第07章_InnoDB数据存储结构

1.数据库的存储结构:页

1.1磁盘与内存交互基本单位:页

1.2页结构概述

1.3页的大小

1.4页的上层结构

2.页的内部结构

第1部分:File Header(文件头部）和File Trailer (文件尾部)

1.数据库的存储结构:页

索引结构给我们提供了高效的索引方式，不过索引信息以及数据记录都是保存在文件上的，确切说是存储在页结构中。另一方面，索引是在存储引擎中实现的，MySQL服务器上的**存储引擎**负责对表中数据的读取和写入工作。不同存储引擎中**存放的格式**一般是不同的，甚至有的存储引擎比如Memory都不用磁盘来存储数据。

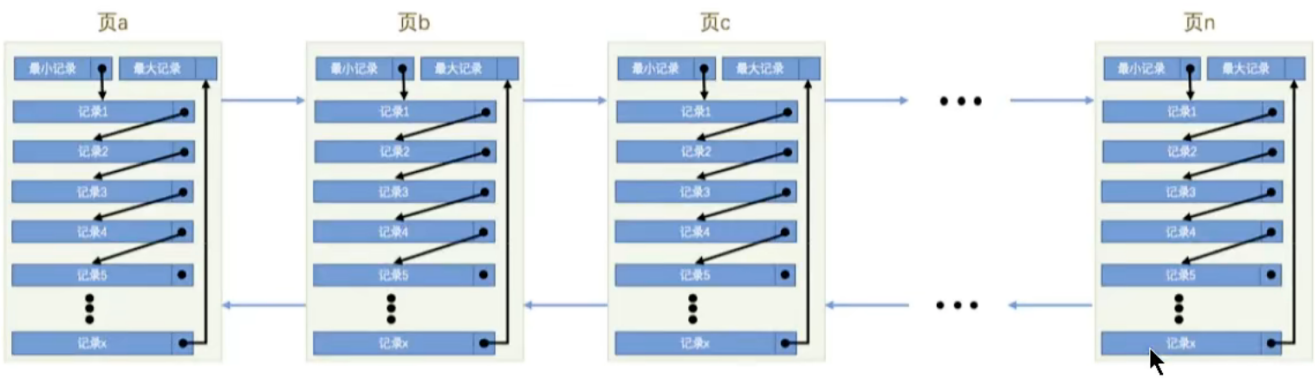
由于InnoDB是MySQL的默认存储引擎，所以本章剖析InnoDB存储引擎的数据存储结构。

1.1磁盘与内存交互基本单位:页

InnoDB将数据划分为若干个页，InnoDB中页的大小默认为 **16KB**

以`页`作为磁盘和内存之间交互的`基本单位`，也就是一次最少从磁盘中读取16KB的内容到内存中，一次最少把内存中的16KB内容刷新到磁盘中。也就是说，**在数据库中，不论读一行，还是读多行，都是将这些行所在的页进行加载。也就是说，数据库管理存储空间的基本单位是页(Page)，数据库I/O操作的最小单位是页。**一个页中可以存储多个行记录

> 记录是按照行来存储的，但是数据库的读取并不以行为单位，否则一次读取（也就是一次I/O操作)只能处理一行数据，效率会非常低。



1.2页结构概述

页a、页b、页c ...页n这些页可以不`在物理结构上相连`，只要通过`双向链表`相关联即可。每个数据页中的记录会按照主键值从小到大的顺序组成一个`单向链表`，每个数据页都会为存储在它里边的记录生成一个`页目录`，在通过主键查找某条记录的时候可以在页目录`中使用二分法`快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。

