Sql高级

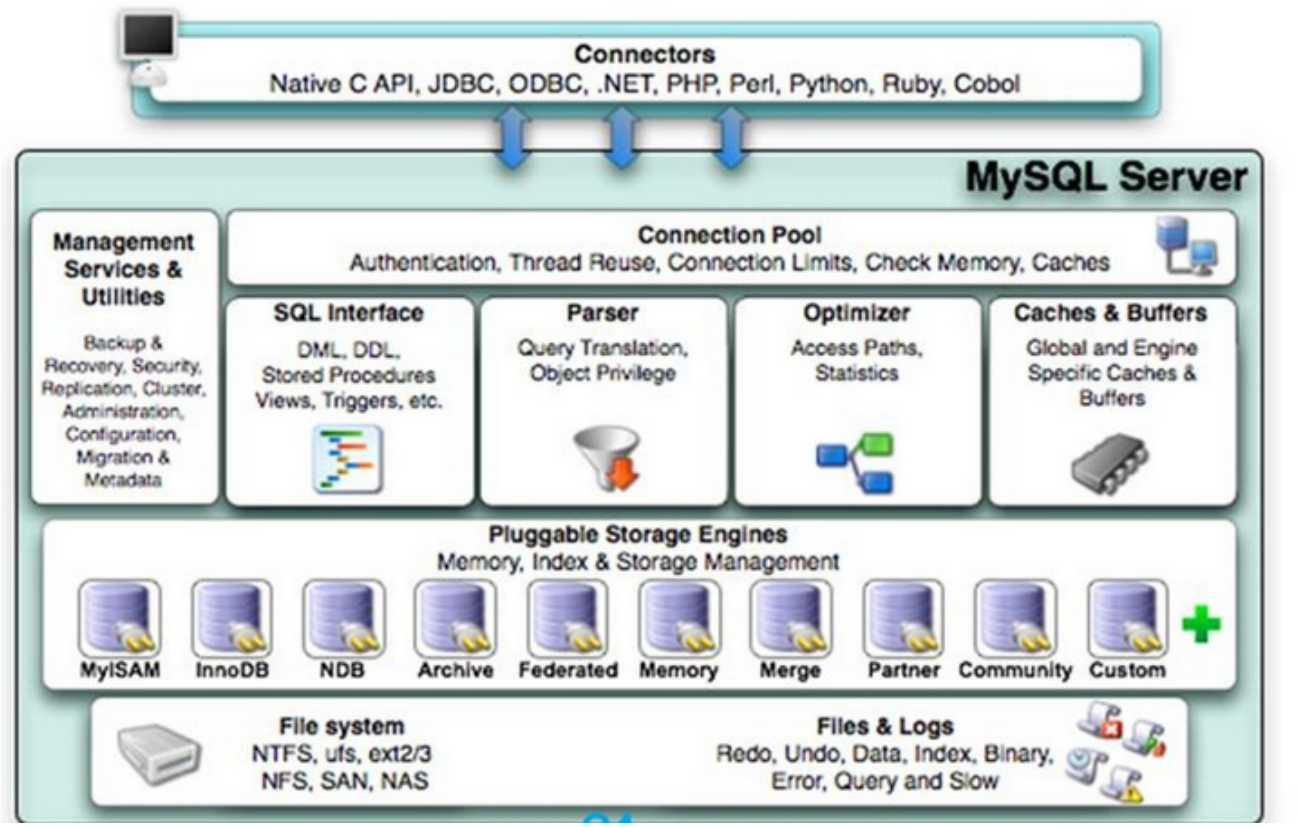
第一章-MySql逻辑架构和引擎

1.MySql逻辑架构

1.1逻辑架构

和其它数据库相比，MySQL有点与众不同，它的架构可以在多种不同场景中应用并发挥良好作用。主要体现在存储引擎的架构上，插件式的存储引擎架构将查询处理和其它的系统任务以及数据的存储提取相分离。这种架构可以根据业务的需求和实际需要选择合适的存储引擎。

2.MySQL逻辑架构



- 连接层

  最上层是一些客户端和连接服务，包含本地sock通信和大多数基于客户端/服务端工具实现的类似于tcp/ip的通信。主要完成一些类似于连接处理、授权认证、及相关的安全方案。在该层上引入了线程池的概念，为通过认证安全接入的客户端提供线程。同样在该层上可以实现基于SSL的安全链接。服务器也会为安全接入的每个客户端验证它所具有的操作权限。

- 服务层

  Management Serveices & Utilities： 系统管理和控制工具

  SQL Interface: SQL接口,接受用户的SQL命令，并且返回用户需要查询的结果。比如select from就是调用SQL Interface

  Parser: 解析器 SQL命令传递到解析器的时候会被解析器验证和解析。

Optimizer: 查询优化器。 SQL语句在查询之前会使用查询优化器对查询进行优化.用一个例子就可以理解：select uid,name from user where gender= 1; 优化器来决定先投影还是先过滤。

Cache和Buffer： 查询缓存。 如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等 缓存是负责读，缓冲负责写。

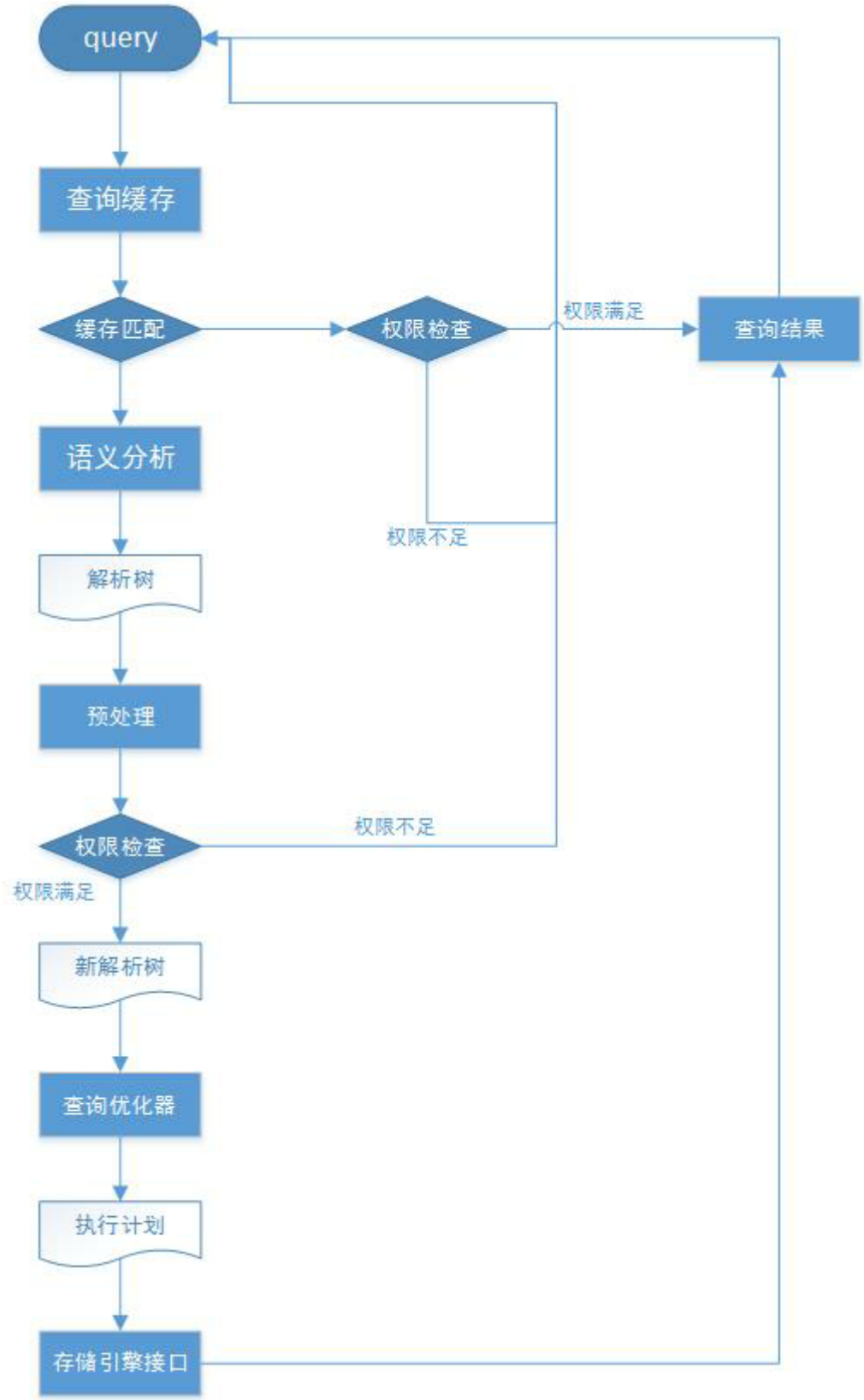
- 引擎层

  存储引擎层，存储引擎真正的负责了MySQL中数据的存储和提取，服务器通过API与存储引擎进行通信。不同的存储引擎具有的功能不同，这样我们可以根据自己的实际需要进行选取。后面介绍MyISAM和InnoDB

