达到 $O(1)$ 的时间复杂度，看起来是不是特别的美好。但是当我们使用如下所示的 SQL 时，哈希的表现就不会这么好了：

```
SELECT * FROM posts WHERE author = 'draven' ORDER BY created_at DESC
SELECT * FROM posts WHERE comments_count > 10
UPDATE posts SET github = 'github.com/draveness' WHERE author = 'draven'
DELETE FROM posts WHERE author = 'draven'
```



如果我们使用哈希作为底层的数据结构，遇到上述的场景时，使用哈希构成的主键索引或者辅助索引可能就没有办法快速处理了，它对于处理范围查询或者排序性能会非常差，只能进行全表扫描并依次判断是否满足条件。全表扫描对于数据库来说是一个非常糟糕的结果，这其实也就意味着我们使用的数据结构对于这些查询没有其他任何效果，最终的性能可能都不如从日志中顺序进行匹配。



使用 **B+** 树其实能够保证数据按照键的顺序进行存储，也就是相邻的所有数据其实都是按照自然顺序排列的，使用哈希却无法达到这样的效果，因为哈希函数的目的就是让数据尽可能被分散到不同的桶中进行存储，所以在遇到可能存在相同键 `author = 'draveness'` 或者排序以及范围查询 `comments_count > 10` 时，由哈希作为底层数据结构的表可能会面对数据库查询的噩梦 —— 全表扫描。

B 树和 **B+** 树在数据结构上其实有一些类似，它们都可以按照某些顺序对索引中的内容进行遍历，对于排序和范围查询等操作，**B** 树和 **B+** 树相比于哈希会带来更好的性能，当然如果索引建立不够好或者 **SQL** 查询非常复杂，依然会导致全表扫描。

与 **B** 树和 **B+** 树相比，哈希作为底层的数据结构的表能够以 $O(1)$ 的速度处理单个数据行的增删改查，但是面对范围查询或者排序时就会导致全表扫描的结果，而 **B** 树和 **B+** 树虽然在单数据行的增删查改上需要 $O(\log n)$ 的时间，但是它会将索引列相近的数据按顺序存储，所以能够避免全表扫描。

数据加载

既然使用哈希无法应对我们常见的 **SQL** 中排序和范围查询等操作，而 **B** 树和 **B+** 树都可以相对高效地执行这些查询，那么为什么我们不选择 **B** 树呢？这个原因其实非常简单 —— 计算机在读写文件时会以页为单位将数据加载到内存中。页的大小可能会根据操作系统的不同而发生变化，不过在大多数的操作系统中，页的大小都是 4KB，你可以通过如下的命令获取操作系统页大小：



```
$ getconf PAGE_SIZE
4096
```

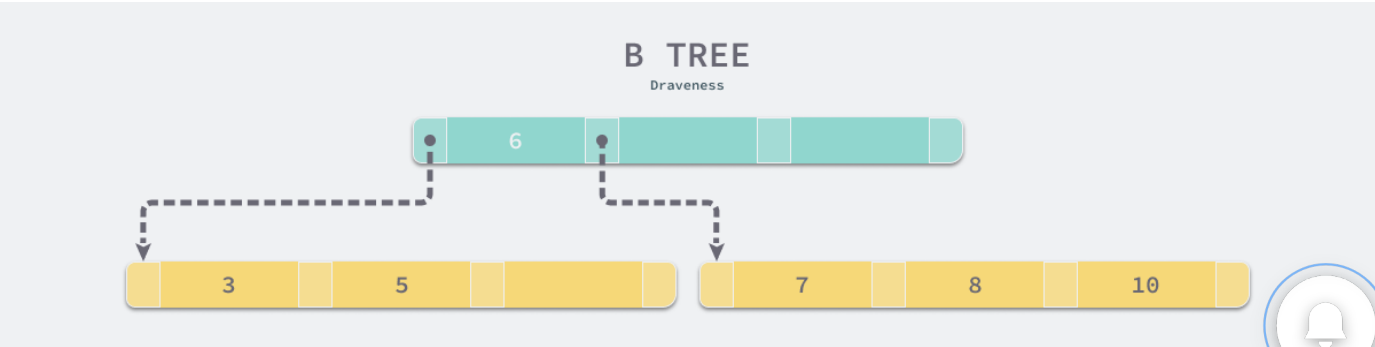
作者使用 macOS 系统的页大小就是 4KB，当然在不同的计算机上得到不同的结果是完全有可能的。