1.3页的大小

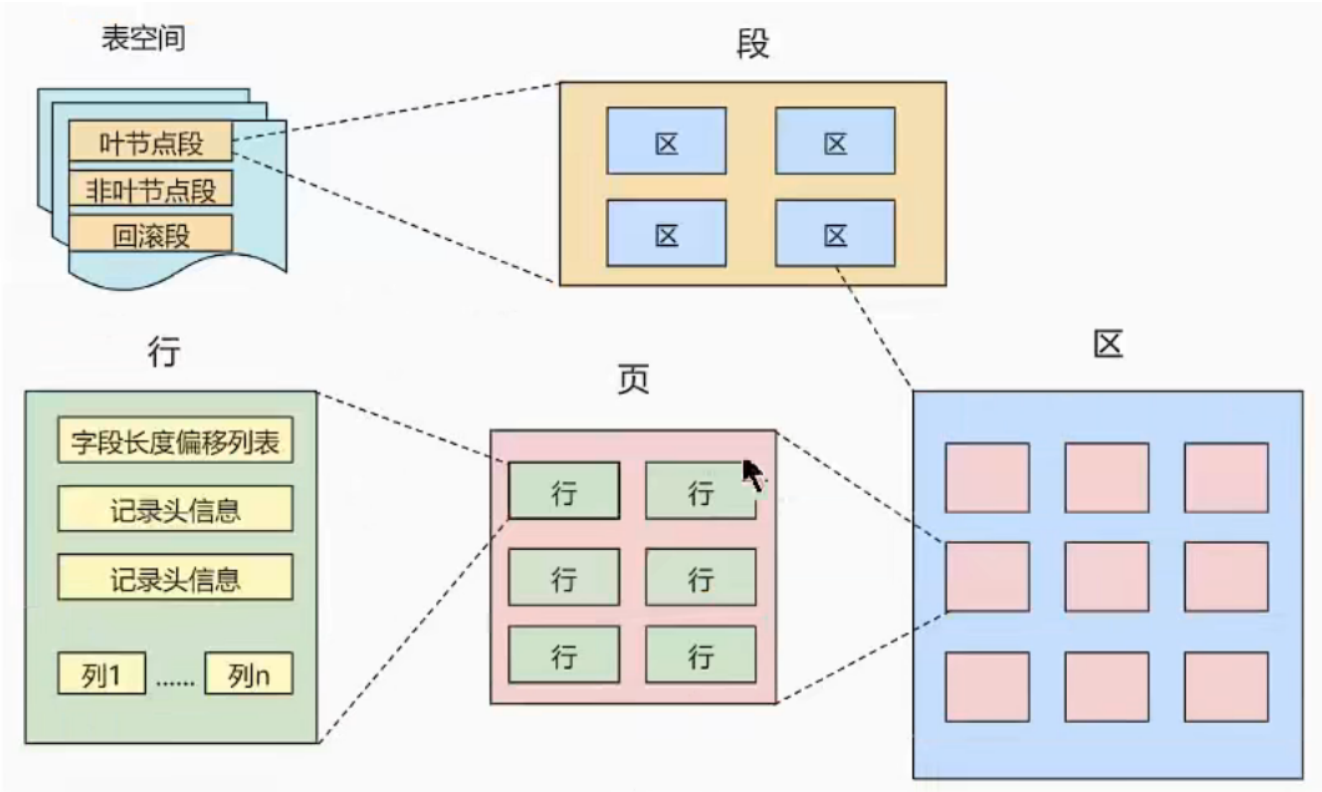
不同的数据库管理系统（简称DBMS）的页大小不同。比如在MySQL的InnoDB存储引擎中，默认页的大小是**16KB**，可以通过下面的命令来进行查看:

```
show variables like '%innodb_page_size%';
/*
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| innodb_page_size | 16384 |
+------------------+-------+
*/
```

SQL Server中页的大小为 `8KB`，而在oracle中用术语"`块`"(Block)来代表"页"，Oralce支持的块大小为2KB，4KB，8KB，16K8，32KB和64KB。

1.4页的上层结构

另外在数据库中，还存在区（Extent)、段(Segment)和表空间（Tablespace)的概念。行、页、区、段、表空间的关系如下图所示:



区(Extent)是比页大一级的存储结构，在InnoDB存储引擎中，一个区会分配 64个连续的页 。因为InnoDB中的页大小默认是16KB，所以一个区的大小是64*16KB= 1MB。

段(Segment)由一个或多个区组成，区在文件系统是一个连续分配的空间（在InnoDB中是连续的64个页），不过在段中不要求区与区之间是相邻的。 段是数据库中的分配单位，不同类型的数据库对象以不同的段形式存在。 当创建数据表、索引的时候，就会相应创建对应的段，比如创建一张表时会创建一个表段，创建一个索引时会创建一个索引段。

表空间（Tablespace)是一个逻辑容器，表空间存储的对象是段，在一个表空间中可以有一个或多个段，但是一个段只能属于一个表空间。数据库由一个或多个表空间组成，表空间从管理上可以划分为系统表空间， 用户表空间 、 撤销表空间 、 临时表空间 等。

2.页的内部结构

页如果按类型划分的话，常见的有 数据页（保存B+树节点） 、 系统页 、 Undo页 和 事务数据页 等。数据页是我们最常使用的页。 数据页的 16KB 大小的存储空间被划分为七个部分，分别是文件头(File Header)、页头(Page Header)、最大最小记录(Infimum+supremum)、用户记录(User Records)、空闲空间(Free Space)、页目录(Page Directory)和文件尾(File Tailer) 。

页结构的示意图如下所示:

这7个部分作用分别如下，简单梳理如下表所示:

名称	占用大小	说明
File Header	38字节	文件头，描述页的信息
Page Header	56字节	页头,页的状态信息
Infimum-Supremum	26字节	最大和最小记录，这是两个虚拟的行记录
User Records	不确定	用户记录，存储行记录内容
Free Space	不确定	空闲记录，页中还没有被使用的空间
Page Directory	不确定	页目录，存储用户记录的相对位置
File Trailer	8字节	文件尾,校验页是否完整

我们可以把这7个结构分成3个部分

第1部分: File Header(文件头部）和File Trailer (文件尾部)

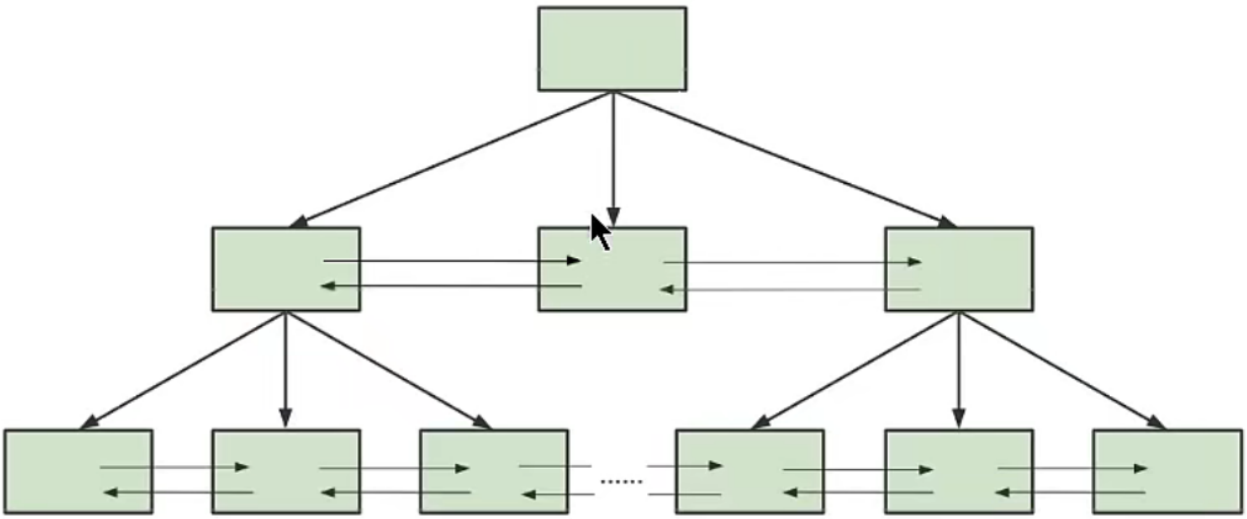
首先是`文件通用部分`，也就是`文件头`和`文件尾`。

① 文件头部信息

名称	占用空间大小	描述
`FIL_PAGE_SPACE_OR_CHKSUM`	`4`字节	页的校验和（checksum值）
`FIL_PAGE_OFFSET`	`4`字节	页号
`FIL_PAGE_PREV`	`4`字节	上一个页的页号
`FIL_PAGE_NEXT`	`4`字节	下一个页的页号
FIL_PAGE_LSN	`8`字节	页面被最后修改时对应的日志序列位置（英文名是: Log Sequence Number)
`FIL_PAGE_TYPE`	`2`字节	该页的类型
FIL_PAGE_FILE_FLUSH_LSN	`8`字节	仅在系统表空间的一个页中定义，代表文件至少被刷新到了对应的LSN值
`FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID`	`4`字节	页属于哪个表空间

2.3 从数据页角度看B + 树如何查询

一棵B+树按照节点类型可以分成两部分: 1．叶子节点，B+树最底层的节点，节点的高度为o，存储行记录。 2．非叶子节点，节点的高度大于0，存储索引键和页面指针，并不存储行记录本身。



当我们从页结构来理解B+树的结构的时候，可以帮我们理解一些通过索引进行检索的原理:

1.B+树是如何进行记录检索的?

如果通过B+树的索引查询行记录，首先是从B+树的根开始，逐层检索，直到找到叶子节点，也就是找到对应的数据页为止，将数据页加载到内存中，页目录中的槽(slot)采用`二分查找`的方式先找到一个粗略的记录分组然后再在分组中通过`链表遍历`的方式查找记录。

2.普通索引和唯一索引在查询效率上有什么不同?

我们创建索引的时候可以是普通索引，也可以是唯一索引，那么这两个索引在查询效率上有什么不同呢?

