

MySQL训练营-MySQL架构初始与索引机制

本内容属于动脑学院Steven老师首创,如需转载,请注明出处!

Steven QQ : 750954883

动脑学院系列课程咨询: QQ 2729772006

MySQL现状

MySQL数据库是风靡全球的关系型数据库

全球各大互联网公司都使用MySQL数据库并作为核心数据库.且这个趋势一旦形成不会发生改变.

- 全球顶级20大互联网网站有18家是使用MySQL的.
MySQL:google,facebook,youtube,yahoo,baidu,QQ,Wiki,Linkedin,Twitter,Amazon,taobao,sina,hao123等
SQLServer: live.com Bing
- MySQL与大数据平台是紧密关联的.MySQL是负责内容生产,hadoop负责内容消费.形成了OLTP系统使用MySQL产生数据,OLAP使用hadoop消费,分析数据的趋势

OLTP, 也叫联机事务处理 (Online Transaction Processing) , 表示事务性非常高的系统, 一般都是高可用的在线系统, 以小的事务以及小的查询为主

OLAP, 也叫联机分析处理 (Online Analytical Processing) 系统, 就是我们说的数据仓库

且这个习惯一旦养成,只会越来越多的追随者,在短时间内根本无法超越和改变.

MySQL发展历史和各版本

发展历史

MySQL数据库是芬兰人 Monty Widenius创建.最早历史可追溯到1979年

1996年MySQL1.0版本发布仅支持solaris系统.后续Monty开始将MySQL移植到Linux以及其他操作系统

2001年发布MySQL 3.23版本多操作系统的支持,如Linux, freeBSD,Solaris,Windows,且MySQL与Linux操作系统配合,占据了大量的开源市场,慢慢成就了目前的霸主地位.

Yahoo是最早使用MySQL大型互联网企业之一.

1995年-2008年,MySQL完全是基于社区驱动的方式进行快速发展.隶属MySQL AB这个公司.

2008年-2010年,MySQL AB被Sun公司以10亿美元收购

2010年-至今, Oracle 以74亿美金收购了Sun公司.

由于MySQL是开源的项目,MySQL AB公司坎坷经历.

在 MySQL 两次易主的过程中, 开发者们普遍担心某一天 MySQL 会成为一个非开源的收费版本
所以MySQL有较多的分支版本.

- **Oracle MySQL** 官方MySQL数据库版本,默认大家耳濡目染的版本
- **MariaDB** Monty于2009年创建的分支版本.一个独立的平行版本.
(创立之初的初心是作为一个MySQL的分支且兼容MySQL,但是从MariaDB10.0版本之后的路,他们在很多的实现上面已经出现了不同的方案)
Monty以他孙女命名的分支.
- **Percona** 由前MySQL性能团队创建的一家MySQL技术服务公司.打造的一个分支版本.
5.6版本之后渐渐的演化成MySQL工具的提供者.
- **Drizzle** 由前MySQL CTO创建的MySQL分支版本.
这个版本学术氛围比较重.各功能不全.可作为研究方向.

默认首选 **Oracle MySQL**,为什么呢?因为其他的存储引擎都缺少一个非常重要的组件-->插拔式的存储引擎的设计.

我们使用的最多的支持事务的存储引擎InnoDB 是Oracle MySQL公司

在官网中下载页面中[<https://www.mysql.com/downloads/>]

我们发现MySQL有MySQL Enterprise Edition版本.MySQL Cluster CGE版本 和MySQL Community 三个版本.

- MySQL Enterprise Edition
- MySQL Cluster CGE
- MySQL Community

<https://www.mysql.com/cn/trials/> 具体三版本的区别可参考此链接

说到性能优化,在大家脑海中第一想到的就是SQL优化或者说叫索引优化.

为什么会有这种意识呢?是因为这样的优化是最直接,最有效的优化方式.那接下来我们就一起了解一下关于索引的相关机制和内容.

MySQL索引

索引的本质及工作机制

索引的本质

索引是一种为了加速对数据库表中数据行的检索而创建的分散存储的**数据结构**.

注:数据库中的索引是**硬盘级索引**!

索引的工作机制

索引的工作机制

Select * from user where id = 102;

idx_userId(id)

数据结构	
101	0x123456
102	0x123457
103	0x123458
....	

表数据(User)

磁盘地址	id	name	age	phoneNum
0x123456	101	zhangsan	18	136*****
0x123457	102	lisi	19	138*****
0x123458	103	wangwu	20	139*****
0x154459	104	zhaoliu	21	131*****
0x923450	105	tianqi	22	132*****

索引的工作机制:

若id=102的数据在user进行比对,当数据量小的时候,我们服务器尚能在效率要求内返回,当我们的数据体达到千万亿级的时候,这种全表逐一比对的方式方法就显得捉襟见肘了.此时,必须引入索引进行数据的搜索的加速.

我们新建一个数据结构,存储id列与数据记录的磁盘指针对应关系..在SQL的执行过程,基于条件值,我可以快速在数据结构中找关键字内容对应的磁盘指针.再基于磁盘指针查找数据返回给请求方.这个过程即是索引的工作机制.

思考:

数据结构性能特点将决定SQL执行的性能特点.

