B/+树

B树

- 排序树
- 每个叶子节点的深度相同
- 每个内部节点有指向子节点的指针
- 内部节点的子节点数很大，所以深度很低

B+树

b+树包含b树特点，并且具有下面的特点：

- 所有叶子节点形成有序链表
- 内部节点仅有索引作用，数据均存放在叶子节点
- 有k个子节点必然有k个关键码

索引

分类

b+树索引

> （mysql技术内幕）

按实现分类

- innodb下的b+树索引方式 （**5.41节**）
  - 聚簇索引
    - 定义
      - 聚簇索引是按表的主键构造一颗B+树，叶子节点就保存了表的行记录数据
      - 每个数据页通过一个双向链表连接
      - 实际的数据页只能按照一个B+树进行排序，所以一张表只能拥有一个聚簇索引
    - 聚簇索引在物理上是无序的，假如保证物理有序成本太高（磁盘io），逻辑有序通过叶节点的双向链表可以实现
  - 辅助索引
    - 辅助索引和聚簇索引结构相同，按索引列构建了一颗新的b+树，
    - 区别是辅助索引的叶子节点不包含行记录的全部数据，而是包含了主键的值（**不知道是否还有其他信息**），查到主键值后，再使用主键查聚簇索引
  - innodb下b+树索引的分裂：（**5.43节**）（那么其他引擎呢MyIsam？）
    - 当插入新节点时，需要叶节点分裂；而B+树的定义方式为：将内部节点从中间分裂
    - 假如数据库的聚簇索引为自增的，1-10的叶子节点满了，插入11时，需要从5处分裂，之后插入11以及更大的主键时，1-5那块叶子节点就浪费了一半的空间（因为不可能再有6-10之间的主键插入了），所以mysql会在插入时判断主键的增长方向，决定分裂节点的位置，如果主键插入是随机的，那么就会选择中间记录

    - **非聚簇索引的索引（MyIsam）分裂方法?** 没找到
        - 非聚簇索引分裂的成本较低（非聚簇索引只保存行记录的地址，io较少）（也许没有做优化？）
- myIsam下的索引方式
    - 非聚簇索引
        - 非聚簇索引也是b+树索引，和聚簇索引的不同在于叶子节点仅保存了真正数据行的地址，需要按地址获得指定数据行（需要一次磁盘io）
- 聚簇索引的优点：
    - 使用主键查找时，可以直接找到行记录，而非聚簇索引还需要通过地址找记录行
    - 可以使用覆盖索引优化（主要是辅助索引可以这样）
    - 因为磁盘io一次读取一个节点（一页）的数据，而一页数据一般不止一行数据，可以实现数据的预读
- 聚簇索引缺点：
    -
        - 聚簇数据最大限度的提高了I/O密集型应用的性能，但如果数据全部都放在内存中，则访问的顺序就没有那么重要了，聚簇索引也就没有那么优势了；
        - 插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到InnoDB表中速度最快的方式。但如果不是按照主键顺序加载数据，那么在加载完成后最好使用OPTIMIZE TABLE命令重新组织一下表。
        - 更新聚簇索引列的代价很高，因为会强制InnoDB将每个被更新的行移动到新的位置。
        - 基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临"页分裂"的问题。当行的主键值要求必须将这一行插入到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳该行，这就是一次分裂操作。页分裂会导致表占用更多的磁盘空间。
        - 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。
        - 二级索引（非聚簇索引）可能比想象的要更大，因为在二级索引的叶子节点包含了引用行的主键列。
        - 二级索引访问需要两次索引查找，而不是一次。
    - 页分裂成本高
    - 主键的选择有问题，以至不停出现页分裂，频率高于非聚簇索引
    - 更新聚簇索引列成本较高（改变主键值），因为需要保证物理有序（**是吗？假如以链表保存，不需要物理有序**）
        - **聚簇索引是否物理有序？无序那么就无法预读，有序维护成本太高**

          许多数据库的文档会这样告诉读者：聚集索引按照顺序物理地存储数据。如果看图 5-14，可能也会有这样的感觉。但是试想一下，如果聚集索引必须按照特定顺序存放物理记录，则维护成本显得非常之高。所以，聚集索引的存储并不是物理上连续的，而是逻辑上连续的。这其中有两点：一是前面说过的页通过双向链表链接，页按照主键的顺序排序；另一点是每个页中的记录也是通过双向链表进行维护的，物理存储上可以同样不按照主键存储。

          书上(mysql技术内幕P194无序)、有些博客讲有序
        - 辅助索引可能较大，叶子节点需要主键值
        - 二级索引需要两次b+树索引搜索