- 存储层

  数据存储层，主要是将数据存储在运行于裸设备的文件系统之上，并完成与存储引擎的交互。

1.2查询说明



首先，mysql的查询流程大致是：

- mysql客户端通过协议与mysql服务器建连接，发送查询语句，先检查查询缓存，如果命中(一模一样的sql才能命中)，直接返回结果，否则进行语句解析,也就是说，在解析查询之前，服务器会先访问查询缓存(query cache)——它存储SELECT语句以及相应的查询结果集。如果某个查询结果已经位于缓存中，服务器就不会再对查询进行解析、优化、以及执行。它仅仅将缓存中的结果返回给用户即可，这将大大提高系统的性能。

- 语法解析器和预处理：首先mysql通过关键字将SQL语句进行解析，并生成一颗对应的“解析树”。mysql解析器将使用mysql语法规则验证和解析查询；预处理器则根据一些mysql规则进一步检查解析数是否合法。
- 查询优化器当解析树被认为是合法的了，并且由优化器将其转化成执行计划。一条查询可以有很多种执行方式，最后都返回相同的结果。优化器的作用就是找到这其中最好的执行计划。。
- 然后，mysql默认使用的BTREE索引，并且一个大致方向是:无论怎么折腾sql，至少在目前来说，mysql最多只用到表中的一个索引。

2.Mysql存储引擎

2.1Mysql存储引擎介绍

1、InnoDB存储引擎

InnoDB是MySQL的默认事务型引擎，它被设计用来处理大量的短期(short-lived)事务。除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑InnoDB引擎。行级锁，适合高并发情况

2、MyISAM存储引擎
MyISAM提供了大量的特性，包括全文索引、压缩、空间函数(GIS)等，但MyISAM不支持事务和行级锁(myisam改表时会将整个表全锁住)，有一个毫无疑问的缺陷就是崩溃后无法安全恢复。

3、Archive引擎
Archive存储引擎只支持INSERT和SELECT操作，在MySQL5.1之前不支持索引。
Archive表适合日志和数据采集类应用。适合低访问量大数据等情况。
根据英文的测试结论来看，Archive表比MyISAM表要小大约75%，比支持事务处理的InnoDB表小大约83%。

4、Blackhole引擎
Blackhole引擎没有实现任何存储机制，它会丢弃所有插入的数据，不做任何保存。但服务器会记录Blackhole表的日志，所以可以用于复制数据到备库，或者简单地记录到日志。但这种应用方式会碰到很多问题，因此并不推荐。

5、CSV引擎
CSV引擎可以将普通的CSV文件作为MySQL的表来处理，但不支持索引。
CSV引擎可以作为一种数据交换的机制，非常有用。
CSV存储的数据直接可以在操作系统里，用文本编辑器，或者excel读取。

6、Memory引擎
如果需要快速地访问数据，并且这些数据不会被修改，重启以后丢失也没有关系，那么使用Memory表是非常有用。Memory表至少比MyISAM表要快一个数量级。(使用专业的内存数据库更快，如redis)

7、Federated引擎
Federated引擎是访问其他MySQL服务器的一个代理，尽管该引擎看起来提供了一种很好的跨服务器的灵活性，但也经常带来问题，因此默认是禁用的。

2.2MyISAM和InnoDB

对比项	MyISAM	InnoDB
主外键	不支持	支持
事务	不支持	支持
行表锁	表锁，即使操作一条记录也会锁住整个表，不适合高并发的操作	行锁,操作时只锁某一行，不对其它行有影响，适合高并发的操作
缓存	只缓存索引，不缓存真实数据	不仅缓存索引还要缓存真实数据，对内存要求较高，而且内存大小对性能有决定性的影响
表空间	大	小
关注点	性能	事务
默认安装	Y	Y
用户表默认使用	N	Y
自带系统表使用	Y	N

2.3查看引擎命令

- 命令

```
show variables like '%storage_engine%';
```

Variable_name	Value
default_storage_engine	InnoDB
storage_engine	InnoDB

第二章-Sql优化

1.Sql性能不高原因

- 查询数据过多(能不能拆，条件过滤尽量少)
- 关联了太多的表，太多join (join 原理。用 A 表的每一条数据 扫描 B表的所有数据。所以尽量先过滤。)
- 没有利用到索引
- 服务器调优及各个参数设置（缓冲、线程数等)

2.回顾Sql执行顺序

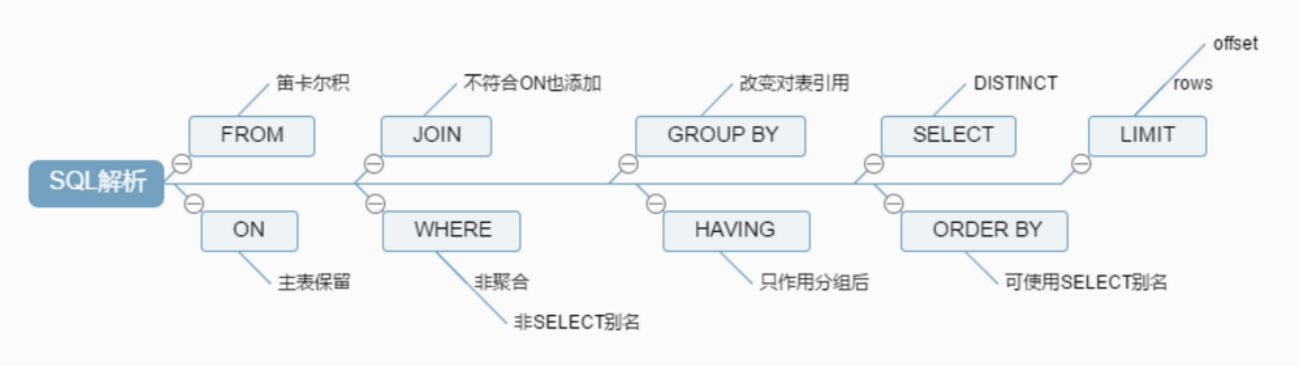
- 编写顺序

```
SELECT DISTINCT
    < select_list >
FROM
    < left_table > < join_type >
JOIN < right_table > ON < join_condition >
WHERE
    < where_condition >
GROUP BY
    < group_by_list >
HAVING
    < having_condition >
ORDER BY
    < order_by_condition >
LIMIT < limit_number >
```

- 机读顺序

```
1 FROM <left_table>
2 ON <join_condition>
3 <join_type> JOIN <right_table>
4 WHERE <where_condition>
5 GROUP BY <group_by_list>
6 HAVING <having_condition>
7 SELECT
8 DISTINCT <select_list>
9 ORDER BY <order_by_condition>
10 LIMIT <limit_number>
```

- Sql解析



3.索引

3.1什么是索引

索引（Index）是帮助MySQL高效获取数据的数据结构。在数据之外,数据库系统还维护着满足特定查找算法的数据结构,这些数据结构以某种方式指向数据,这样就可以在这些数据结构上实现高效的查找算法.这种数据结构,就是索引.

一般来说索引本身也很大,不可能全部存储在内存中,因此往往以索引文件的形式存放在磁盘中. 我们平常所说的索引,如果没有特别说明都是指BTree索引(平衡多路搜索树). 其中聚集索引,次要索引,覆盖索引 复合索引,前缀索引,唯一索引默认都是使用的BTree索引,统称索引. 除了BTree索引之后,还有哈希索引

3.2索引原理

3.2.1说明

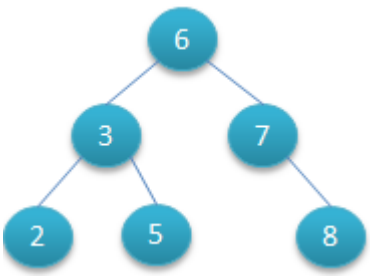
索引（Index）是帮助MySQL高效获取数据的数据结构。不同的引擎使用的数据结构也不尽相同.