当我们需要在数据库中查询数据时，CPU 会发现当前数据位于磁盘而不是内存中，这时就会触发 I/O 操作将数据加载到内存中进行访问，数据的加载都是以页的维度进行加载的，然而将数据从磁盘读取到内存中所需的成本是非常大的，普通磁盘（非 SSD）加载数据需要经过队列、寻道、旋转以及传输的这些过程，大概要花费 10ms 左右的时间。



我们在估算 MySQL 的查询时就可以使用 10ms 这个数量级对随机 I/O 占用的时间进行估算，这里想要说的是随机 I/O 对于 MySQL 的查询性能影响会非常大，而顺序读取磁盘中的数据时速度可以达到 40MB/s，这两者的性能差距有几个数量级，由此我们也应该尽量减少随机 I/O 的次数，这样才能提高性能。

B 树与 B+ 树的最大区别就是，B 树可以在非叶结点中存储数据，但是 B+ 树的所有数据其实都存储在叶子节点中，当一个表底层的数据结构是 B 树时，假设我们需要访问所有『大于 4，并且小于 9 的数据』：



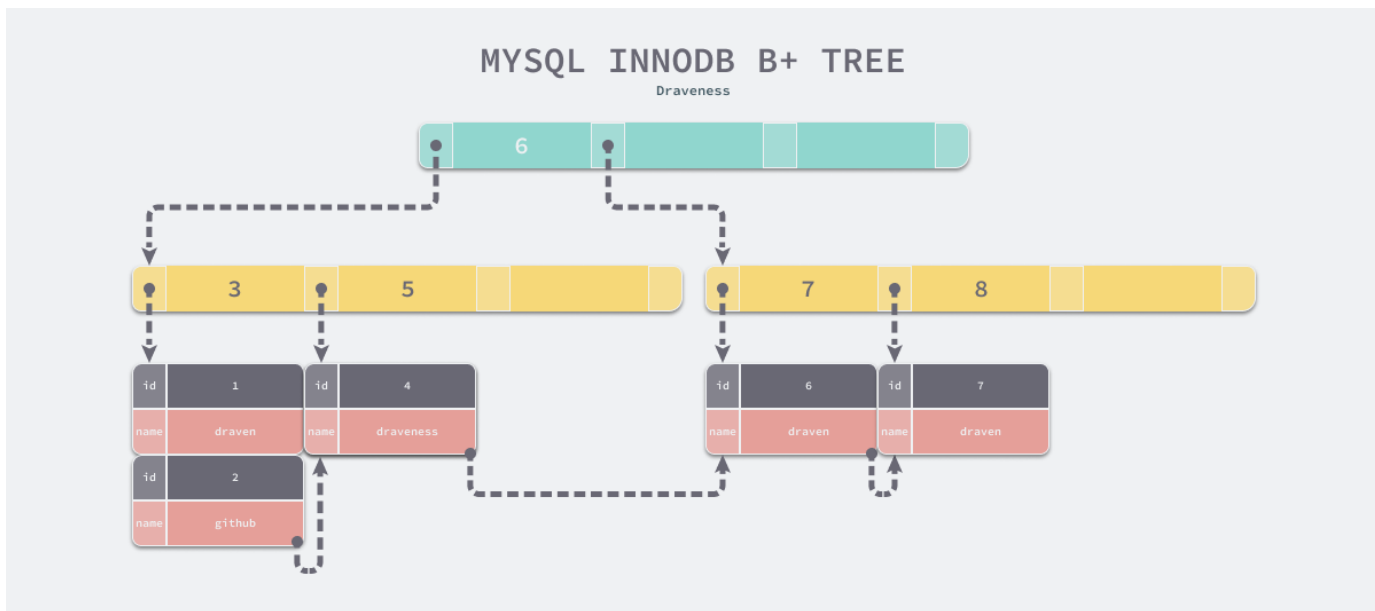
如果不考虑任何优化，在上面的简单 B 树中我们需要进行 4 次磁盘的随机 I/O 才能找到所有满足条件的数据行：