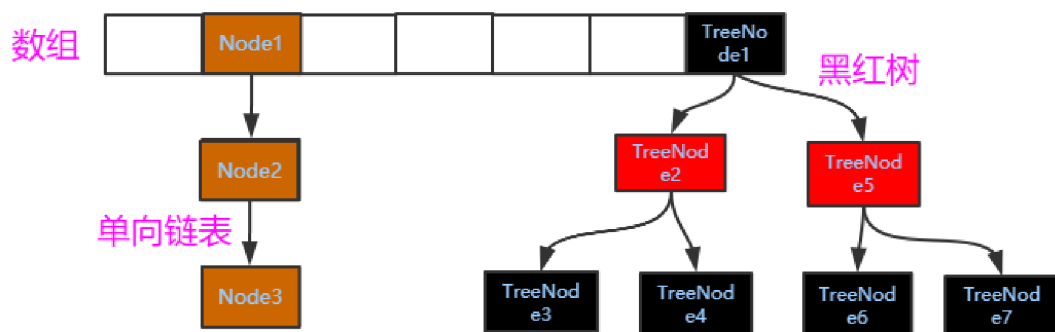
那哪些数据结构能带来数据检索的加速呢? ----> hash表, 二叉查找树, 平衡二叉查找树, B-树,B+树....接下来我们就逐一分析这些数据结构的特点.

MySQL为什么选择B+Tree索引

Hash索引

以底层数据结构为hashtable的索引结构.

我们以java的HashMap的底层数据结构示意图讲解:



我们在Hash表中进行数据的录入或者删除时,通用步骤:

1. 基于数据的key进行hashcode计算,得到int类型的数值
2. 基于hash int值与hash表的length进行计算得到下标位置.

3. 基于数组的下标进行数据查找,若查找的下标位存在元素,再判断该存储元素的数据结构
4. 如果存储元素是链表结构则顺序递归比对
5. 如果存储元素是红黑树则递归二分查找

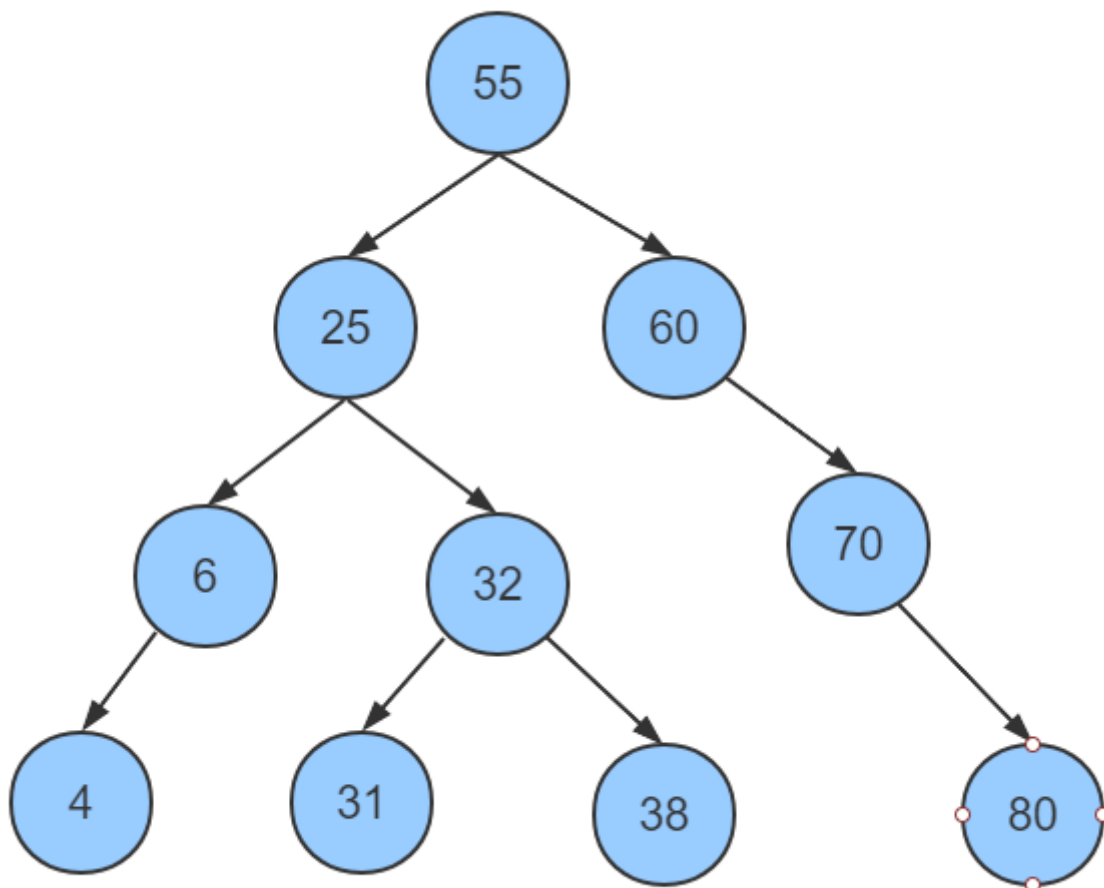
索引的底层数据结构为Hash表结构的话,我们在等值的匹配过程中只需要进行基于数组的下标查找.时间复杂度为 $O(1)$ 级别.但是我们也发现在存储元素进行hashcode计算之后,原本可比较大小的多个元素可能得到的hash int值确并不满足原本比较需求.