按使用分类

- 普通索引
  - innodb下的普通索引：即辅助索引
  - MyISam下的普通索引和主键索引结构相同：一颗b+树，叶节点的数据存放着行记录的地址
  - 非唯一索引实现

1 “溢出页”方法

为了简化问题，B^+树算法的研究一般都假设不存在键值重复的情况[4]。现有处理重复键值的方法主要采用“溢出页”[5]：当某个数据键值对应的记录数大于 1 时，分配一个“溢出页”用来存放所有的重复键值及其对应记录的偏移量，如图 1 所示。

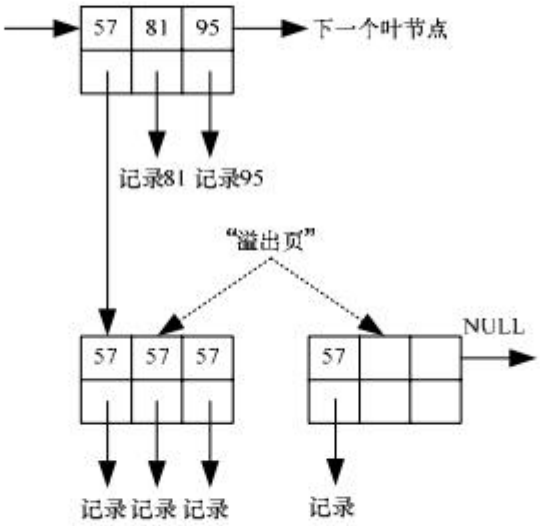
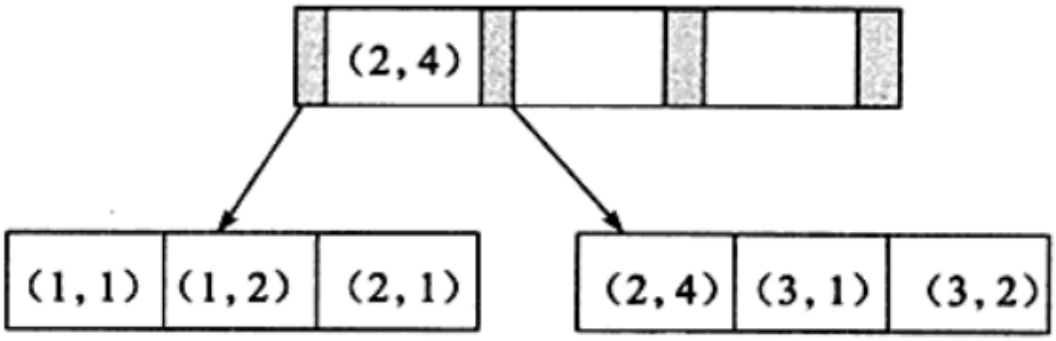


图 1 “溢出页”方法

- 唯一索引
  - 基本与普通索引一样
  - 只不过 索引不能重复，允许空值
- 主键索引
  - 唯一且不空
- 组合索引
  - 多字段的组合，作为索引
  - 索引内部结构：



    - 优点：
        - 联合排序的各字段，已排序，但需要从前缀匹配
        - 可以使用覆盖索引：比如联合索引(a, b)，当查询，select b where a = xxx;时。
    - 缺点
        - 内部节点须保存组合索引，相比普通索引，内部节点需要的空间更大
        - 维护成本更高
- 前缀索引
    - select * from users where username like 'xxx%' 可以使用辅助索引
    - 复合索引 查询条件为前若干字段可以使用索引
- **覆盖索引**
    - myisam下有覆盖索引吗？没有意义，因为数据域存的是地址
    - innodb下通过辅助索引取到主键值，需要再找一遍聚簇索引
    - 而覆盖索引意味着只需辅助索引就可以查到所需的结果：
        - 假如username上有索引，那么count(username)只需username所在的辅助索引就可以得到答案，虽然数据个数一样，但辅助索引的数据大小小很多，需要的磁盘io也少
        - 复合索引name_role，select role from users where username = xxx;这样就可以仅使用name_role索引获取结果，搜索的范围缩小，且不需要重新搜索聚簇索引

优化器选择不使用索引

> 不使用索引，指不使用辅助索引

- 比如：select * from users where role=xxx; 尽管有Index(role, rolename)，但是索引并不能覆盖所有的查询列，仍然需要查到主键后再走聚簇索引，而这样查到的主键很可能不是连续的（符合条件的主键有可能不连续），这样无法预读；（**主键读出来的顺序可能是无序的，但排序之后不就可以预读了吗？ 查出来的主键会排序吗**）（也许因为在**数据量大时就算排序**，也相当于遍历全部聚簇索引，所以还不如直接遍历，而**数量少时，可能排序了也不一定连续**，还是不能预读，所以直接主键查找）（竟然真有排序。。下两节**MRR**）
- 假如需要访问的数据量很小，优化器还是会选择使用辅助索引，因为可以直接用主键查到行，且不需要多次磁盘io
- 否则，会选择主键索引遍历