1. 加载根节点所在的页，发现根节点的第一个元素是 6，大于 4；
2. 通过根节点的指针加载左子节点所在的页，遍历页面中的数据，找到 5；
3. 重新加载根节点所在的页，发现根节点不包含第二个元素；
4. 通过根节点的指针加载右子节点所在的页，遍历页面中的数据，找到 7 和 8；

当然我们可以通过各种方式来对上述的过程进行优化，不过 B 树能做的优化 B+ 树基本都可以，所以我们不需要考虑优化 B 树而带来的收益，直接来看看什么样的优化 B+ 树可以做，而 B 树不行。

由于所有的节点都可能包含目标数据，我们总是要从根节点向下遍历子树查找满足条件的数据行，这个特点带来了大量的随机 I/O，也是 B 树最大的性能问题。

B+ 树中就不存在这个问题了，因为所有的数据行都存储在叶节点中，而这些叶节点可以通过『指针』依次按顺序连接，当我们在如下所示的 B+ 树遍历数据时可以直接在多个子节点之间进行跳转，这样能够节省大量的磁盘 I/O 时间，也不需要不同层级的节点之间对数据进行拼接和排序；通过一个 B+ 树最左侧的叶子节点，我们可以像链表一样遍历整个树中的全部数据，我们也可以引入双向链表保证倒序遍历时的性能。



有些读者可能会认为使用 B+ 树这种数据结构会增加树的高度从而增加整体的耗时，然而高度为 3 的 B+ 树就能够存储千万级别的数据，实践中 B+ 树的高度最多也就 4 或者 5，所以这并不是影响性能的根本问题。

总结



任何不考虑应用场景的设计都不是最好的设计，当我们明确的定义了使用 **MySQL** 时的常见查询需求并理解场景之后，再对不同的数据结构进行选择就成了理所当然的事情，当然 **B+** 树可能无法对所有 **OLTP** 场景下的查询都有着较好的性能，但是它能够解决大多数的问题。

我们在这里重新回顾一下 **MySQL** 默认的存储引擎选择 **B+** 树而不是哈希或者 **B** 树的原因：

- 哈希虽然能够提供 $O(1)$ 的单数据行操作性能，但是对于范围查询和排序却无法很好地支持，最终导致全表扫描；
- **B** 树能够在非叶节点中存储数据，但是这也导致在查询连续数据时可能会带来更多的随机 I/O，而 **B+** 树的所有叶节点可以通过指针相互连接，能够减少顺序遍历时产生的额外随机 I/O；

如果想要追求各方面的极致性能也不是没有可能，只是会带来更高的复杂度，我们可以为一张表同时建 **B+** 树和哈希构成的存储结构，这样不同类型的查询就可以选择相对更快的数据结构，但是会导致更新和删除时需要操作多份数据。

从今天的角度来看，**B+** 树可能不是 **InnoDB** 的最优选择，但是它一定是能够满足当时设计场景的需要，从 **B+** 树作为数据库底层的存储结构到今天已经过了几十年的时间，我们不得不说优秀的工程设计确实有足够的生命力。而我们作为工程师，在选择数据库时也应该非常清楚地知道不同数据库适合的场景，因为软件工程中并没有银弹。

到最后，我们还是来看一些比较开放的相关问题，有兴趣的读者可以仔细思考一下下面的问题：

- 常用于分析的 **OLAP** 数据库一般会使用什么样的数据结构存储数据？为什么？
- **Redis** 是如何对数据进行持久化存储的？常见的数据结构都有什么？

如果对文章中的内容有任何疑问或者想要了解更多软件工程上一些设计决策背后的原因，可以在博客下面留言，作者会及时回复本文相关的疑问并选择其中合适的主题作为后续的内容。

Reference

- [B+ tree · Wikipedia](#)
- [What is the difference between Mysql InnoDB B+ tree index and hash index? Why does MongoDB use B-tree?](#)
- [B+Trees and why I love them, part I](#)
- [What are the main differences between INNODB and MYISAM](#)
- [B+ Tree File Organization](#)
- [Database Index: A Re-visit to B+ Tree](#)