索引表索引的特点:

- 关键字等值匹配效率高
- 不支持数据的范围查找

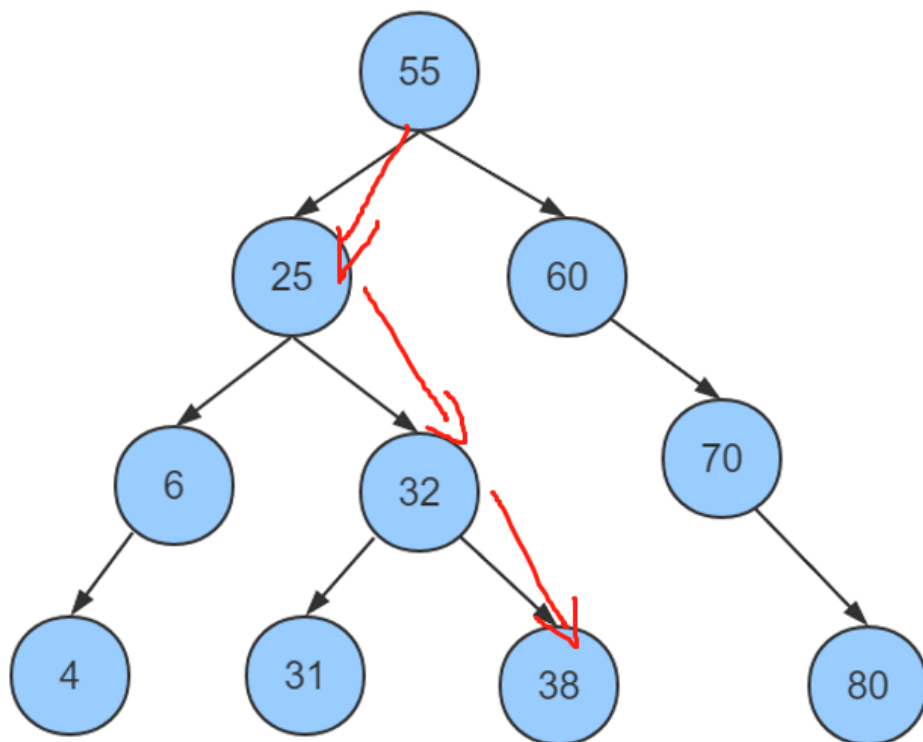
在MySQL各大存储引擎中,Memory引擎支持Hash索引.在我们的Innodb引擎存储中虽然没有用户操作级别的Hash索引,但是Innodb内存结构设计中,存在自适应的Hash索引.

二叉查找树



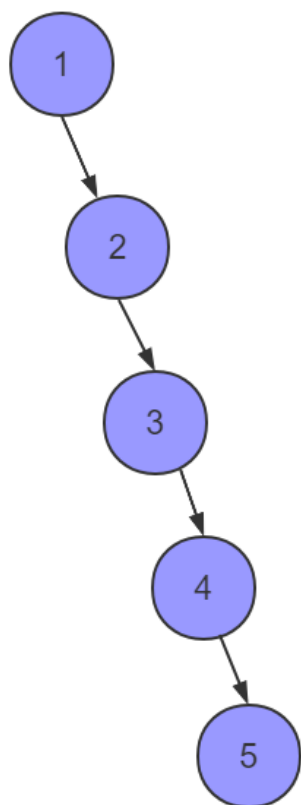
二叉树数据的组织方式是左小右大.即当前存入的数据比当前节点的关键字小,递归左子节点.若当前存入数据比当前节点的关键字大,递归右子节点.

数据的查找过程,我们发现基于二叉查找树的结构我们可以大大减少数据搜索量.如图,查找关键字为38的节点过程.



但是我们发现二叉查找树结构存在数据的倾斜(不规则)性.

若我针对表中ID字段建立了自增序列.若此时我选用二叉查找树数据结构作为索引的底层数据结构.针对ID的二叉树结构将变为线性链表结构.



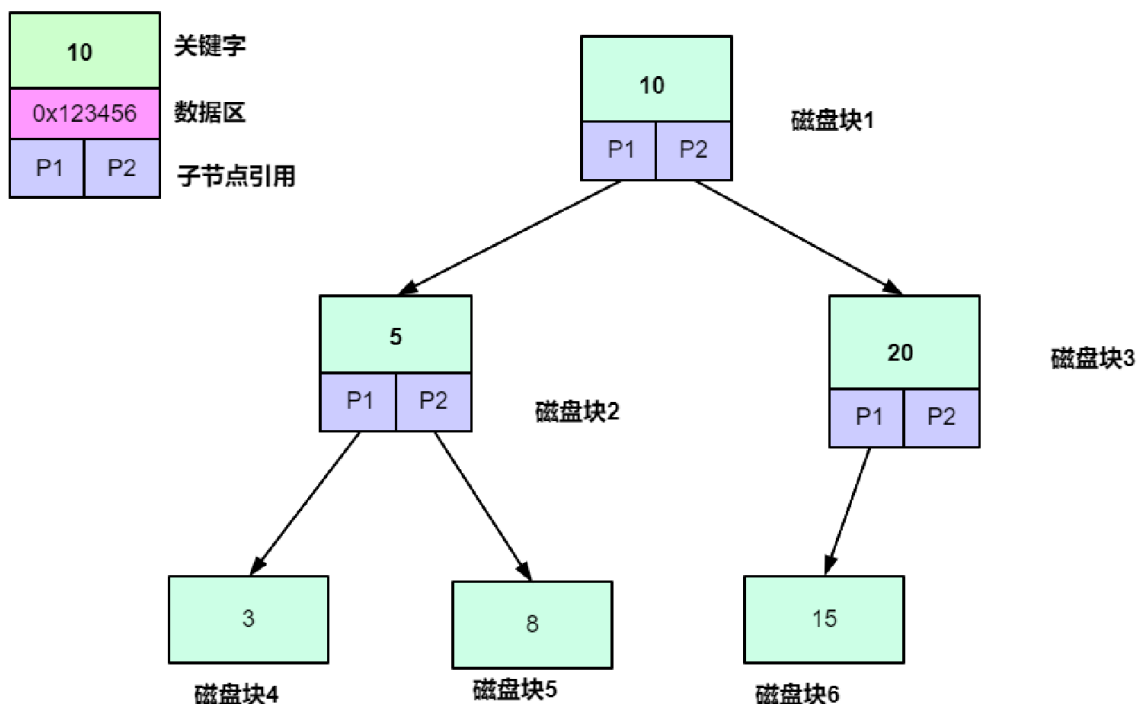
这是不是一颗二叉树?