索引提示

> 显示地告诉优化器使用哪个索引

两种使用情况：

- 优化器使用了错误的索引，这种可能性很低很低

- 某个sql语句可用索引非常多，这是优化器可能需要选择使用哪个索引（执行路径），需要计算每个执行路径的成本，而计算成本的时间可能很长；所以可以强制优化器使用某个索引。
- 关键字 select * from xx **use/force index**;

Multi-Range Read

> 目的是为了减少磁盘随机访问时间，将随机访问转化为较为顺序的数据访问，对应io瓶颈的sql查询带来性能提高；适用于range，**ref/非唯一索引，eq_ref/唯一索引（这俩是啥啊https://www.cnblogs.com/man1s/p/10267669.html）**类型查询
>
> explain 显示using index condition标志使用了MRR

优点

- MRR 使数据访问变得较为顺序：查询辅助索引后，首先将查询到的主键排序，并按照主键排序进行聚簇索引查找
- 减少缓冲页被替换的次数
- 批量处理对键值的查询操作

对于Innodb和MyISAM的范围查询和join查询，MRR工作方式：

- 将查到的辅助索引键、值放于一个缓存（此时缓存是按索引排序的）
- 将缓存中的键值按RowID（https://blog.csdn.net/zhaoyangjian724/article/details/49303797）排序
- 按RowID顺序访问实际数据文件

Index Condition Pushdown优化 ICP

> explain 显示using index condition标志使用了ICP

假如有联合索引，而查询条件有多个，且联合索引包含查询条件，那么取出主键时，会进行过滤，再去获取记录。

hash索引

> 假如当前服务器内存128g，怎么从内存中得到某一个被缓存的页呢？每次遍历时间复杂度高，这时hash可以实现，O1级别

实现：5.7哈希算法

- 冲突-链表，
- hash函数-取余

以O(1)时间查找，但失去了有序性

- 无法排序与分组
- 只支持精确查找，无法部分查找和范围查找

InnoDb：自适应哈希索引，当某个索引被使用频繁时，会在B+树索引之上再创建一个哈希索引

全文索引

- select * from users where username like '%xxx%' 无法使用b+索引，可以使用全文索引
- 倒排索引（inverted index）：实现全文索引的方法
    - 它在**辅助表（auxiliary table）**中存储了单词与单词自身在一个或多个文档中所在位置的映射，有两种表现形式：
        - inverted file index 表现形式：{单词， 单词所在文档ID}

        - full inverted index 表现形式：{单词，(单词所在文档ID, 在具体文档中的位置)}
    - inverted file index 先查到所在文档，再对该文档全文查询；full inverted index 则可以直接拿到位置
- InnoDb全文索引
    - 采用full inverted index，**辅助表**放在磁盘中，且有6张**辅助表**，提高并行
    - FTS(Full Text Search) Index Cache 是一颗**内存中的红黑树**，根据（word, ilist/）排序，(ilist: (documentID, position), word 单词)
        - 查找时，可能FTS缓存改变了，但没有写入磁盘，这时会将FTS中的要查的word的节点合并到辅助表
        - FTS不会每次被修改就写入磁盘，而是批量更新辅助表
- 全文检索用法
    - select * from users where match(username) against ('xxx')

索引优化

- 独立的列

  索引不能是表达式的一部分，也不能是函数参数
- 覆盖索引
- 索引顺序
    - 将选择性最强的列放在前面
- 前缀索引
    - 对于blog、text、varchar使用前缀索引，只索引开始部分字符
    - 语法 key(username(7))，username前7字符索引

慢查询

explain

- select_type 查询类型
- key：使用索引
- rows：扫描行数

优化数据访问

1. 减少请求数据量
    - 只返回必要的列，避免select *
    - 只返回必要的行，limit
    - 缓存重复查询数据
2. 减少引擎扫描行数
    - 覆盖索引
    - IPC

重构查询方式

- 切分大查询

  大查询假如一次性执行可能一次锁住很多数据，性能开销大
- 分解大连接查询

先单表查询，查出后在代码中进行关联

优点

- 缓存更高效：连接后的结果某张单表变化就需要失效，而分开只会失效一张表的结果
- 减少冗余查询：分解成多个单表查询，缓存结果可能被其他查询用到
- 减少锁竞争
- 更容易对数据库拆分，更容易实现高性能、可伸缩