MyISAM采取的是BTree, InnoDB采取的是B+TREE

我们工作的里面的数据库一般用的是InnoDB, 所以我们重点来讲解B+TREE. B+树索引是B+树在数据库中的一种实现，是最常见也是数据库中使用最为频繁的一种索引。B+树中的B代表平衡（balance），而不是二叉（binary），因为B+树是从最早的平衡二叉树演化而来的。在讲B+树之前必须先了解二叉查找树、平衡二叉树（AVLTree）和平衡多路查找树（B-Tree），B+树即由这些树逐步优化而来。

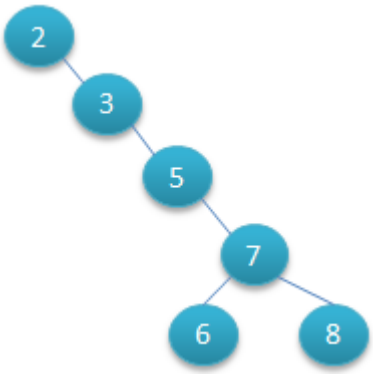
3.2.2二叉查找树

二叉树具有以下性质：左子树的键值小于根的键值，右子树的键值大于根的键值。
如下图所示就是一棵二叉查找树，



对该二叉树的节点进行查找发现深度为1的节点的查找次数为1，深度为2的查找次数为2，深度为n的节点的查找次数为n，因此其平均查找次数为 (1+2+2+3+3+3) / 6 = 2.3次

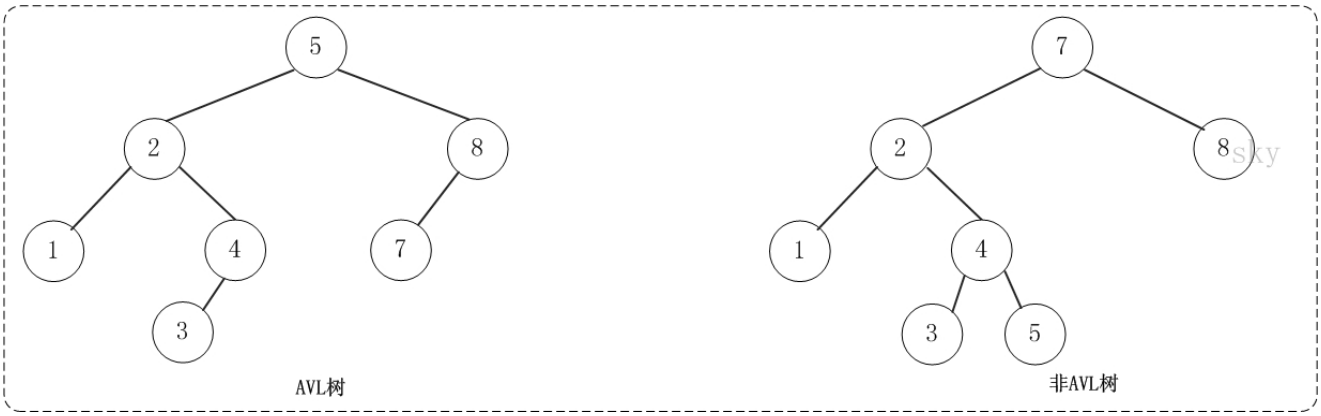
二叉查找树可以任意地构造，同样是2,3,5,6,7,8这六个数字，也可以按照下图的方式来构造：



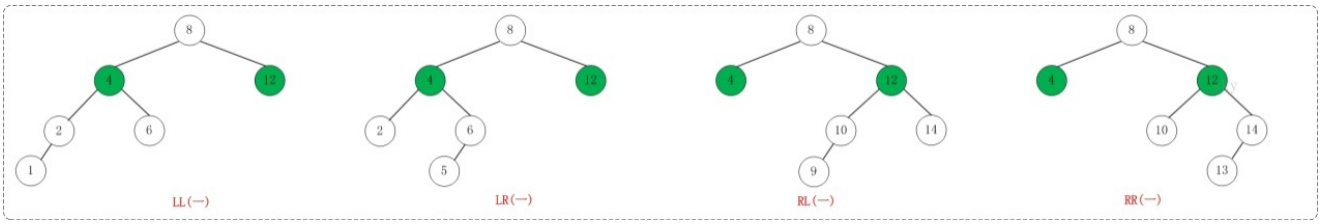
但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义——平衡二叉树，或称AVL树。

3.2.3平衡二叉树（AVL Tree）

平衡二叉树（AVL树）在符合二叉查找树的条件下，还满足任何节点的两个子树的高度最大差为1。下面的两张图片，左边是AVL树，它的任何节点的两个子树的高度差<=1；右边的不是AVL树，其根节点的左子树高度为3，而右子树高度为1；



如果在AVL树中进行插入或删除节点，可能导致AVL树失去平衡，这种失去平衡的二叉树可以概括为四种姿态：LL（左左）、RR（右右）、LR（左右）、RL（右左）。它们的示意图如下：



这四种失去平衡的姿态都有各自的定义：
LL：LeftLeft，也称“左左”。插入或删除一个节点后，根节点的左孩子（Left Child）的左孩子（Left Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

RR：RightRight，也称“右右”。插入或删除一个节点后，根节点的右孩子（Right Child）的右孩子（Right Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

LR：LeftRight，也称“左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

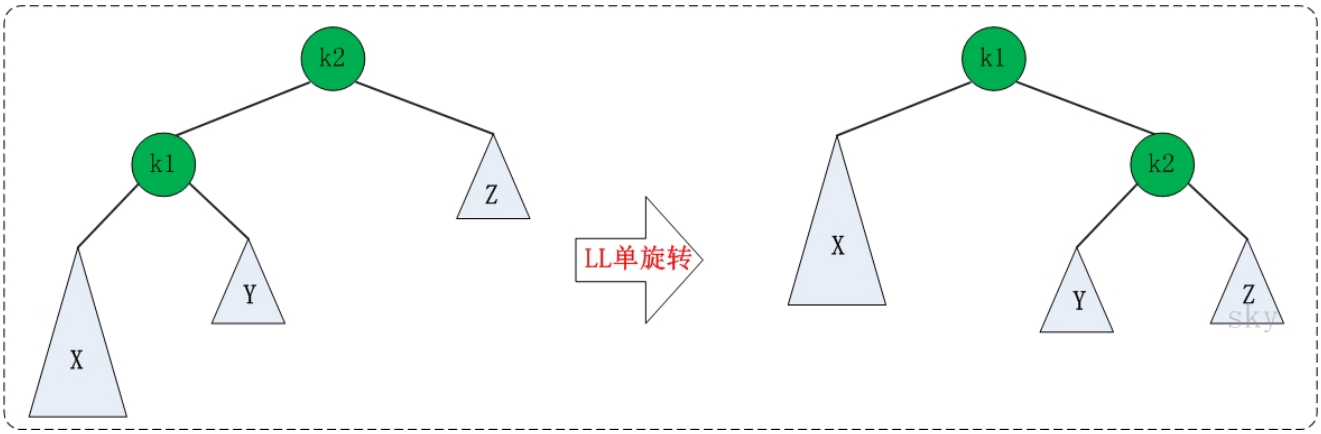
RL：RightLeft，也称“右左”。插入或删除一个节点后，根节点的右孩子（Right Child）的左孩子（Left Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