若此时我们查找的数据处于线性链表结构的尾部,此时带来的时间消耗与我们的全表扫描从理论上讲并没有差别.

所以二叉查找树,存在数据组织过程中弊病.

平衡二叉树(AVL)

AVL树是最先发明的自平衡二叉查找树,在AVL树中任何节点的两个子树的高度最大差别为1, 所以它也被称为高度平衡树.



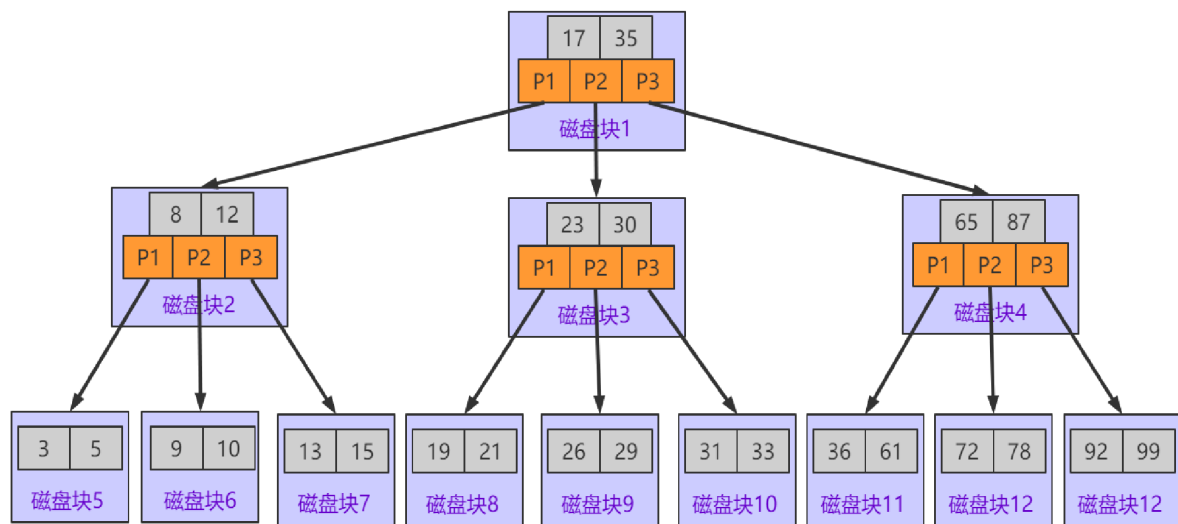
AVL为了保证树的平衡性,在数据的插入和删除的过程中,会进行一系列的旋转.将树进行左/右旋转满足树的平衡性.可以通过数据结构模拟的网站,进行验证.<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

既然AVL能保证树的平衡,那就解决了二叉查找树结构的线性链表问题,那么为什么MySQL并没有选择二叉结构AVL树呢?主要原因有以下两点:

- **AVL的树高问题.**在树形结构中,数据的所处树的位置将带来IO的加载次数.如上图,我们需要在图中查找8的数据,我们需要进行磁盘块1[10],磁盘块2[5],磁盘块5[8]的三次加载才能找到数据.AVL虽然能保证平衡,但是由于是二叉树结构必然在大数据量的情况下树高将会很高.
- **磁盘IO的利用率问题.**在前面我们已经普及到,我们的RDBMS系统中,索引都是存储在磁盘中的.操作系统与磁盘的交互是采用页为基本的交换单位.一页数据大小是4KB,再加上操作系统的预读能力[空间局部性原理],一次磁盘的IO交互将会带来N页的数据返回.但是在上图中,我们不难发现,一个磁盘块能承载的数据内容只有一个关键字,一个数据区内容,两个子节点的指针.肯定填充不满4KB或是8KB,16KB的空间.一次低性能的磁盘IO,确只能带来仅仅少量的有效数据.

多路平衡查找树(B-树)

B-树又叫多路平衡查找树,是一种绝对平衡的树形结构.他的绝对平衡指的是所有的叶子节点数据都同一个水平线上.



首先我们理解一下B-树是如何保证树的绝对平衡的.我们可以通过数据结构模拟网站<https://www.cs.usfca.edu/~galles/visualization/BTree.html> 进行模拟.通过模拟我们发现,数据在进行插入或是删除的过程中会基于节点个数的变化进行节点的合并和分裂.以达到树的绝对平衡.

B-树的优势

- 将磁盘IO的低效操作通过内存中数据比较进行替换.