唯一索引就是在普通索引上增加了约束性，也就是关键字唯一，找到了关键字就停止检索。而普通索引，可能会存在用户记录中的关键字相同的情况，根据页结构的原理，当我们读取一条记录的时候，不是单独将这条记录从磁盘中读出去，而是将这个记录所在的页加载到内存中进行读取。InnoDB存储引擎的页大小为16KB，在一个页中可能存储着上千个记录，因此在普通索引的字段上进行查找也就是在内存中多几次“`判断下一条记录`”的操作，对于CPU来说，这些操作所消耗的时间是可以忽略不计的。所以对一个索引字段进行检索，采用普通索引还是唯一索引在检索效率上基本上没有差别。

3.InnoDB行格式(或记录格式)

我们平时的数据以行为单位来向表中插入数据，这些记录在磁盘上的存放方式也被称为行格式或者记录格式。

InnoDB存储引擎设计了4种不同类型的行格式，分别是Compact（紧密）、Redundant（冗余）、Dynamic（动态）和Compressed（压缩）行格式。查看

MySQL8 与 MySQL5.7的默认行格式:

```
mysql> select @@innodb_default_row_format;
+-------------------------------+
| @@innodb_default_row_format |
+-------------------------------+
| dynamic                       |
+-------------------------------+
1 row in set (0.00 sec)

# 查询单张表行格式
```

```
mysql> show table status like 'departments' \G
*************************** 1. row ***************************
           Name: departments
         Engine: InnoDB
        Version: 10
 #行格式  Row_format: Dynamic
           Rows: 27
 Avg_row_length: 606
    Data_length: 16384
Max_data_length: 0
   Index_length: 49152
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2022-03-23 14:56:38
    Update_time: 2022-03-23 14:56:38
     Check_time: NULL
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.01 sec)
```

4.区、段与碎片区

4.1为什么要有区?

B+树的每一层中的页都会形成一个双向链表，如果是以页为单位来分配存储空间的话，双向链表相邻的两个页之间的物理位置可能离得非常远。我们介绍B+树索引的适用场景的时候特别提到范围查询只需要定位到最左边的记录和最右边的记录,然后沿着双向链表一直扫描就可以了，而如果链表中相邻的两个页物理位置离得非常远，就是所谓的随机I/O。再一次强调，磁盘的速度和内存的速度差了好几个数量级，随机I/O是非常慢的，所以我们应该尽量让链表中相邻的页的物理位置也相邻，这样进行范围查询的时候才可以使用所谓的顺序I/O。

> 这样利用了磁盘的预读特性

查看4.n 扩展 理解mysql如何利用预读特性

引入区的概念，一个区就是在物理位置上**连续**的64个页。因为InnoDB 中的页大小默认是16KB，所以一个区的大小是64*16KB=1MB。在表中数据量大的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照区为单位分配，甚至在表中的数据特别多的时候，可以一次性分配多个连续的区。虽然可能造成一点点空间的浪费（数据不足以填充满整个区），但是从性能角度看，可以消除很多的随机I/O，功大于过！

> 这里是连续的64个页， 但是具体的两个页之间还是用指针相连的。保证一大块区域连续。

4.2为什么要有段?

对于范围查询，其实是对B+树叶子节点中的记录进行顺序扫描，而如果不区分叶子节点和非叶子节点，统统把节点代表的页面放到申请到的区中的话，进行范围扫描的效果就大打折扣了。所以InnoDB对B+树的`叶子节点`和`非叶子节点`进行了区别对待，也就是说叶子节点有自己独有的区，非叶子节点也有自己独有的区。存放叶子节点的区的集合就算是一个`段( segment)`，存放非叶子节点的区的集合也算是一个段。也就是说一个索引会生成2个段，一个`叶子节点段`，一个`非叶子节点段`。

除了索引的叶子节点段和非叶子节点段之外，InnoDB中还有为存储一些特殊的数据而定义的段，比如回滚段。所以，常见的段有`数据段`、`索引段`、`回滚段`。数据段即为B+树的叶子节点，索引段即为B+树的非叶子节点。

在InnoDB存储引擎中，对段的管理都是由引擎自身所完成，DBA不能也没有必要对其进行控制。这从一定程度上简化了DBA对于段的管理。

段其实不对应表空间中某一个连续的物理区域，而是一个逻辑上的概念，由若干个零散的页面以及一些完整的区组成。

> 零散的页面，看碎片区

4.3为什么要有碎片区?