AVL树失去平衡之后，可以通过旋转使其恢复平衡。下面分别介绍四种失去平衡的情况下对应的旋转方法。

LL的旋转。LL失去平衡的情况下，可以通过一次旋转让AVL树恢复平衡。步骤如下：

1. 将根节点的左孩子作为新根节点。
2. 将新根节点的右孩子作为原根节点的左孩子。
3. 将原根节点作为新根节点的右孩子。

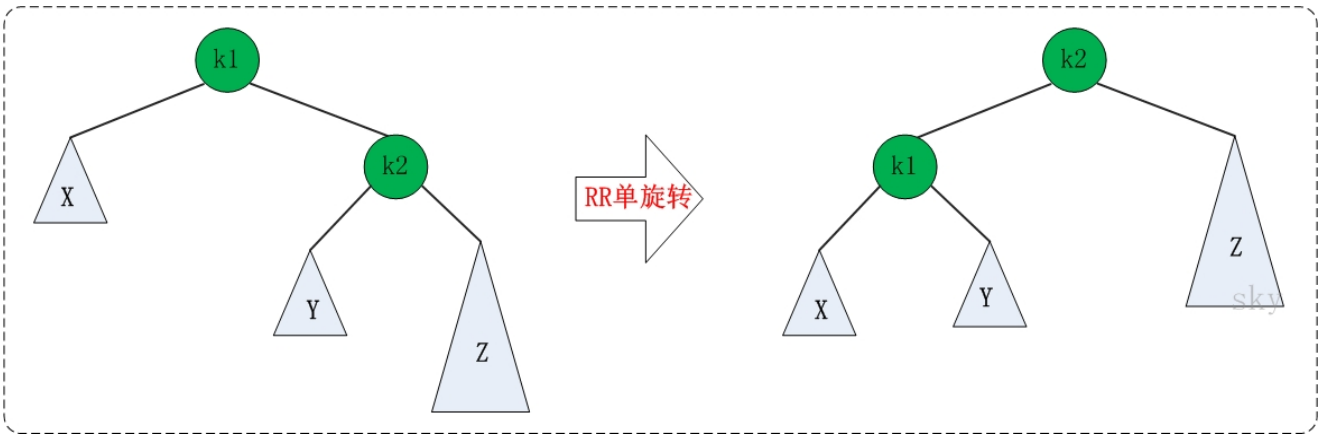
LL旋转示意图如下：



RR的旋转：RR失去平衡的情况下，旋转方法与LL旋转对称，步骤如下：

1. 将根节点的右孩子作为新根节点。
2. 将新根节点的左孩子作为原根节点的右孩子。
3. 将原根节点作为新根节点的左孩子。

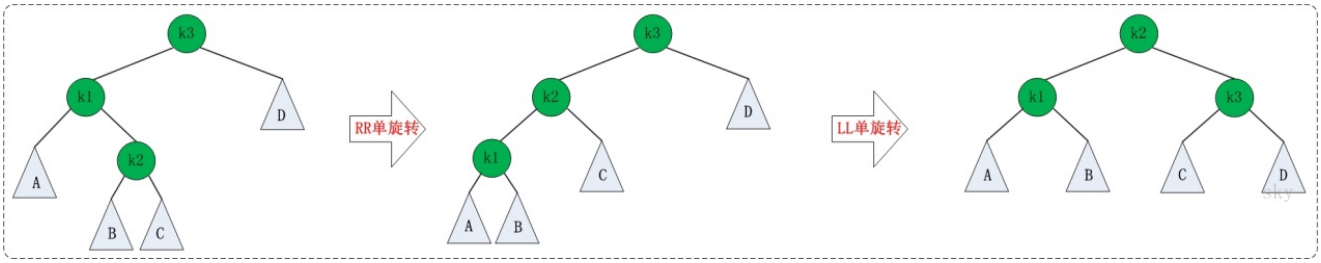
RR旋转示意图如下：



LR的旋转：LR失去平衡的情况下，需要进行两次旋转，步骤如下：

1. 围绕根节点的左孩子进行RR旋转。
2. 围绕根节点进行LL旋转。

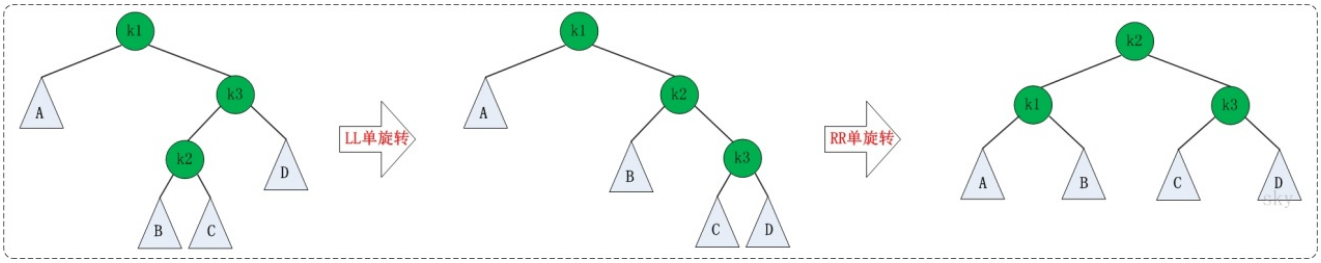
LR的旋转示意图如下：



RL的旋转：RL失去平衡的情况下也需要进行两次旋转，旋转方法与LR旋转对称，步骤如下：

1. 围绕根节点的右孩子进行LL旋转。
2. 围绕根节点进行RR旋转。

RL的旋转示意图如下：



3.2.4平衡多路查找树（B-Tree）

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

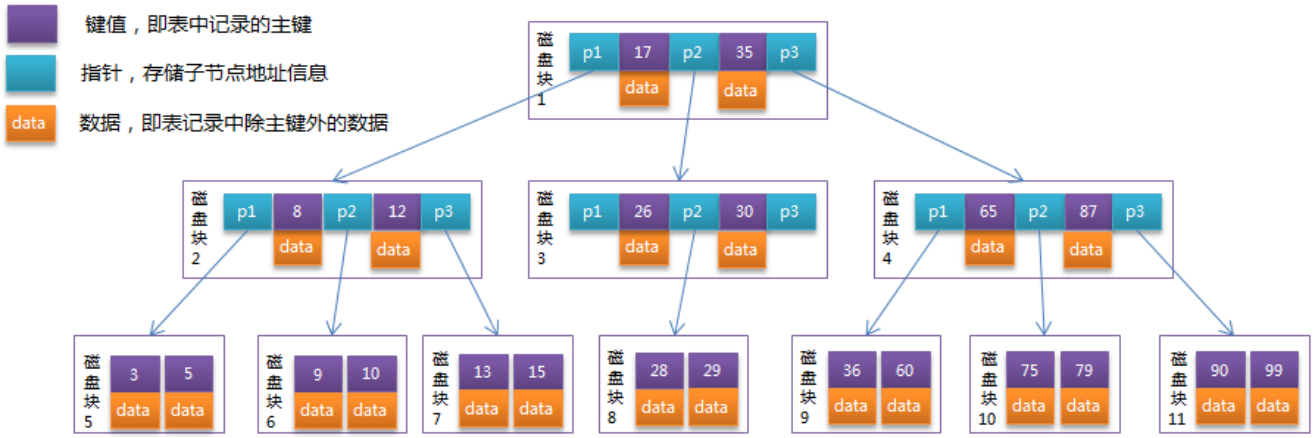
InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb_page_size将页的大小设置为4K、8K、16K.

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

一棵m阶的B-Tree有如下特性：

1. 每个节点最多有m个孩子。
2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。
3. 若根节点不是叶子节点，则至少有2个孩子
4. 所有叶子节点都在同一层，且不包含其它关键字信息
5. 每个非终端节点包含n个关键字信息（P0,P1,...Pn, k1,...kn）
6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1
7. ki(i=1,...n)为关键字，且关键字升序排序。
8. Pi(i=1,...n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：



每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

3.2.5B+Tree

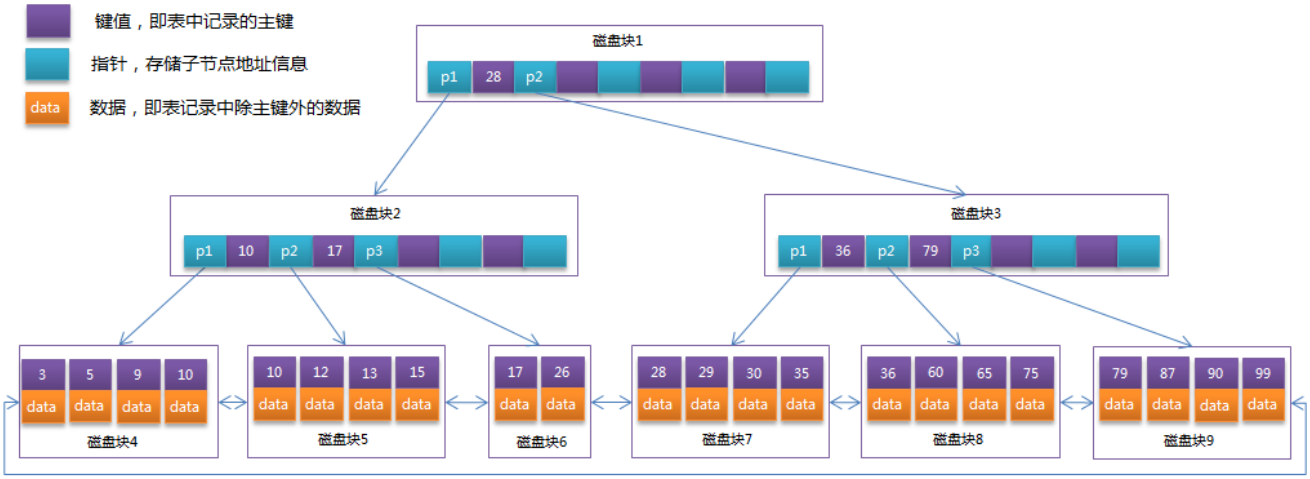
B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：



通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 * 10^3 * 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。mysql的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据

3.3索引的优缺点

- 优点:

  提高数据查询的效率,降低数据库的IO成本

  通过索引对数据进行排序,降低数据排序的成本,降低CPU的消耗

- 缺点:

  索引本身也是一张表,该表保存了主键与索引字段,并指向实体表的记录,所以索引列也要占用空间

  虽然索引大大提高了查询的速度,同时反向影响增删改操作的效率,因为表中数据变化之后,会导致索引内容不准,所以也需要更新索引表信息,增加了数据库的工作量

  随着业务的不断变化,之前建立的索引可能不能满足查询需求,需要消耗我们的时间去更新索引

3.4索引的类别

3.4.1普通索引

是最基本的索引，它没有任何限制.

```
CREATE index 索引名 on 表名(列名)
```

3.4.2唯一索引

与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。

```
CREATE UNIQUE index 索引名 on 表名(列名)
```

3.4.3主键索引

是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引.也就是说主键约束默认索引

3.4.4复合索引

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合

```
CREATE index 索引名 on 表名(列名,列名...)
```

3.4.5全文索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的where语句的参数匹配。fulltext索引配合match against操作使用，而不是一般的where语句加like。它可以在create table，alter table ，create index使用，不过目前只有char、varchar，text 列上可以创建全文索引。值得一提的是，在数据量较大时候，现将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多

```
CREATE TABLE `table` (
    `id` int(11) NOT NULL AUTO_INCREMENT ,
    `title` char(255) CHARACTER NOT NULL ,
    `content` text CHARACTER NULL ,
    `time` int(10) NULL DEFAULT NULL ,
    PRIMARY KEY (`id`),
    FULLTEXT (content)
);
```

3.5索引的基本语法

- 创建

```
ALTER mytable ADD  [UNIQUE]  INDEX [indexName] ON 表名(列名)
```

- 删除

```
DROP INDEX [indexName] ON 表名;
```

- 查看

```
SHOW INDEX FROM 表名
```

- alter命令

```
-- 有四种方式来添加数据表的索引：
ALTER TABLE tbl_name ADD PRIMARY KEY (column_list): 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。

ALTER TABLE tbl_name ADD UNIQUE index_name (column_list): 这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次）。

ALTER TABLE tbl_name ADD INDEX index_name (column_list): 添加普通索引，索引值可出现多次。

ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list):该语句指定了索引为 FULLTEXT ，用于全文索引。
```

3.6索引的使用场景

3.6.1适合使用索引

1. 频繁作为查询条件的字段应该创建索引
2. 多表查询中与其它表进行关联的字段,外键关系建立索引
3. 单列索引/复合索引的选择,高并发下倾向于创建复合索引
4. 查询中经常用来排序的字段
5. 查询中经常用来统计或者分组字段

3.6.2不适合使用索引

1. 频繁更新的字段: 每次更新都会影响索引树
2. where条件查询中用不到的字段
3. 表记录太少
4. 经常增删改的表: 更新了表,索引也得更新才行
5. 注意: 如果一张表中,重复的记录非常多,为它建立索引就没有太大意义

4.性能分析

4.1Query Optimizer

MySQL Optimizer是一个专门负责优化SELECT 语句的优化器模块，它主要的功能就是通过计算分析系统中收集的各种统计信息，为客户端请求的Query 给出他认为最优的执行计划，也就是他认为最优的数据检索方式。

4.2MySQL常见瓶颈

1. CPU饱和:CPU饱和的时候,一般发生在数据装入内存或从磁盘上读取数据的时候
2. IO瓶颈: 磁盘IO瓶颈发生在装入数据远大于内存容量的时候
3. 服务器硬件的性能瓶颈

4.3执行计划Explain

4.3.1Explain概述

使用explain关键字可以模拟优化器执行SQL查询语句,从而知道MYSQL是如何处理SQL语句的.我们可以用执行计划来分析查询语句或者表结构的性能瓶颈

4.3.2Explain作用

1. 查看表的读取顺序
2. 查看数据库读取操作的操作类型
3. 查看哪些索引有可能被用到
4. 查看哪些索引真正被用到
5. 查看表之间的引用
6. 查看表中有多少行记录被优化器查询

4.3.3语法

- 语法

```
explain  sql语句
```

- 示例

```
explain select * from tsmall;
```

4.3.4各字段解释

- 准备工作

```
create table t1(
  id int primary key,
  name varchar(20),
  col1 varchar(20),
  col2 varchar(20),
  col3 varchar(20)
);
create table t2(
  id int primary key,
  name varchar(20),
   col1 varchar(20),
   col2 varchar(20),
  col3 varchar(20)
);
create table t3(
  id int primary key,
  name varchar(20),
    col1 varchar(20),
   col2 varchar(20),
  col3 varchar(20)
);
insert into t1 values(1,'zs1','col1','col2','col3');
insert into t2 values(1,'zs2','col2','col2','col3');
insert into t3 values(1,'zs3','col3','col2','col3');

create index ind_t1_c1 on t1(col1);
create index ind_t2_c1 on t2(col1);
create index ind_t3_c1 on t3(col1);

create index ind_t1_c12 on t1(col1,col2);
create index ind_t2_c12 on t2(col1,col2);
create index ind_t3_c12 on t3(col1,col2);
```

执行explain sql语句后:

```
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
```

4.3.4.1 id

- select 查询的序列号,包含一组数字,表示查询中执行Select子句或操作表的顺序
- 三种情况:

1. id值相同,执行顺序由上而下

```
explain select t2.* from t1,t2,t3 where t1.id = t2.id and t1.id= t3.id and t1.name = 'zs';
```

```
mysql> explain select t2.*
    -> from t1,t2,t3
    -> where t1.id = t2.id and t1.id= t3.id and t1.name = 'zs';
+----+-------------+-------+--------+---------------+---------+---------+-----------+------+-------------+
| id | select_type | table | type   | possible_keys | key     | key_len | ref       | rows | Extra       |
+----+-------------+-------+--------+---------------+---------+---------+-----------+------+-------------+
|  1 | SIMPLE      | t1    | ALL    | PRIMARY       | NULL    | NULL    | NULL      |    1 | Using where |
|  1 | SIMPLE      | t2    | eq_ref | PRIMARY       | PRIMARY | 4       | test.t1.id |    1 |             |
|  1 | SIMPLE      | t3    | eq_ref | PRIMARY       | PRIMARY | 4       | test.t1.id |    1 | Using index |
+----+-------------+-------+--------+---------------+---------+---------+-----------+------+-------------+
```

2. id值不同,id值越大优先级越高,越先被执行

```
explain select t2.* from t2 where id = (select id from t1 where id = (select t3.id from t3
where t3.name='zs3'));
```

```
+----+-------------+-------+-------+---------------+---------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+------+------+-------------+
|  1 | PRIMARY     | t2    | const | PRIMARY       | PRIMARY | 4       | const|    1 |             |
|  2 | SUBQUERY    | t1    | const | PRIMARY       | PRIMARY | 4       |      |    1 | Using index |
|  3 | SUBQUERY    | t3    | ALL   | NULL          | NULL    | NULL    | NULL |    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+------+------+-------------+
```

3. id值有相同的也有不同的,如果id相同,从上往下执行,id值越大,优先级越高,越先执行

```
explain select t2.* from (select t3.id from t3 where t3.name='zs3') s1,t2 where s1.id =
t2.id;
```

```
+----+-------------+------------+--------+---------------+---------+---------+------+------+-------------+
| id | select_type | table      | type   | possible_keys | key     | key_len | ref  | rows | Extra       |
+----+-------------+------------+--------+---------------+---------+---------+------+------+-------------+
|  1 | PRIMARY     | <derived2> | system | NULL          | NULL    | NULL    | NULL |    1 |             |
|  1 | PRIMARY     | t2         | const  | PRIMARY       | PRIMARY | 4       | const|    1 |             |
|  2 | DERIVED     | t3         | ALL    | NULL          | NULL    | NULL    | NULL |    1 | Using where |
+----+-------------+------------+--------+---------------+---------+---------+------+------+-------------+
```

4.3.4.2select_type

查询类型,主要用于区别

- SIMPLE : 简单的select查询,查询中不包含子查询或者UNION
- PRIMARY: 查询中若包含复杂的子查询,最外层的查询则标记为PRIMARY
- SUBQUERY : 在SELECT或者WHERE列表中包含子查询
- DERIVED : 在from列表中包含子查询被标记为DRIVED衍生,MYSQL会递归执行这些子查询,把结果放到临时表中
- UNION: 若第二个SELECT出现在union之后,则被标记为UNION, 若union包含在from子句的子查询中,外层select被标记为:derived
- UNION RESULT: 从union表获取结果的select

```
explain select col1,col2 from t1 union select col1,col2 from t2;
```

```
+------+--------------+------------+-------+---------------+-----------+---------+------+------+-------------+
| id   | select_type  | table      | type  | possible_keys | key       | key_len | ref  | rows | Extra       |
+------+--------------+------------+-------+---------------+-----------+---------+------+------+-------------+
|    1 | PRIMARY      | t1         | index | NULL          | ind_t1_c12| 126     | NULL |    1 | Using index |
|    2 | UNION        | t2         | index | NULL          | ind_t2_c12| 126     | NULL |    1 | Using index |
| NULL | UNION RESULT | <union1,2> | ALL   | NULL          | NULL      | NULL    | NULL | NULL |             |
+------+--------------+------------+-------+---------------+-----------+---------+------+------+-------------+
```