在二叉树中,我们一次只能加载一个关键字进行匹配.但是在B-树,我们一次可以加载N个关键字,若我们将磁盘块(节点)的空间大小固定(MySQL中定义为16KB).磁盘块能存储的关键字个数就会与单个关键字内容占用的空间相关.基于预读和操作系统磁盘交互特性.我们磁盘IO一次加载的内容正好都是我们需要比对的内容.讲内容的多次IO加载转换成在内存中进行数据的比较

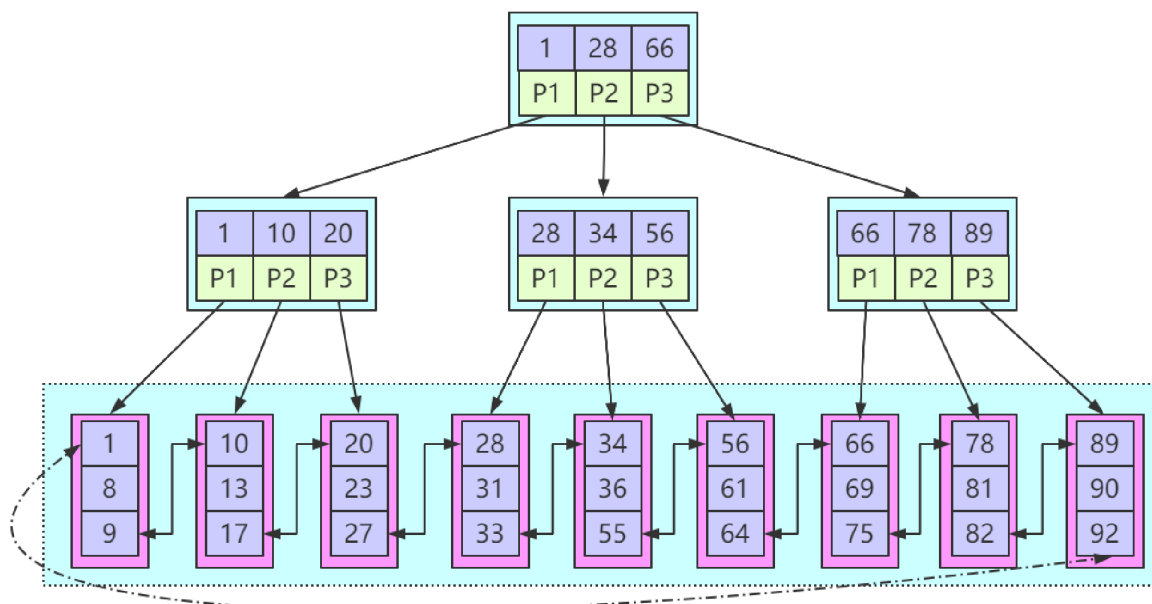
- 合理的降低树的高度,减少IO的次数

在树形结构中数据的所处高度位置,将带来IO的次数不一样.B-树结构合理的降低了树的高度.也就意味着数据的IO次数将降低

基于以上的特点,所以很多关系型数据库都采用B-树,或是B-树的变种数据结构作为索引的底层数据结构.

在MySQL中采用加强版的B+树.在Oracle中采用B*树.等等

加强版的多路平衡查找树(B+树)



B+树是基于B-树结构的加强版树形结构.

- B-树拥有的特性B+树都拥有.
- B+树的数据匹配规则采用闭合区间的方式
- B+树的非叶子节点上不保存关键字对应的数据区
- B+树的叶子节点上保存数据区
- 在叶子节点上的数据产生形成首尾相连的链式结构带来更高效的数据排序

基于以上你的结构特点B+树相较于B-树又有哪些优势呢?

- B-树拥有的特性,B+树都拥有
- B+树拥有更强劲的磁盘IO能力.

在非叶子节点不存在数据区,将大大降低节点的空间占用.能存储更多的关键字.一次磁盘IO带回来的有效数据将更多更精准

- B+树拥有更好的数据排序能力

这是B+树的天然优势,在最末尾的叶子节点这一层天然即有序的链式结构

- 基于B+树的扫表能力更强

在B+树的数据结构中,若需要扫表,只需要扫描最末尾的叶子节点即可.

- 基于B+树结构的索引的查询,更趋于稳定

在B-树结构中,我们发现,我们的关键字在某一个节点进行匹配成功就完成数据区内容的返回.

但是在B+树结构中,我们采用的是闭合区间的比对方式即数据的最终加载一定会在叶子节点上.

在B-树结构中,有可能针对一张表的查询,有些数据需要3次IO.有些只需要1次IO.前后的查询效率跌宕起伏不稳定.但是在B+树结构中一定恒定的IO次数,带来更稳定的查询效率.

基于以上的对比和总结,MySQL最终选择了B+Tree作为索引的数据结构.

索引在存储引擎中的实现




前面我们已经了解到,在MySQL的体系中,插拔式的存储引擎设计是Oracle MySQL最为优秀的设计之一.从体系结构中我们也了解到跟file system(文件系统)打交道的是各存储引擎.前面我也讲到了.我们数据库的索引是硬盘级的索引.所以我们不难推导.索引的实现一定跟存储引擎相关.