默认情况下，一个使用InnoDB存储引擎的表只有一个聚簇索引，一个索引会生成2个段，而段是以区为单位申请存储空间的，一个区默认占用1M (64*16Kb=1024Kb）存储空间，所以**默认情况下一个只存了几条记录的小表也需要2M的存储空间么?**以后每次添加一个索引都要多申请2M的存储空间么?这对于存储记录比较少的表简直是天大的浪费。这个问题的症结在于到现在为止我们介绍的区都是非常**纯粹的**，也就是一个区被整个分配给某一个段，或者说区中的所有页面都是为了存储同一个段的数据而存在的，即使段的数据填不满区中所有的页面，那余下的页面也不能挪作他用。

为了考虑以完整的区为单位分配给某个段对于`数据量较小`的表太浪费存储空间的这种情况，InnoDB提出了一个`碎片(fragment)区`的概念。在一个碎片区中，并不是所有的页都是为了存储同一个段的数据而存在的，而是碎片区中的页可以用于不同的目的，比如有些页用于段A，有些页用于段B，有些页甚至哪个段都不属于。`碎片区直属于表空间`，并不属于任何一个段。

所以此后为某个段分配存储空间的策略是这样的:

- 在刚开始向表中插入数据的时候，段是从某个碎片区以单个页面为单位来分配存储空间的
- 当某个段已经占用了`32个碎片区`页面之后，就会申请以完整的区为单位来分配存储空间。

所以现在段不能仅定义为是某些区的集合，更精确的应该是**某些零散的页面**以及**一些完整的区**的集合。

4.4区的分类

区大体上可以分为4种类型:

- `空闲的区(FREE)`: 现在还没有用到这个区中的任何页面。
- `有剩余空间的碎片区(FREE_FRAG)：` 表示碎片区中还有可用的页面。
- `没有剩余空间的碎片区(FULL_FRAG)`：表示碎片区中的所有页面都被使用，没有空闲页面。
- `附属于某个段的区(FSEG)：`每一个索引都可以分为叶子节点段和非叶子节点段。

处于`FREE`、`FREE_FRAG`以及`FULL_FRAG`这三种状态的区都是独立的，直属于表空间。而处于FSEG状态的区是附属于某个段的。

> 如果把表空间比作是一个集团军，段就相当于师，区就相当于团。一般的团都是隶属于某个师的，就像是处于`FSEG`的区全都隶属于某个段，而处于`FREE`、`FREE_FRAG`以及`FULL_FRAG`这三种状态的区却直接隶属于表空间，就像独立团直接听命于军部一样。

4.n 扩展

那么，计算机怎样才能判断一个数据接下来可能被用到?

> 时间局部性（Temporal Locality）

时间局部性：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。

// 这是可以理解的，用过的数据当然可能再次被用到。

> 空间局部性（Spatial Locality）

空间局部性：在最近的将来将用到的信息很可能与现在正在使用的信息在空间地址上是临近的。

// 正在使用的某个数据地址旁边的数据，当然也是很可能被用到的，比如某个数组、集合等等。

> 顺序局部性（Order Locality）

顺序局部性：在典型程序中，除转移类指令外，大部分指令是顺序进行的。顺序执行和非顺序执行的比例大致是5:1。此外，对大型数组访问也是顺序的。

指令的顺序执行、数组的连续存放等是产生顺序局部性的原因。

// 正在执行的某个指令以及还在排队等候处理的指令，大部分是按照顺序来执行的。

> 磁盘预读原理

内存比磁盘的读写速度要快很多，但内存容量要远小于磁盘，而数据、程序的执行要调入内存后才能执行，所以内存和磁盘要经常进行I/O操作，I/O操作是个费事的过程，虽然现代系统已经有了通道（I/O处理机）技术的支持，但这远远不够（CPU的处理速度远远大于磁盘I/O的速度）。

所以磁盘读取的时候会顺带加载附近的数据到缓存

> 磁盘读取（详细）

磁盘存取，磁盘I/O涉及机械操作。磁盘是由大小相同且同轴的圆形盘片组成，磁盘可以转动(各个磁盘须同时转动)。磁盘的一侧有磁头支架，磁头支架固定了一组磁头，每个磁头负责存取一个磁盘的内容。磁头不动，磁盘转动，但磁臂可以前后动，用于读取不同磁道上的数据。磁道就是以盘片为中心划分出来的一系列同心环。磁道又划分为一个个小段，叫扇区，是磁盘的最小存储单元。

磁盘读取时，系统将数据逻辑地址传给磁盘，磁盘的控制电路会解析出物理地址（哪个磁道，哪个扇区），于是磁头需要前后移动到相应的磁道——寻道，消耗的时间叫——寻道时间，磁盘旋转将对应的扇区转到磁头下（磁头找到对应磁道的对应扇区），消耗的时间叫——旋转时间，这一系列操作是非常耗时。

重点

为了尽量减少I/O操作，计算机系统一般采取预读的方式，预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后`连续读取一页或几页载入内存中`，然后异常返回，程序继续运行。