4.3.4.3table

显示这一行的数据是和哪张表相关

4.3.4.4type

访问类型: all, index,range,ref,eq_ref, const,system,null

最好到最差依次是: system > const > eq_ref>ref >range > index > all , 最好能优化到range级别或则ref级别

- system: 表中只有一行记录(系统表), 这是const类型的特例, 基本上不会出现
- const: 通过索引一次查询就找到了,const用于比较primary key或者unique索引,因为只匹配一行数据,所以很快,如将主键置于where列表中,mysql就会将该查询转换为一个常量

```
explain select * from (select * from t1 where id=1) s1;
```

- eq_ref: 唯一性索引扫描, 对于每个索引键,表中只有一条记录与之匹配, 常见于主键或者唯一索引扫描

```
explain select * from t1,t2 where t1.id = t2.id;
```

- ref : 非唯一性索引扫描,返回匹配某个单独值的所有行,本质上也是一种索引访问,它返回所有符合条件的行,然而它可能返回多个符合条件的行

```
explain select * from t1 where col1='zs1';
```

- range : 只检索给定范围的行, 使用一个索引来选择行.key列显示的是真正使用了哪个索引,一般就是在where条件中使用between,>,<,in 等范围的条件,这种在索引范围内的扫描比全表扫描要好,因为它只在某个范围中扫描,不需要扫描全部的索引

```
explain select * from t1 where id between 1 and 10;
```

- index : 扫描整个索引表, index 和all的区别为index类型只遍历索引树. 这通常比all快,因为索引文件通常比数据文件小,虽然index和all都是读全表,但是index是从索引中读取,而all是从硬盘中读取数据

```
explain select id from t1;
```

- all : full table scan全表扫描 ,将遍历全表以找到匹配的行

```
explain select * from t1;
```

- 注意: 开发中,我们得保证查询至少达到range级别,最好能达到ref.

  如果百万条数据出现all, 一般情况下就需要考虑使用索引优化了

4.3.4.5possible_keys

SQL查询中可能用到的索引,但查询的过程中不一定真正使用

4.3.4.6key

查询过程中真正使用的索引,如果为null,则表示没有使用索引

查询中使用了覆盖索引,则该索引仅出现在key列表中

```
explain select t2.* from t1,t2,t3 where t1.col1 = ' ' and t1.id = t2.id and t1.id= t3.id;
```

```
mysql> explain select t2.* from t1,t2,t3 where t1.col1 = ' ' and t1.id = t2.id and t1.id= t3.id;
+----+-------------+-------+--------+---------------------------+----------+---------+-------------+------+--------------------------+
| id | select_type | table | type   | possible_keys             | key      | key_len | ref         | rows | Extra                    |
+----+-------------+-------+--------+---------------------------+----------+---------+-------------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref    | PRIMARY,ind_t1_c1,ind_t1_c12 | ind_t1_c1 | 63      | const       |    1 | Using where; Using index |
|  1 | SIMPLE      | t2    | eq_ref | PRIMARY                   | PRIMARY  | 4       | heima.t1.id |    1 |                          |
|  1 | SIMPLE      | t3    | eq_ref | PRIMARY                   | PRIMARY  | 4       | heima.t1.id |    1 | Using index              |
+----+-------------+-------+--------+---------------------------+----------+---------+-------------+------+--------------------------+
```

```
explain select col1 from t1;
```

```
mysql> explain select col1 from t1;
+----+-------------+-------+-------+---------------+----------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys | key      | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+----------+---------+------+------+-------------+
|  1 | SIMPLE      | t1    | index | NULL          | ind_t1_c1 | 63      | NULL |    1 | Using index |
+----+-------------+-------+-------+---------------+----------+---------+------+------+-------------+
```

4.3.4.7key_len

索引中使用的字节数,可通过该列计算查询中使用的索引的长度,在不损失精确度的情况下,长度越短越好, key_len显示的值为索引字段的最大可能长度,并非实际使用长度, 即key_len是根据表定义计算而得

```
explain select * from t1 where col1='c1';
```

```
mysql> explain select * from t1 where col1='c1';
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys | key      | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------------+
|  1 | SIMPLE      | t1    | ref  | ind_t1_c1,ind_t1_c12 | ind_t1_c1 | 63      | const |    1 | Using where |
+----+-------------+-------+------+---------------+----------+---------+-------+------+-------------+
```

```
explain select * from t1 where col1='col1' and col2 = 'col2';

-- 注意: 为了演示这个结果,我们删除了c1上面的索引
alter table t1 drop index ind_t1_c1;
-- 执行完成之后,再次创建索引
create index ind_t1_c1 on t1(col1);
```

```
ysql> explain select * from t1 where col1='col1' and col2='col2';
----+-------------+-------+------+---------------+------------+---------+-------------+------+-------------+
 id | select_type | table | type | possible_keys | key        | key_len | ref         | rows | Extra       |
----+-------------+-------+------+---------------+------------+---------+-------------+------+-------------+
  1 | SIMPLE      | t1    | ref  | ind_t1_c12    | ind_t1_c12 | 126     | const,const |    1 | Using where |
----+-------------+-------+------+---------------+------------+---------+-------------+------+-------------+
```

4.3.4.8ref

显示索引的哪一列被使用了,如果可能的话,是一个常数.哪些列或者常量被用于查找索引列上的值

```
explain select * from t1,t2 where t1.col1 = t2.col1 and t1.col2 = 'col2';
```

```
mysql> explain select * from t1,t2 where t1.col1 = t2.col1 and t1.col2 = 'col2';
+----+-------------+-------+------+-----------------------+-----------+---------+---------------+------+-------------+
| id | select_type | table | type | possible_keys         | key       | key_len | ref           | rows | Extra       |
+----+-------------+-------+------+-----------------------+-----------+---------+---------------+------+-------------+
|  1 | SIMPLE      | t1    | ALL  | ind_t1_c12            | NULL      | NULL    | NULL          |    1 | Using where |
|  1 | SIMPLE      | t2    | ref  | ind_t2_c1,ind_t2_c12  | ind_t2_c1 | 63      | heima.t1.col1 |    1 | Using where |
+----+-------------+-------+------+-----------------------+-----------+---------+---------------+------+-------------+
```

4.3.4.9rows

根据表统计信息及索引选用的情况,估算找出所需记录要读取的行数 (有多少行记录被优化器读取) ,越少越好

4.3.4.10extra

包含其它一些非常重要的额外信息

- Using filesort : 说明mysql会对数据使用一个外部的索引排序,而不是按照表内的索引顺序进行读取,Mysql中无法利用索引完成的排序操作称为文件排序

```
explain select col1 from t1 where col1='col1' order by col3;
```

```
mysql> explain select col1 from t1 where col1='col1' order by col3;
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+-----------------------------------------------------+
| id | select_type | table | type | possible_keys                       | key       | key_len | ref   | rows | Extra                                               |
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+-----------------------------------------------------+
|  1 | SIMPLE      | t1    | ref  | ind_t1_c123,ind_t1_c12,ind_t1_c1    | ind_t1_c123 | 63      | const |    7 | Using where; Using index; Using filesort |
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+-----------------------------------------------------+
```

```
-- 上面这条SQL语句出现了using filesort,但是我们去执行下面这条SQL语句的时候它,又不会出现using filesort
 explain select col1 from t1 where col1='col1' order by col2;
```

```
mysql> explain select col1 from t1 where col1='col1' order by col2;
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys                       | key       | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref  | ind_t1_c12,ind_t1_c1,ind_t1_c123    | ind_t1_c12 | 63      | const |    1 | Using where; Using index |
+----+-------------+-------+------+-------------------------------------+-----------+---------+-------+------+--------------------------+
```

```
-- 如何优化第一条SQL语句 ?
 create index ind_t1_c13 on t1(col1,col3);
 explain select col1 from t1 where col1='col1' order by col3;
```

```
mysql> explain select * from t1 where col1='col1' order by col3;
+----+-------------+-------+------+-------------------------------------------------+-----------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys                                   | key       | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+-------------------------------------------------+-----------+---------+-------+------+-------------+
|  1 | SIMPLE      | t1    | ref  | ind_t1_c12,ind_t1_c1,ind_t1_c123,ind_t1_c13     | ind_t1_c13 | 63      | const |    1 | Using where |
+----+-------------+-------+------+-------------------------------------------------+-----------+---------+-------+------+-------------+
```