- Fundamentals of database systems

{% include related/whys-the-design.md %}



微信搜一搜

🔍 真没什么逻辑

转载申请



本作品采用 知识共享署名 4.0 国际许可协议 进行许可，转载时请注明原文链接，图片在使用时请保留全部内容，可适当缩放并在引用处附上图片所在的文章链接。

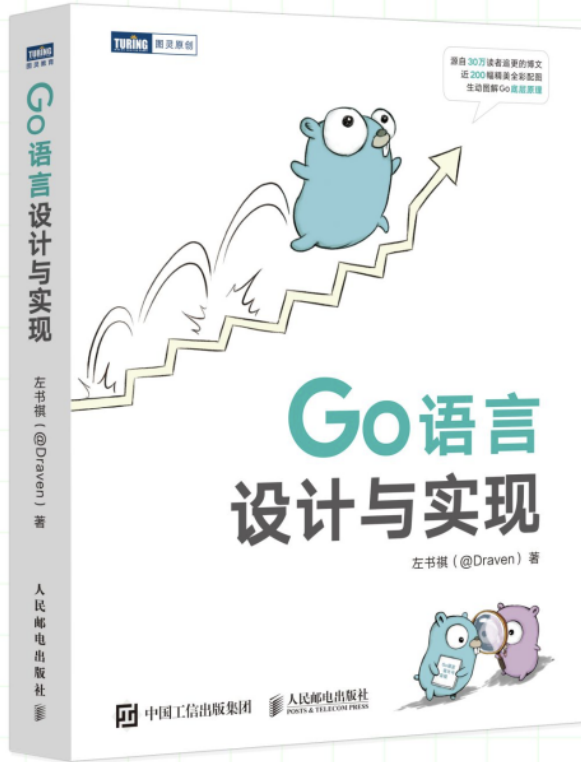
Go 语言设计与实现

各位读者朋友，很高兴大家通过本博客学习 **Go** 语言，感谢一路相伴！《Go语言设计与实现》 的纸质版图书已经上架京东，本书目前已经四印，印数超过 **10,000** 册，有需要的朋友请点击 链接 或者下面的图片购买。



30万读者共同学习的 Go 底层原理书

全彩印刷
基于 Go 1.15



近 200
幅全彩精美配图

600
多精选源码片段

酣畅淋漓的
文字剖析

以前所未有的方式轻松读懂 Go 源码

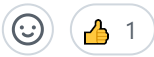
※ 数据来源于图书及《Go 语言设计与实现》博客流量

文章图片

你可以在 [技术文章配图指南](#) 中找到画图的方法和素材。



1 个反应



55 条评论 · 1+ 条回复 - 由 giscus 驱动


最旧 最新

输入 预览 Aa

登录以评论

支持 Markdown

GitHub 登录

 hhyvs111 2019 年 12 月 12 日


一个面试题，想问一下博主，一条普通的sql语句，如


```
t1
id
a b c d

select * from t1 where c = 5;
```


id是主键，然后有c的索引，这一条语句会访问几次磁盘？
我的理解是b+树的高度就是访问磁盘的次数，也就是说先用3次（假如树的高度为3）找到c索引的键值，然后再从主键索引里找出对应的数据，一共就是6次。这样理解对吗？

↑ 1



 1

0 条回复


 draveness 2019 年 12 月 12 日 所有者

一个面试题，想问一下博主，一条普通的sql语句，如

```
t1
id
a b c d

select * from t1 where c = 5;
```

id是主键，然后有c的索引，这一条语句会访问几次磁盘？
我的理解是b+树的高度就是访问磁盘的次数，也就是说先用3次（假如树的高度为3）找到c索引的键值，然后再从主键索引里找出对应的数据，一共就是6次。这样理解对吗？



你可以这么理解，但是 MySQL 中一般情况下所有的非叶节点都会在内存中，只有访问叶节点才会需要随机 I/O 加载数据，如果不考虑这个就是 6 次，但是一般情况下最多只会有 2 次

↑ 1



0 条回复



Leviathan1995 2019 年 12 月 13 日

已编辑

@hhyvs111

一个面试题，想问一下博主，一条普通的sql语句，如

```
t1
id
a b c d

select * from t1 where c = 5;
```

id是主键，然后有c的索引，这一条语句会访问几次磁盘？

我的理解是b+树的高度就是访问磁盘的次数，也就是说先用3次（假如树的高度为3）找到c索引的键值，然后再从主键索引里找出对应的数据，一共就是6次。这样理解对吗？

MySQL中Page存在于Buffer Pool和是否为叶子节点没有关系，按照你的描述，这里只能假设Page均不在Buffer Pool中，所以需要6次IO: 二级索引3次，主键索引3次。在InnoDB中的实现中Buffer Pool会进行read ahead预读，所以真正具体IO次数无法确定。