接下来我们就一起来学习一下在MySQL的Myisam与Innodb引擎是如何落地索引的

使用下面SQL语句创建了2张表,分别用Myisam和Innodb进行表修饰.

```
CREATE TABLE `user_innodb` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(32) DEFAULT NULL,  
  `age` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `idx_name` (`name`) USING BTREE  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;  
  
CREATE TABLE `user_myisam` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(32) DEFAULT NULL,  
  `age` int(11) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
```

我们通过 `show variables like 'datadir'` 打开数据目录,发现基于这两张表产生了5个文件

 user_innodb.frm 类型: FRM 文件	修改日期: 2019/11/7 14:26 大小: 8.41 KB
 user_innodb.ibd 类型: IBD 文件	修改日期: 2020/2/10 12:57 大小: 696 MB
 user_myisam.frm 类型: FRM 文件	修改日期: 2019/5/31 17:43 大小: 8.41 KB
 user_myisam.MYD 类型: MYD 文件	修改日期: 2020/2/10 13:01 大小: 80 字节
 user_myisam.MYI 类型: MYI 文件	修改日期: 2020/2/10 13:01 大小: 2.00 KB

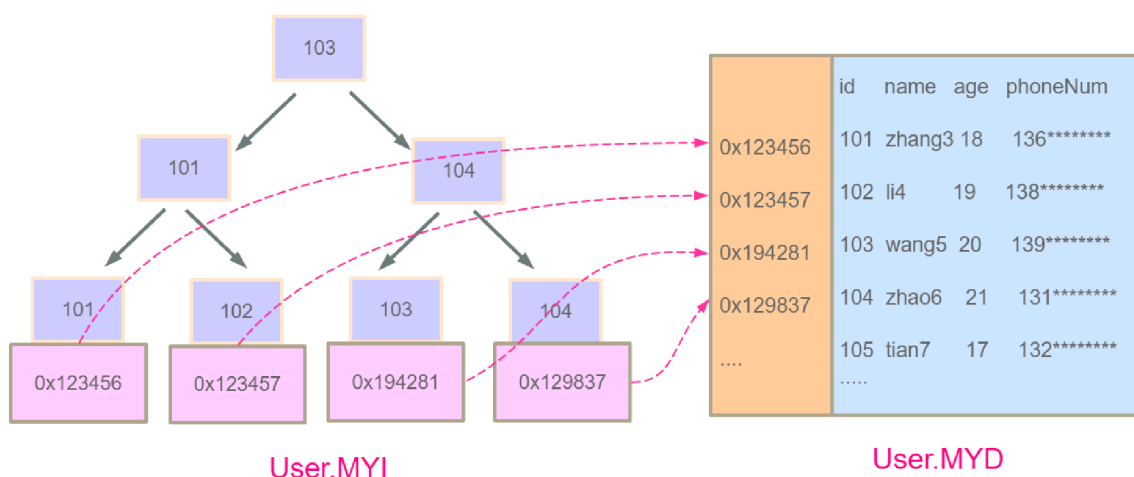
user_innodb.frm 和 user_myisam.frm 这两个文件代表的是表的结构定义文件.我们指定的表的相关属性如编码,最后更新的时间,表的结构等等都将保存在frm文件中.

Myisam

在数据目录中,我们发现除了frm文件之外,还额外存在user_myisam.MYD 和 user_myisam.MYI两个文件.

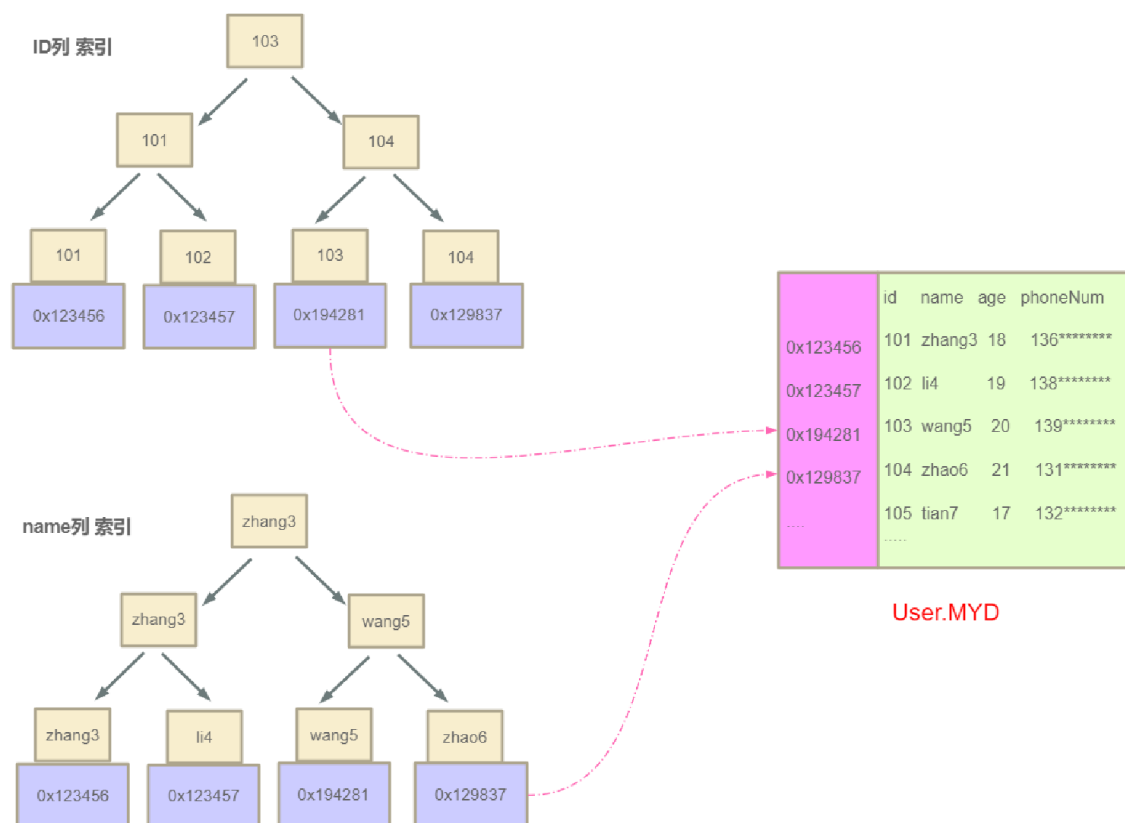
user_myisam.MYI user_myisam表的索引文件.即该表的B+Tree索引数据都将存储在这个文件中

user_myisam.MYD user_myisam表的数据文件.即该表的数据都将存储在这个文件中.