- Using temporary : 使用了临时表保存中间结果,Mysql在对查询结果排序时使用了临时表,常见于order by 和分组查询group by

```
explain select col1 from t1 where col1>'col1' group by col2;
```

```
mysql> explain select col1 from t1 where col1>'col1' group by col2;
+----+-------------+-------+-------+-------------------------+-----------+---------+------+------+-----------------------------------------------------------------+
| id | select_type | table | type  | possible_keys           | key       | key_len | ref  | rows | Extra                                                           |
+----+-------------+-------+-------+-------------------------+-----------+---------+------+------+-----------------------------------------------------------------+
|  1 | SIMPLE      | t1    | index | ind_t1_c12,ind_t1_c1    | ind_t1_c12 | 126     | NULL |    1 | Using where; Using index; Using temporary; Using filesort |
+----+-------------+-------+-------+-------------------------+-----------+---------+------+------+-----------------------------------------------------------------+
```

```
explain select col1 from t1 where col1 >'col1' group by col1,col2;
```

```
mysql> explain select col1 from t1 where col1 >'col1' group by col1,col2;
+----+-------------+-------+-------+-------------------------------+-----------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys                 | key       | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+-------------------------------+-----------+---------+------+------+--------------------------+
|  1 | SIMPLE      | t1    | index | ind_t1_c12,ind_t1_c1,ind_t1_c123 | ind_t1_c12 | 126     | NULL |    1 | Using where; Using index |
+----+-------------+-------+-------+-------------------------------+-----------+---------+------+------+--------------------------+
```

- Using index :
- 查询操作中使用了覆盖索引(查询的列和索引列一致),避免访问了表的数据行,效率好
- 如果同时出现了using where, 表明索引被用来执行索引键值的查找
- 如果没有同时出现using where, 表明索引用来读取数据而非执行查找动作
- 覆盖索引: 查询的列和索引列一致, 换句话说查询的列要被所键的索引覆盖,就是select中数据列只需从索引中就能读取,不必读取原来的数据行,MySql可以利用索引返回select列表中的字段,而不必根据索引再次读取数据文件

```
explain select col2 from t1 where col1='col1';
```

```
mysql> explain select col2 from t1 where col1='col1';
+----+-------------+-------+------+-----------------------+-----------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys         | key       | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+-----------------------+-----------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | t1    | ref  | ind_t1_c123,ind_t1_c12 | ind_t1_c12 | 63      | const |    7 | Using where; Using index |
+----+-------------+-------+------+-----------------------+-----------+---------+-------+------+--------------------------+
```

```
explain select col2 from t1;
```

```
mysql> explain select col2 from t1;
+----+-------------+-------+-------+---------------+------------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys | key        | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+------------+---------+------+------+-------------+
|  1 | SIMPLE      | t1    | index | NULL          | ind_t1_c12 | 126     | NULL |    7 | Using index |
+----+-------------+-------+-------+---------------+------------+---------+------+------+-------------+
```

- using where : 表明使用了where条件过滤
- using join buffer : 表明使用了连接缓存, join次数太多了可能会出现
- impossible where : where子句中的值总是false,不能用来获取任何数据

```
explain select * from t1 where col1='zs' and col1='ls';
```

- select tables optimized away :
- 在没有group by 子句的情况下, 基于索引优化min/max操作或者对于MyISAM存储引擎优化count(*)操作,不必等到执行阶段再进行计算,查询执行计划生成阶段即完成优化
- distinct : 优化distinct操作,在找到第一个匹配的数据后即停止查找同样的值的动作

4.3.5小练习

```
explain select a1.name,(select id from t3) a2
from
        (select id,name from t1 where name='zs1') a1
union
        select name,id from t2;
```

```
mysql> explain select a1.name,(select id from t3) a2
    -> from
    -> (select id,name from t1 where name='zs1') a1
    -> union
    -> select name,id from t2;
+----+--------------+------------+--------+---------------+----------+---------+------+------+-------------+
| id | select_type  | table      | type   | possible_keys | key      | key_len | ref  | rows | Extra       |
+----+--------------+------------+--------+---------------+----------+---------+------+------+-------------+
|  1 | PRIMARY      | <derived3> | system | NULL          | NULL     | NULL    | NULL |    1 |             |
|  3 | DERIVED      | t1         | ALL    | NULL          | NULL     | NULL    | NULL |    1 | Using where |
|  2 | SUBQUERY     | t3         | index  | NULL          | ind_t3_c1 | 63     | NULL |    1 | Using index |
|  4 | UNION        | t2         | ALL    | NULL          | NULL     | NULL    | NULL |    1 |             |
| NULL | UNION RESULT | <union1,4> | ALL  | NULL          | NULL     | NULL    | NULL | NULL |             |
+----+--------------+------------+--------+---------------+----------+---------+------+------+-------------+
```

执行顺序如下:

id=4, select_type为union, union后的select语句先被执行

id=3, 因为(select id,name from t1 where name='zs1')被当作一张表处理,所以为select_type 为derived

id=2, select_type为SUBQUERY,为select后面的子查询

id=1, 表示union中的第一个select, select_type为primary表示该查询为外层查询,table被标记为<derived3>,表示结果来自衍生表

id=null,代表从union的临时表中读取行记录, <union1,4>表示将id=1和id=4的结果进行union操作

5.优化实战

5.1单表查询优化

需求: 查询 category_id 为1 且 comments 大于 1 的情况下,views 最多的 article_id。

- 建表语句

```
CREATE TABLE IF NOT EXISTS `article` (
`id` INT(10) UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
`author_id` INT(10) UNSIGNED NOT NULL,
`category_id` INT(10) UNSIGNED NOT NULL,
`views` INT(10) UNSIGNED NOT NULL,
`comments` INT(10) UNSIGNED NOT NULL,
`title` VARBINARY(255) NOT NULL,
`content` TEXT NOT NULL
);

INSERT INTO `article`(`author_id`, `category_id`, `views`, `comments`, `title`, `content`) VALUES
(1, 1, 1, 1, '1', '1'),
(2, 2, 2, 2, '2', '2'),
(1, 1, 3, 3, '3', '3');
```

- 执行计划

```
EXPLAIN SELECT id,author_id FROM article WHERE category_id = 1 AND comments > 1 ORDER BY views
DESC LIMIT 1;
```

- 没有索引情况

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	article	ALL	(Null)	(Null)	(Null)	(Null)	3	Using where; Using filesort

结论：很显然,type 是 ALL,即最坏的情况。Extra 里还出现了 Using filesort,也是最坏的情况。优化是必须的。

- 优化一:

```
create index idx_article_ccv on article(category_id,comments,views);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	article	range	idx_article_ccv	idx_artic	8	(Null)	1	Using where; Using filesort

结论: type 变成了 range,这是可以忍受的。但是 extra 里使用 Using filesort 仍是无法接受的。但是我们已经建立了索引,为啥没用呢? 这是因为按照 BTree 索引的工作原理, 先排序 category_id, 如果遇到相同的 category_id 则再排序 comments,如果遇到相同的 comments 则再排序 views。当 comments 字段在联合索引里处于中间位置时,因 comments > 1 条件是一个范围值(所谓 range),MySQL 无法利用索引再对后面的 views 部分进行检索,即 range 类型查询字段后面的索引无效。

- 优化二:

```
-- 先删除优化一索引
DROP INDEX idx_article_ccv ON article;
-- 创建索引
create index idx_article_cv on article(category_id,views);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	article	ref	idx_article_cv	idx_article_cv	4	const	2	Using where

5.2关联查询优化

5.2.1示例

需求: 使用左外连接查询class和book

- 建表语句

```
CREATE TABLE IF NOT EXISTS `class` (
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`card` INT(10) UNSIGNED NOT NULL,
PRIMARY KEY (`id`)
);
CREATE TABLE IF NOT EXISTS `book` (
`bookid` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`card` INT(10) UNSIGNED NOT NULL,
PRIMARY KEY (`bookid`)
);

INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO class(card) VALUES(FLOOR(1 + (RAND() * 20)));

INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
INSERT INTO book(card) VALUES(FLOOR(1 + (RAND() * 20)));
```