↑ 1



0 条回复



draveness 2019 年 12 月 13 日 所有者

已编辑

MySQL中Page存在于Buffer Pool和是否为叶子节点没有关系，按照你的描述，这里只能假设Page均不在Buffer Pool中，所以需要6次IO: 二级索引3次，主键索引3次。

对于大表所有的索引页都会缓存在内存里，但是内存里一般是放不下整个表中全部数据的。

在InnoDB中的实现中Buffer Pool会进行read ahead预读，所以真正具体IO次数无法确定。

你说得对，不过文中说过了不考虑优化策略。

↑ 1



0 条回复



Leviathan1995 2019 年 12 月 13 日

大表所有的索引页都会缓存在内存

这个我看代码倒没有发现这方面的优化，本质上Buffer Pool是LRU策略，针对大表缓存 non leaf page 和

IO会减少，并且Buffer Pool对表的大小也是无感知的



LRU 云/件大，并且 Buffer Pool 对表的大小也是无感知的。

↑ 1



0 条回复



draveness 2019 年 12 月 13 日 所有者

这个我看代码倒没有发现这方面的优化，本质上 Buffer Pool 是 LRU 策略，针对大表缓存 non leaf page 和 LRU 会冲突，并且 Buffer Pool 对表的大小也是无感知的。

与整个表的大小相比，索引页的大小完全可以存储在内存中，而访问索引页的频率会大概率比数据页高得多，所以只要内存足够，最终索引页一定会缓存在内存中。

↑ 1



0 条回复



Leviathan1995 2019 年 12 月 13 日

@draveness

这个我看代码倒没有发现这方面的优化，本质上 Buffer Pool 是 LRU 策略，针对大表缓存 non leaf page 和 LRU 会冲突，并且 Buffer Pool 对表的大小也是无感知的。

与整个表的大小相比，索引页的大小完全可以存储在内存中，而访问索引页的频率会大概率比数据页高得多，所以只要内存足够，最终索引页一定会缓存在内存中。

你说的是可能的结果，MySQL 没有这样的机制。

↑ 1



0 条回复



draveness 2019 年 12 月 13 日 所有者

@Leviathan1995

@draveness

这个我看代码倒没有发现这方面的优化，本质上 Buffer Pool 是 LRU 策略，针对大表缓存 non leaf page 和 LRU 会冲突，并且 Buffer Pool 对表的大小也是无感知的。

与整个表的大小相比，索引页的大小完全可以存储在内存中，而访问索引页的频率会大概率比数据页高得多，所以只要内存足够，最终索引页一定会缓存在内存中。

你说的是可能的结果，MySQL 没有这样的机制。

行 👍

↑ 1



1

0 条回复



 arrayadd 2019 年 12 月 17 日

还是挺感慨的，软件工程师们想尽各种办法来做优化，性能上才能提升那么一点点，而硬件上的一旦有进步，比起来简直就是小巫见大巫。顺序读取1MB数据，机械硬盘是20ms，SSD只需要1ms。20倍差距，B,B+优劣在这里就....

↑ 1



0 条回复



MikeoPerfect 2019 年 12 月 17 日

基本上讲到点子上了，MySQL的InnoDB存储引擎，之所以采用B+树的设计思路，和InnoDB的数据存储方式离不开关系，操作系统的页默认大小为4KB，innodb的页大小为16KB，当表中的记录存入InnoDB时，为了查询方便会采用操作系统文件管理中的文件索引思路存放记录数据，也即文中介绍的指针型节点，一个记录从访问到被访问基本会经过如下过程：目录页、页目录、页、记录，其中前三者体现的就是索引的价值，最后一者就是我们说的数据了，而这就是B+树为什么非叶子节点只有索引，叶子节点才有数据的根源

↑ 1



0 条回复



MikeoPerfect 2019 年 12 月 18 日

已编辑

今天碰巧看到一篇博文，介绍的就是MySQL innodb_page_size，可能对大家有所帮助，innodb_page_size是一个初始化数据库实例的参数，在目前的版本中（>=5.7.6），可以选择的值有4096, 8192, 16384, 32768, 65536。默认是16KB