所以在Myisam引擎中,数据文件与索引文件是隔离存储的.在索引文件中,每一颗B+Tree树的叶子节点的数据区保存的是指向数据文件的数据行指针信息.

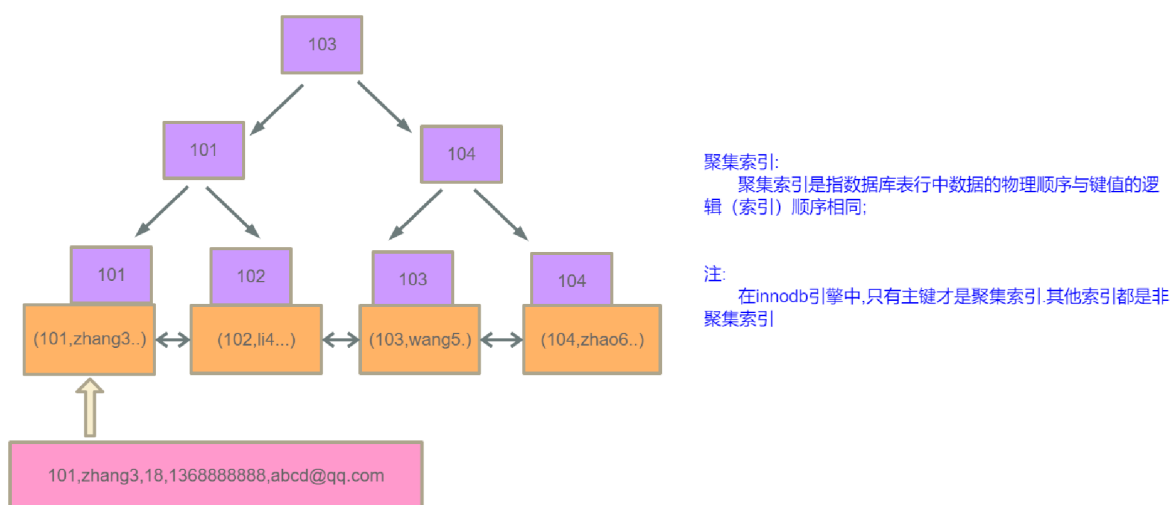
如果一个表存在多个索引即除了主键索引还存在额外的二级索引时如下图



通过图示我们了解到MYI文件中存储的是表的索引的结构.即主键索引,辅助索引(二级索引).在Myisam引擎中,并不存在有索引的主次之分.他们是一视同仁平等的关系

Innodb

在数据目录中,我们发现Innodb类型的表只有user_innodb.frm 和 user_innodb.ibd文件, frm文件我们已经介绍.是表的结构信息文件,那在Innodb中就只有一个IBD类型的文件,那Innodb中如何存储数据和索引的呢.



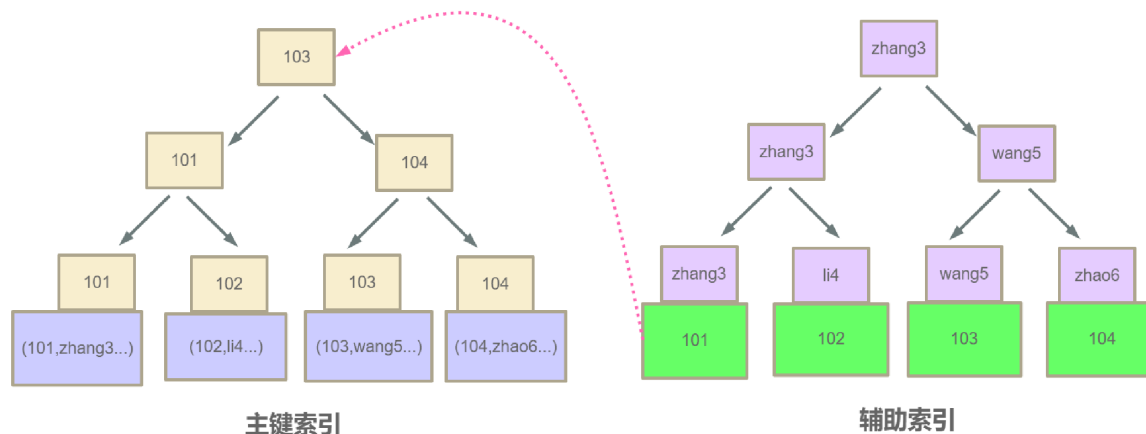
聚集索引:
聚集索引是指数据库表中数据的物理顺序与键值的逻辑(索引)顺序相同;

注:
在innodb引擎中,只有主键才是聚集索引.其他索引都是非聚集索引

原来,在Innodb存储引擎中,数据的存储是根据主键来进行存储的.在主键索引的叶子节点层存储的并不是数据的真实磁盘指针,而是存储的当前主键代表的数据行的所有内容.如上图,若User表中有5个字段分别是,ID、name、phoneNum、email.那么在整个叶子节点的数据区存储的就是该主键代表的数据行的所有字段的信息.