- 执行计划

```
EXPLAIN SELECT * FROM class LEFT JOIN book ON class.card = book.card;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	class	ALL	(Null)	(Null)	(Null)	(Null)	20	
1	SIMPLE	book	ALL	(Null)	(Null)	(Null)	(Null)	20	

- 优化

```
ALTER TABLE `book` ADD INDEX Y ( `card`);
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	class	ALL	(Null)	(Null)	(Null)	(Null)	20	
1	SIMPLE	book	ref	Y	Y	4	mysql_c	1	Using index

可以看到第二行的 type 变为了 ref,rows 也变成了优化比较明显。
这是由左连接特性决定的。LEFT JOIN 条件用于确定如何从右表搜索行,左边一定都有,所以右边是我们的关键点,一定需要建立索引。

也就是说: 外连接在相反方创建索引

5.2.2优化建议

- 保证被驱动表的join字段已经被索引 (被驱动表 join 后的表为被驱动表 (需要被查询))
- left join 时，选择小表作为驱动表，大表作为被驱动表。
- inner join 时，mysql会自己帮你把小结果集的表选为驱动表。

5.3子查询优化

- 子查询尽量不要放在被驱动表，有可能使用不到索引。

```
select a.name ,bc.name from t_emp a left join
         (select b.id , c.name from t_dept b
         inner join t_emp c on b.ceo = c.id)bc
         on bc.id = a.deptid

上段查询中用到了子查询，必然 bc 表没有索引。肯定会进行全表扫描
上段查询 可以直接使用 两个 left join 优化

select a.name , c.name from t_emp a
    left outer join t_dept b on a.deptid = b.id
    left outer join t_emp c on b.ceo=c.id
所有条件都可以使用到索引

若必须用到子查询，可将子查询设置为驱动表，因为驱动表的type 肯定是 all，而子查询返回的结果表没有索引，必定也是all
```

5.4order by优化

5.4.1环境准备

- 建表语句

```
CREATE TABLE tblA(
  id int primary key not null auto_increment,
  age INT,
  birth TIMESTAMP NOT NULL,
  name varchar(200)
);

INSERT INTO tblA(age,birth,name) VALUES(22,NOW(),'abc');
INSERT INTO tblA(age,birth,name) VALUES(23,NOW(),'bcd');
INSERT INTO tblA(age,birth,name) VALUES(24,NOW(),'def');

CREATE INDEX idx_A_ageBirth ON tblA(age,birth,name);
```

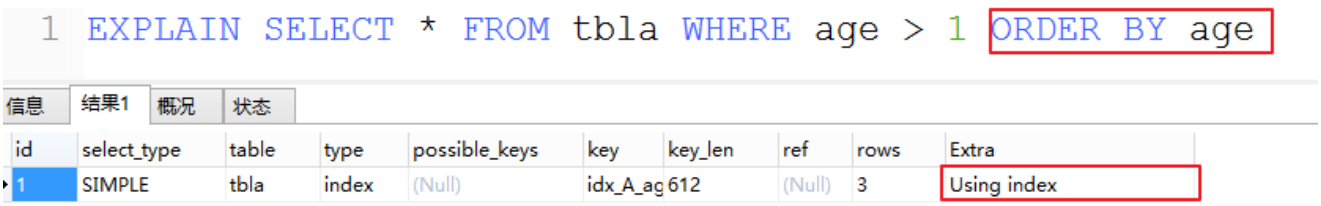
5.4.2说明

MySQL支持二种方式的排序，FileSort和Index，Index效率高. 它指MySQL扫描索引本身完成排序。FileSort方式效率较低。

5.4.3ORDER BY会使用Index方式排序

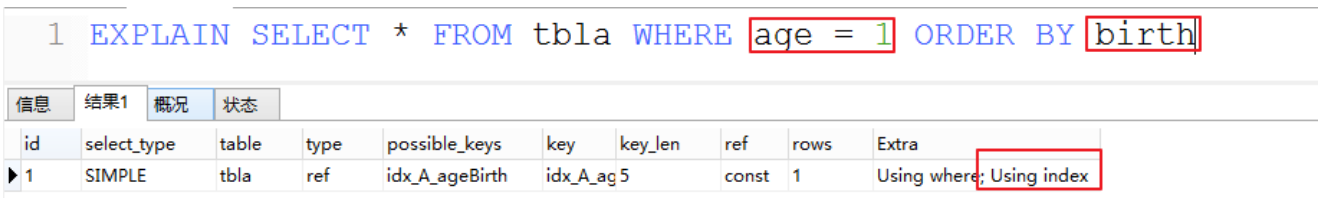
- ORDER BY 语句使用索引最左前列

```
EXPLAIN SELECT * FROM tbla WHERE age > 1 ORDER BY age
```



- 使用Where子句与Order BY子句条件列组合满足索引最左前列

```
EXPLAIN SELECT * FROM tbla WHERE age = 1 ORDER BY birth
```



- where子句中如果出现索引的范围查询(即explain中出现range)会导致order by 索引失效

```
1 EXPLAIN SELECT * FROM tbla WHERE age > 1 ORDER BY birth
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tbla	index	idx_A_ageBirth	idx_A_ag	612	(Null)	3	Using where; Using index; Using filesort

- Order by 复合条件排序, 按照索引顺序, 前后的排序类别不一致,会导致order by 索引失效

```
1 EXPLAIN SELECT * FROM tbla ORDER BY age DESC, birth DESC
```

信息 | 结果1 | 概况 | 状态

不会失效

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tbla	index	(Null)	idx_A_ag	612	(Null)	3	Using index

```
1 EXPLAIN SELECT * FROM tbla ORDER BY age DESC, birth ASC
```

信息 | 结果1 | 概况 | 状态

失效

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tbla	index	(Null)	idx_A_ag	612	(Null)	3	Using index; Using filesort

5.4.4练习

```
mysql> EXPLAIN SELECT * FROM tblA WHERE age > 20 ORDER BY age;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
|  1 | SIMPLE      | tblA  | index | idx_A_ageBirth | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM tblA WHERE age > 20 ORDER BY age,birth;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
|  1 | SIMPLE      | tblA  | index | idx_A_ageBirth | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM tblA WHERE age > 20 ORDER BY birth;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                                        |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
|  1 | SIMPLE      | tblA  | index | idx_A_ageBirth | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index; Using filesort |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM tblA WHERE age > 20 ORDER BY birth,age;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                                        |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
|  1 | SIMPLE      | tblA  | index | idx_A_ageBirth | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index; Using filesort |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM tblA ORDER BY birth;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                                        |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
|  1 | SIMPLE      | tblA  | index | NULL          | idx_A_ageBirth | 9       | NULL |    3 | Using index; Using filesort |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM tblA WHERE birth > '2016-01-28 00:00:00' ORDER BY birth;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                                        |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
|  1 | SIMPLE      | tblA  | index | NULL          | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index; Using filesort |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------------------------+
1 row in set (0.01 sec)

mysql> EXPLAIN SELECT * FROM tblA WHERE birth > '2016-01-28 00:00:00' ORDER BY age;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
|  1 | SIMPLE      | tblA  | index | NULL          | idx_A_ageBirth | 9       | NULL |    3 | Using where; Using index |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+--------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * FROM tblA ORDER BY age ASC, birth DESC;
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------+
| id | select_type | table | type  | possible_keys | key           | key_len | ref  | rows | Extra                      |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------+
|  1 | SIMPLE      | tblA  | index | NULL          | idx_A_ageBirth | 9       | NULL |    3 | Using index; Using filesort |
+----+-------------+-------+-------+---------------+---------------+---------+------+------+----------------------------+
1 row in set (0.00 sec)
```

5.5GROUP BY优化

5.5.1说明

group by实质是先排序后进行分组，遵照索引建的最佳左前缀

5.5.2注意

where高于having，能写在where限定的条件就不要去having限定了

5.6limit优化

5.6.1说明

limit常用于分页处理,时常会伴随order by 从句使用, 因此大多时候会使用Filesorts,这样会造成大量的IO问题

5.6.2优化

- 步骤一: 使用有索引的列或者主键进行Order By操作
- 步骤二: 记录上次返回的主键,在下次查询的时候使用注解过滤(如果主键不是连续的,是字符串类型,可以创建一个列记录)

6.总结

6.1总结

- 全值匹配我最爱
- 最佳左前缀法则(如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列)
- 不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描
- 存储引擎不能使用索引中范围条件右边的列
- 尽量使用覆盖索引(只访问索引的查询(索引列和查询列一致))，减少select *
- mysql 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描
- is not null 也无法使用索引,但是is null是可以使用索引的
- like以通配符开头('%abc...')mysql索引失效会变成全表扫描的操作
- 字符串不加单引号索引失效
- 少用or,用它来连接时会索引失效

6.2口诀

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