一般越小，内存划分粒度越大，使用率越高，但是会有其他问题，就是限制了索引字段还有整行的大小。innodb引擎读取内存还有更新都是一页一页更新的，这个innodb_page_size决定了，一个基本页的大小。常用B+Tree索引，B+树是为磁盘及其他存储辅助设备而设计一种平衡查找树（不是二叉树）。B+树中，所有记录的节点按大小顺序存放在同一层的叶子节点中，各叶子节点用指针进行连接。MySQL将每个叶子节点的大小设置为一个页的整数倍，利用磁盘的预读机制，能有效减少磁盘I/O次数，提高查询效率。如果一个行数据，超过了一页的一半，那么一个页只能容纳一条记录，这样B+Tree在不理想的情况下就变成了双向链表。

原文出处：[MySQL原理 - InnoDB表的限制](#)

↑ 1



0 条回复



draveness 2019 年 12 月 18 日 所有者

已编辑

今天碰巧看到一篇博文，介绍的就是MySQL innodb_page_size，可能对大家有所帮助，

innodb_page_size是一个初始化数据库实例的参数，在目前的版本中（>=5.7.6），可以选择的值有4096, 8192, 16384, 32768, 65536。默认是16KB

一般越小，内存划分粒度越大，使用率越高，但是会有其他问题，就是限制了索引字段还有整行的大小。innodb引擎读取内存还有更新都是一页一页更新的，这个innodb_page_size决定了，一个基本页的大小。常用B+Tree索引，B+树是为磁盘及其他存储辅助设备而设计一种平衡查找树（不是二叉树）。B+树中，所有记录的节点按大小顺序存放在同一层的叶子节点中，各叶子节点用指针进行连接。MySQL将每个叶子节点的大小设置为一个页的整数倍，利用磁盘的

预读机制，能有效减少磁盘I/O次数，提高查询效率。如果一个行数据，超过了一页的一半，那么一个页只能容纳一条记录，这样B+ Tree在不理想的情况下就变成了双向链表。

原文出处: [MySQL原理 - InnoDB表的限制](#)

之前写过 MySQL 和 InnoDB 的文章，可以看下 [『深入浅出』MySQL 和 InnoDB](#)

↑ 1



0 条回复



XhinLiang 2019 年 12 月 21 日

博主，你的博客貌似挂了 <https://draveness.me>

↑ 1



0 条回复



draveness 2019 年 12 月 21 日 所有者

已编辑

博主，你的博客貌似挂了 <https://draveness.me>

你是北京联通么，北京联通把我的博客劫持了，我跟他们投诉了，你可以先访问 <https://draven.co>

↑ 1



0 条回复



boykait 2019 年 12 月 23 日

所以在遇到可能存在相同键 `author = 'draven'` 这个hash应该都会散列到同一个hash桶里面吧

↑ 1



0 条回复

25 个隐藏项
[加载更多...](#)