计算机系统是分页读取和存储的，一般一页为4KB（8个扇区，每个扇区125B，8*125B=4KB），每次读取和存取的最小单元为一页，而**磁盘预读时通常会读取页的整倍数**。根据文章上述的【局部性原理】①当一个数据被用到时，其附近的数据也通常会马上被使用。②程序运行期间所需要的数据通常比较集中。由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），所以即使只需要读取一个字节，磁盘也会读取一页的数据。

至于磁盘分页，参考计算机操作系统的分页，分段存储管理——逻辑地址和物理地址被分为大小相同的页面，逻辑地址中叫页，物理地址中叫块。

为什么使用B-Tree/B+Tree

二叉查找树进化品种的红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-Tree/B+Tree作为索引结构。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

分析B-Tree/B+Tree检索一次最多需要访问节点：

h=

$$\log_{\lceil m/2 \rceil}(\frac{n+1}{2})+1$$

数据库系统巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

> 每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O（h）=O（logmN）。一般实际应用中，m是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O（h），效率明显比B-Tree差很多。

B树与B+Tree

B-Tree：如果一次检索需要访问4个节点，数据库系统设计者利用磁盘预读原理，把节点的大小设计为一个页，那读取一个节点只需要一次I/O操作，完成这次检索操作，最多需要3次I/O(根节点常驻内存)。数据记录越小，每个节点存放的数据就越多，树的高度也就越小，I/O操作就少了，检索效率也就上去了。

B+Tree：非叶子节点只存key，大大滴减少了非叶子节点的大小，那么每个节点就可以存放更多的记录，树更矮了，I/O操作更少了。所以B+Tree拥有更好的性能。

5.表空间

表空间可以看做是InnoDB存储引擎逻辑结构的最高层，所有的数据都存放在表空间中。

表空间是一个`逻辑容器`，表空间存储的对象是段，在一个表空间中可以有一个或多个段，但是一个段只能属于一个表空间。表空间数据库由一个或多个表空间组成，表空间从管理上可以划分为`系统表空间` (System tablespace)、`独立表空间` (File-per-table tablespace)、`撤销表空间` (Undo Tablespace)和`临时表空间` (Temporary Tablespace) 等。

5.1独立表空间

独立表空间，即每张表有一个独立的表空间，也就是数据和索引信息都会保存在自己的表空间中。独立的表空间(即:单表)可以在不同的数据库之间进行`迁移`。

空间可以回收(DROPTABLE操作可自动回收表空间;其他情况，表空间不能自己回收)。如果对于统计分析或是日志表，删除大量数据后可以通过: `alter table TableName engine=innodb`;回收不用的空间。对于使用独立表空间的表，不管怎么删除，表空间的碎片不会太严重的影响性能，而且还有机会处理。

独立表空间结构

独立表空间由段、区、页组成。前面已经讲解过了。

真实表空间对应的文件大小 我们到数据目录里看，会发现一个新建的表对应的`.ibd`文件只占用了`96K`，才6个页面大小(MySQL5.7中)，这是因为一开始表空间占用的空间很小，因为表里边都没有数据。不过别忘了这些.ibd文件是`自扩展的`，随着表中数据的增多，表空间对应的文件也逐渐增大。

查看InnoDB的表空间类型:

```
# 查看是否独立表空间
mysql> show variables like 'innodb_file_per_table';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_file_per_table | ON    |
+-----------------------+-------+
1 row in set, 1 warning (0.00 sec)
```

MySQL8.0中 7个页面大小。原因.idb 还存了 表结构。。。表结构.frm取消了

5.2系统表空间

系统表空间的结构和独立表空间基本类似，只不过由于整个MySQL进程只有一个系统表空间，在系统表空间中会额外记录一些有关整个系统信息的页面，这部分是独立表空间中没有的。

InnoDB数据字典

每当我们向一个表中插入一条记录的时候，`MySQL校验过程`如下:

先要校验一下插入语句对应的表存不存在，插入的列和表中的列是否符合，如果语法没有问题的话，还需要知道该表的聚簇索引和所有二级索引对应的根页面是哪个表空间的哪个页面，然后把记录插入对应索引的B+树中。所以说，MySQL除了保存着我们插入的用户数据之外，还需要保存许多额外的信息，比方说:

```
-某个表属于哪个表空间，表里边有多少列
-表对应的每一个列的类型是什么
-该表有多少索引，每个索引对应哪几个字段，该索引对应的根页面在哪个表空间的哪个页面
-该表有哪些外键，外键对应哪个表的哪些列
-某个表空间对应文件系统上文件路径是什么
- ...
```

上述这些数据并不是我们使用`INSERT`语句插入的用户数据，实际上是为了更好的管理我们这些用户数据而不得已引入的一些额外数据，这些数据也称为`元数据`。InnoDB存储引擎特意定义了一些列的`内部系统表`(internalsystem table)来记录这些这些元数据:

表名	描述
`SYS_TABLES`	整个InnoDB存储引擎中所有的表的信息
`SYS_COLUMNS`	整个InnoDB存储引擎中所有的列的信息
`SYS_INDEXES`	整个InnoDB存储引擎中所有的索引的信息
`SYS_FIELDS`	整个InnoDB存储引擎中所有的索引对应的列的信息
SYS_FOREIGN	整个InnoDB存储引擎中所有的外键的信息
SYS_FOREIGN_COLS	整个InnoDB存储引擎中所有的外键对应列的信息
SYS_TABLESPACES	整个InnoDB存储引擎中所有的表空间信息
SYS_DATAFILES	整个InnoDB存储引擎中所有的表空间对应文件系统的文件路
SYS_VIRTUAL	整个InnoDB存储引擎中所有的虚拟生成列的信息

这些系统表也被称为`数据字典`，它们都是以`B+`树的形式保存在系统表空间的某些页面中，其中`SYS_TABLES`.`SYS_COLUNNS`、`SYS_INDEXES`、`SYS_FIELDS`这四个表尤其重要，称之为基本系统表(basic system tables)

注意:用户是`不能直接访问`InnoDB的这些内部系统表，除非你直接去解析系统表空间对应文件系统上的文件。不过考虑到查看这些表的内容可能有助于大家分析问题，所以在系统数据库`information_schema`中提供了一些以innodb_sys开头的表:

```
mysql> USE information_schema ;
Database changed
mysql> SHOW TABLES LIKE 'innodb_sys%';
+-----------------------------------------+
| Tables_in_information_schema (innodb_sys%) |
+-----------------------------------------+
| INNODB_SYS_DATAFILES                    |
| INNODB_SYS_VIRTUAL                      |
| INNODB_SYS_INDEXES                      |
| INNODB_SYS_TABLES                       |
| INNODB_SYS_FIELDS                       |
```

```
| INNODB_SYS_TABLESPACES                     |
| INNODB_SYS_FOREIGN_COLS                    |
| INNODB_SYS_COLUMNS                         |
| INNODB_SYS_FOREIGN                         |
| INNODB_SYS_TABLESTATS                      |
+--------------------------------------------+
10 rows in set (0.00 sec)
```

在 `information_schema` 数据库中的这些以 `INNODB_SYS` 开头的表并不是真正的内部系统表(内部系统表就是我们上边以SYS开头的那些表)，而是在存储引擎启动时读取这些以 `SYS` 开头的系统表，然后填充到这些以 `INNODB_SYS` 开头的表中。以 `INNODB_SYS` 开头的表和以 `SYS` 开头的表中的字段并不完全一样，但供大家参考已经足矣。

附录:数据页加载的三种方式

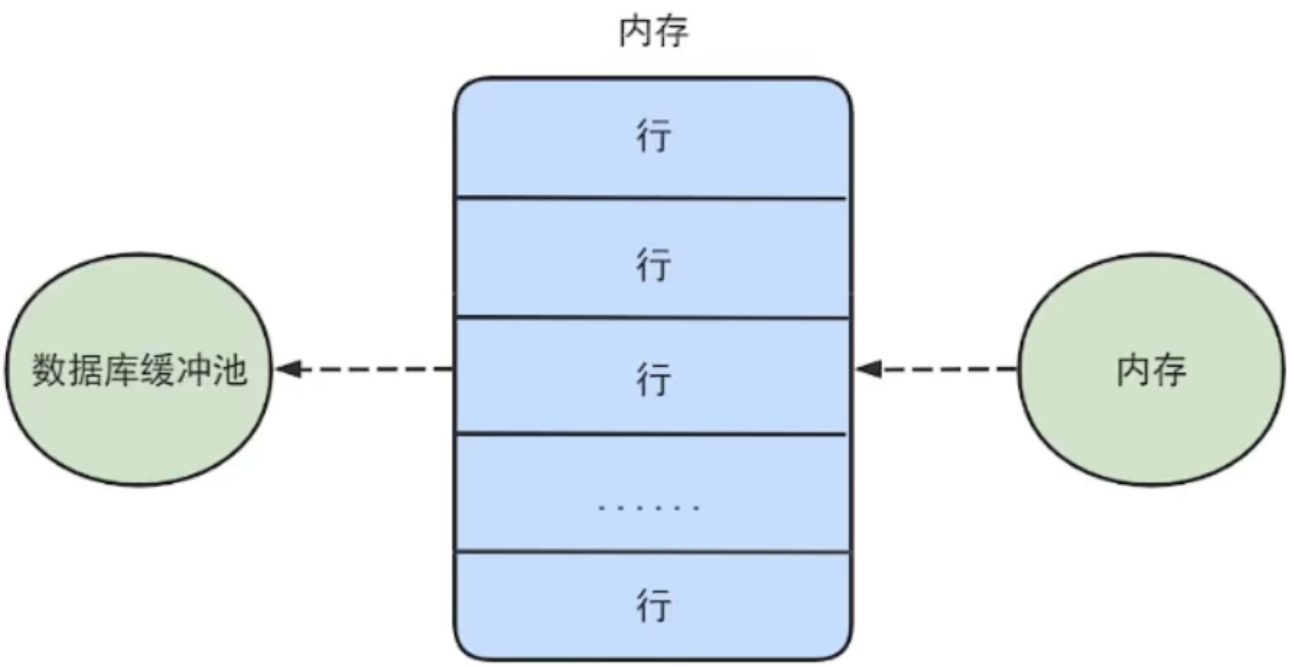
InnoDB从磁盘中读取数据的 `最小单位` 是数据页。而你想得到的id = xoxx的数据，就是这个数据页众多行中的一行。