这种以主键索引进行数据整理的方式,其数据的物理排序跟主键的逻辑顺序必然会保持一致.所以也主键索引在Innodb中也叫聚集(簇)索引.且在Innodb引擎中有且只有一个聚集(簇)索引.所以在Innodb类型表的数据存储就只有一个IBD文件也就能解释了.

如果在Innodb类型的表中不仅有主键索引也包含其他的二级(辅助)索引.此时数据的搜索过程又是怎样的呢?



从上图中,我们知道.在辅助(二级)索引的叶子节点上保存的数据是主键索引的值.若此时我们的查询条件是基于name的条件进行查询,首先我们会基于辅助索引的树进行一次比对和查询,查询到结果之后,拿到主键索引的值.再基于主键索引进行二次搜索进行数据的返回.这样的二次扫描的过程我们称之为**回表**

思考题:为什么在辅助索引的叶子节点为什么存储的是主键的值,为什么不保存数据行的指针位置呢?为什么还需要基于主键进行**回表**操作?

是因为在前面的数据结构推演过程中,我们已经知道B-树,B+树,为了保证树的绝对平衡,会在数据的组织阶段进行一系列的计算和操作.如节点的合并,节点的分裂.这样的操作会带来的数据的指针位置的变化.在Innodb引擎中,一切都是以主键为基准进行设计和考量的.在我们的数据进行相应的变化之后.我们如果此时在辅助索引存储指针的话,我们势必要回头去修改所有的辅助(二级)索引叶子节点的内容.

索引的优化

列的离散性

离散度计算公式: $\text{count}(\text{distinct colName}) / \text{count}(\text{colName})$ 比值越大离散性越好.

离散性是衡量索引列选择的很重要的指标.是因为在查询的优化阶段,离散度不好的查询,优化器会自动基于成本的原由,摒弃离散性不好的索引查询.

如: `select * from user where name like 'hello-%'` 和 `select * from user where name like 'hello-123123%'`

如果我们遵循了like 后面的条件 % 写在最右,但是有可能在优化器介入的阶段,由于name字段的数据绝大多数都是以'hello-'开头,导致该列的唯一性选择很差,从而执行引擎在执行的过程中,并不使用name字段的索引.

```
1 EXPLAIN select * from user_innodb where name like 'hello-%';
```

id	select_type	table	partype	possible_key	key	rows	filtered	Extra
1	SIMPLE	user_innodb	ALL	idx_name (Null)	(Null)	37388		50 Using where

```
1 EXPLAIN select * from user_innodb where name like 'hello-123123%';
```

id	select_type	table	partype	possible_key	key	rows	filtered	Extra
1	SIMPLE	user_innodb	range	idx_name	idx_name	131	11	100 Using index condition

索引项关键字比对规则

在索引项中关键字的比对是从左往右依次进行.在没有强行指定数据的字符的编码排序规则时.默认采用ASCII的编码进行字符的比对.

比如:数字的比对 $100 > 20$

字符串的比对 "100" < "20"

"abc" < "adc"

那为什么 "abc" < "adc" 呢.首先abc 转换成ASCII编码 97 98 99 adc转换成ASCII编码 97 100 98

因为在字符的比较过程中是从左至右依次进行的.所以 abc<adc

联合索引

- 联合索引是多个索引么?

联合索引是一个索引,是由表字段中多列组成的索引项中关键字的一个索引.

例如.index(name) 我们索引项的关键字就是name的值.

index(name+phoneNum) 我们索引项的关键字就是 name+phoneNum的值.

基于前面我们介绍的索引项关键字比对规则.故而在联合索引中有最左前缀原则.即经常使用的字段需放置联合索引的最前面.若前面的字段查询条件不存在,不能使用到中间字段的进行索引扫描.

即:带头大哥不能死, 中间兄弟不能断;

- 联合索引与单列索引的关系?

单列索引是一种特殊的联合索引.

- 若一个表中创建了index(name,phoneNum) 和 index(name)两个索引是否有必要?

没有必要.通过前面我们已知索引项的排序是从左到右依次进行比对的.所以index(name,phoneNum)索引name字段的排序就包含了 index(name).所以index(name)是冗余索引.

- 联合索引使用过程中遵循哪些原则?

- 最左前缀

最常使用的字段放置联合索引的前面

- 离散度高

离散度高的字段选择性将会更好.

- 最少空间

我们在选择联合索引字段时,尽量选择占用空间少的字段.避免空间浪费.且能让我们单索引页数据能存储更多的关键字.

覆盖索引

通过索引项的信息可直接返回所需的查询列,则该索引称之为查询SQL的覆盖索引

表: Users innodb 引擎

索引: PK(id) key(name,phoneNum) unique(userNum)

下面哪些SQL使用了覆盖索引?

select userNum from teacher where userNum= ?

select * from teacher where name = ?

select id,userNum from teacher where userNum= ?

select name,phoneNum from teacher where userNum= ?

select phoneNum from teacher where name = ?

上述SQL的第1, 3, 5 条SQL 将使用覆盖索引.

覆盖索引能提高查询性能,减少不必要回表的操作.

具体案例:

不使用覆盖索引

```
1 select * from user_innodb where `name` like 'hello-3%';
```

查询时间: 3.574s

使用到覆盖索引

```
select `name` from user_innodb where `name` like 'hello-3%';
```

查询时间: 1.583s

结果显然可见速度提升一倍.

索引打油诗

- 全值匹配我最爱, 最左前缀要遵守;
- 带头大哥不能死, 中间兄弟不能断;
- 索引列上少计算, 范围之后全失效;

- Like百分写最右，覆盖索引不写星；
- 不等空值还有or，索引失效要少用。