techone577 2020 年 11 月 11 日

您好，《为什么 MongoDB 使用 B 树》这篇文章是已经删除了吗

↑ 1



0 条回复



draveness 2020 年 11 月 11 日 所有者

您好，《为什么 MongoDB 使用 B 树》这篇文章是已经删除了吗

这篇文章有问题，没时间改暂时关掉了

↑ 1



0 条回复



techone577 2020 年 11 月 11 日

@draveness

您好，《为什么 MongoDB 使用 B 树》这篇文章是已经删除了吗

这篇文章有问题，没时间改暂时关掉了

好的感谢解答，我看现在网上包括官方文档对 MongoDB 使用 B 树还是 B+ 树的描述都有些不清晰，所以期待大佬修改后的文章

↑ 1



0 条回复



xiaoheiAh 2021 年 2 月 2 日

{% include related/whys-the-design.md %} 关联的子页面失效了

↑ 1



0 条回复



ceezyyy 2021 年 3 月 25 日

大佬，B+ tree 为啥叶子结点没有 6？

↑ 1



0 条回复



draveness 2021 年 3 月 27 日 所有者

@ceezyyy 大佬，B+ tree 为啥叶子结点没有 6？

因为没人写所以没有？

↑ 1



0 条回复



rookiezq 2021 年 4 月 1 日

想问一下，既然层与层之间会有 I/O，那在遍历叶子节点时会有 I/O 吗？叶子节点时连续存储的吗？谢谢！

↑ 1



0 条回复

**nobject** 2021 年 6 月 16 日

一直没搞明白树的层数与磁盘io的关系，比如红黑树层数高，磁盘的io就高，是因为同一层的节点容易存在同一个数据页中么？

↑ 1



0 条回复

**MakeChan** 2021 年 6 月 19 日**@nobject**

一直没搞明白树的层数与磁盘io的关系，比如红黑树层数高，磁盘的io就高，是因为同一层的节点容易存在同一个数据页中么？

我和你有同样的问题，目前也没搞明白，“为什么一个硬盘块只能存储一个树节点”。希望有博主或其他懂的大佬能解答我的疑惑。

↑ 1



0 条回复

**draveness** 2021 年 6 月 23 日 所有者**@nobject**

一直没搞明白树的层数与磁盘io的关系，比如红黑树层数高，磁盘的io就高，是因为同一层的节点容易存在同一个数据页中么？

我和你有同样的问题，目前也没搞明白，“为什么一个硬盘块只能存储一个树节点”。希望有博主或其他懂的大佬能解答我的疑惑。

你希望访问一个节点的时候触发多次 IO 么

↑ 1



0 条回复

**rookiezq** 2021 年 6 月 23 日**@nobject**

一直没搞明白树的层数与磁盘io的关系，比如红黑树层数高，磁盘的io就高，是因为同一层的节点容易存在同一个数据页中么？

不是的，磁盘页大小是4k，mysql的页是16k，所以mysql一页≈4页磁盘页；mysql中按照B+树存储，每一个树节点占了一个mysql页，所以当节点指向另一个节点时就会发生io

↑ 1



0 条回复



MakeChan 2021 年 6 月 23 日

@nobject

一直没搞明白树的层数与磁盘io的关系，比如红黑树层数高，磁盘的io就高，是因为同一层的节点容易存在同一个数据页中么？

我和你有同样的问题，目前也没搞明白，“为什么一个硬盘块只能存储一个树节点”。希望有博主或其他懂的大佬能解答我的疑惑。

你希望访问一个节点的时候触发多次 IO 么

不是很明白您的意思。我的意思是，为什么不能用二叉树，将多个节点储存到一个块中，每个节点的指针可以设计为 块号+偏移。每次访问一个节点，和它同属一个磁盘块的节点也能一同读出。为什么一个磁盘块只能存储至多一个节点，这个我弄不懂。

↑ 1



1 条回复



Yuu177 2022 年 10 月 19 日



我的问题和你一样，但是回复到这里就断了，这个疑惑你解决了吗？



zhouwentong1993 2021 年 7 月 26 日

@draveness

博主你好 有点疑问：

问题1：

我看到网上别篇文章对b树的说法是，b树非叶子节点也会存数据，所以会导致树高度增加，从而导

致的额外随机i/o。

但博主的文章是，B+ 树这种数据结构会增加树的高度

这个能给出来链接么，质量高一些的引用

问题2：

另外，文章说到，B+ 树的所有叶节点可以通过指针相互连接，能够减少顺序遍历时产生的额外随机 I/O

这么说，B+和B树的效率的最大影响是B+树可以通过『指针』依次按顺序连接，这样理解有问题么

是的，连接比较方便一些

博主好，《数据密集型应用系统设计》中的数据库核心：数据结构一章中，在「优化 B-tree」章节中提到了 B+ 树的层数少于 B 树。见 P83。

↑ 0



0 条回复





zhangyuanxue 2022 年 7 月 21 日

前文降到讲到计算机在读写文件时会以页为单位将数据加载到内存中，后文提到一种B树的 4 次磁盘的随机 I/O

加载根节点所在的页，发现根节点的第一个元素是 6，大于 4；
通过根节点的指针加载左子节点所在的页，遍历页面中的数据，找到 5；
重新加载根节点所在的页，发现根节点不包含第二个元素；
通过根节点的指针加载右子节点所在的页，遍历页面中的数据，找到 7 和 8；

从上面这段话可以看出B数的每一层是在不同的页里的，这一点大佬并没有提及，可否补充点细节呢？不太明白为什么不同层在不同的页



0 条回复



zhangyuanxue 2022 年 10 月 19 日

您发的邮件我已收到，我会马上查看的，谢谢。



0 条回复