对于MySQL存放的数据，逻辑概念上我们称之为表，在磁盘等物理层面而言是 `按数据页` 形式进行存放的，当其加载到MysQL中我们称之为 `缓存页` 。

如果缓冲池中没有该页数据，那么缓冲池有以下三种读取数据的方式,每种方式的读取效率都是不同的:

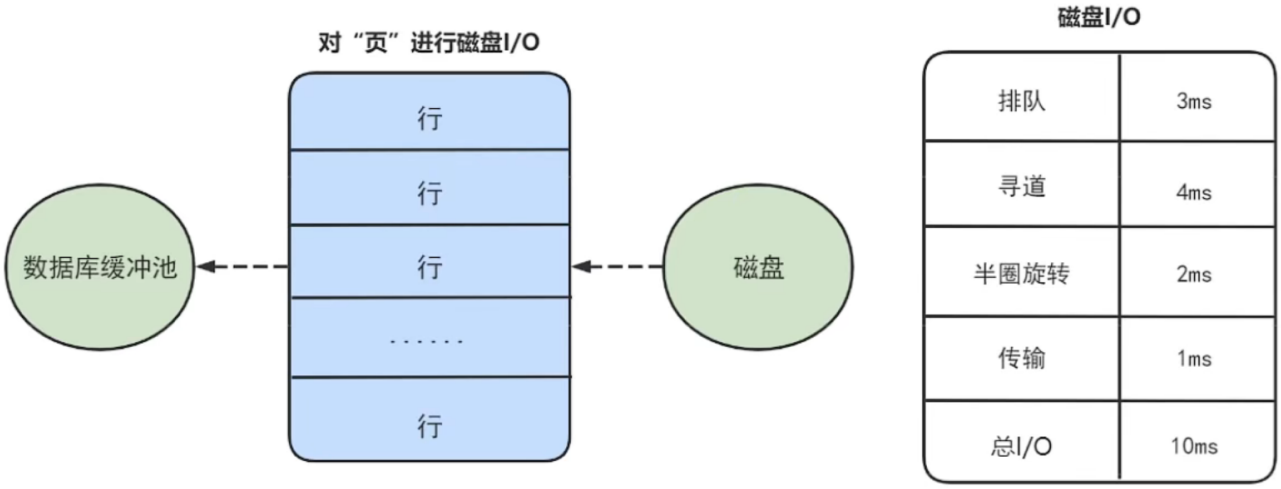
1.内存读取

如果该数据存在于内存中，基本上执行时间在1ms左右，效率还是很高的。



2.随机读取

如果数据没有在内存中，就需要在磁盘上对该页进行查找，整体时间预估在`10ms`左右，这10ms 中有6ms是磁盘的实际繁忙时间(包括了`寻道和半圈旋转时间`），有3ms是对可能发生的排队时间的估计值，另外还有1ms的传输时间，将页从磁盘服务器缓冲区传输到数据库缓冲区中。这10ms 看起来很快，但实际上对于数据库来说消耗的时间已经非常长了，因为这还只是一个页的读取时间。



3.顺序读取

顺序读取其实是一种批量读取的方式，因为我们请求的`数据在磁盘上往往都是相邻存储的`，顺序读取可以帮我们批量读取页面，这样的话，一次性加载到缓冲池中就不需要再对其他页面单独进行磁盘I/O操作了。如果一个磁盘的吞吐量是40MB/S，那么对于一个16KB大小的页来说，一次可以顺序读取2560 (40MB/16KB)个页，相当于一个页的读取时间为0.4ms。采用批量读取的方式，即使是从磁盘上进行读取，效率也比从内存中只单独读取一个页的效率要高。