高性能表结构及索引设计最佳实践

有道云链接：http://note.youdao.com/noteshare?id=fd5eae820148eb4f7bd9ced08f48aab4&sub=16864635FE004046831CAF4D0B37E087

课程学习须知

课程章节安排

图灵学院-MySQL进阶课程表

章节	章节名称
1	高性能业务表结构及索引设计最佳实践
2	Mysql内核查询成本计算实战
3	从架构师角度全局理解Mysql性能优化
4	Mysql执行原理之索引合并详解
5	Mysql内核查询优化规则详解
6	InnoDB引擎底层原理及Mysql 8.0新增特性详解

本课程是在第四期MySQL课程上做的一个升级和加深，会更加关注于MySQL内核执行和底层原理，所以请在学习掌握了第四期MySQL课程后再学习本课程，而且在第四期MySQL课程中已经讲述过的内容在本课程中不再重复讲述。同时：

1、一个知识点如果大部分同学明白，不会重复讲解，未明白的同学请看视频、笔记、请教同学或加老师QQ。

2、以上为本课的章节安排，不是课时安排，如果一章内容在一次课内未讲完，则会顺延到后面的课程继续讲解。

3、课程章节用不同的颜色标注了难度：

绿色较容易

红色表示进阶知识，较难，需要思考和反复学习

MySQL的版本众多，目前最新版本为MySQL8，考虑到实际的情况，本次课程以CentOS7下MySQL5.7.32社区 版本进行讲解，但是所讲述的内容在MySQL8中亦可适用。

如何安装运行MySQL，请咨询班主任获得往期视频讲解，并自行安装。如安装过程出错，请保证安装包完整无误,依赖包无误，并仔细阅读安装错误日志和检查操作系统层面的用户、用户组、文件和目录是否存在，各种权限是否正确等！同时安装Windows下的MySQL并不影响本课程的学习！

快速复习第四期MySQL

深入理解Mysql索引底层数据结构与算法

索引底层数据结构红黑树、B+树详解

面试常问的B树与B+树的区别是什么

索引在B+树上如何快速定位

千万级数据表如何用B+树索引快速查找

MylSAM与Innodb存储引擎底层索引实现区别

聚集索引、聚簇索引与稀疏索引到底是什么

为什么推荐使用自增整型的主键而不是UUID

很少使用的索引底层结构Hash是怎样的

联合索引底层数据存储结构又是怎样的

索引最左前缀原则底层实现原理

Explain详解与索引最佳实践

Mysql执行计划Explain工具详解

Mysql优化经常用到的覆盖索引详解

从B+树底层来分析下常见索引优化规则

经常用到的like查询应该如何优化

索引优化最佳实践

一条SQL在MySQL中是如何执行的

梳理下MySQL内部组件结构

为什么说Mysql的查询缓存很鸡肋

Mysql词法分析器原理详解

Mysql底层优化器与执行器详解

Mysql归档日志bin-log详解

不小心删库了如何快速恢复

Mysql索引优化实战

Mysql索引下推优化详解

为什么范围查找Mysql没有用索引下推优化

Mysql内部选择索引机制揭秘

Mysql索引成本计算工具trace详解

看下常用的Order by与Group by优化细节

Using filesort文件排序原理详解

文件单路排序与双路排序详细过程

文件排序优化机制详解

互联网公司索引设计核心原则

社交场景APP索引设计优化实战

Mysql索引优化实战二

最常用的分页查询如何高效优化

Join表关联查询优化

表关联嵌套循环连接Nested-Loop Join(NLJ)算法详解

基于块的嵌套循环连接Block Nested-Loop Join(BNL)算法

in和exsits优化细节小表驱动大表详解

count查询的各种形式优化细节

阿里巴巴Mysql优化规范详解

MySQL数据类型选择优化

深入理解Mysql事务隔离级别与锁机制

Mysql事务及其ACID属性详解

Mysql事务隔离级别详解

Mysql底层锁机制详解

实例演示各种事务隔离级别效果

Mysql底层脏读与幻读如何解决

Mysql底层间隙锁(Gap Lock)详解与优化

Mysql底层临键锁(Next-key Locks)详解

lnnoDB的行锁到底锁的是什么

深入理解MVCC与BufferPool缓存机制

彻底理解MVCC多版本并发控制机制

undo日志版本链与read view机制详解

通过实例演示理解MVCC内部版本链比对规则

lnnodb引擎SQL执行的BufferPool缓存机制

数据库表设计

良好的表结构设计是高性能的基石，应该根据系统将要执行的业务查询来设计，这往往需要权衡各种因素。糟糕的表结构设计，会浪费大量的开发时间，严重延误项目开发周期，让人痛苦万分，而且直接影响到数据库的性能，并需要花费大量不必要的优化时间，效果往往还不怎么样。

在数据库表设计上有个很重要的设计准则，称为范式设计。

范式设计

什么是范式?

范式来自英文Normal Form，简称NF。MySQL是关系型数据库，但是要想设计一个好的关系，必须使关系满足一定的约束条件，此约束已经形成了规范，分成几个等级，一级比一级要求得严格。满足这些规范的数据库是简洁的、结构明晰的，同时，不会发生插入(insert)、删除(delete)和更新(update)操作异常。反之则是乱七八糟，不仅给数据库的编程人员制造麻烦，而且面目可憎，可能存储了大量不需要的冗余信息。

目前关系数据库有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式）。满足最低要求的范式是第一范式（1NF）。在第一范式的基础上进一步满足更多规范要求的称为第二范式（2NF），其余范式以次类推。一般来说，数据库只需满足第三范式(3NF）就行了。

数据库设计的第一范式

定义： 属于第一范式关系的所有属性都不可再分，即数据项不可分。

理解： 第一范式强调数据表的原子性，是其他范式的基础。例如下表

id	name-age
1	张三-23

name-age列具有两个属性，一个name,一个 age不符合第一范式，把它拆分成两列

id	name	age
1	张三	23

上表就符合第一范式关系。但日常生活中仅用第一范式来规范表格是远远不够的，依然会存在数据冗余过大、删除异常、插入异常、修改异常的问题，此时就需要引入规范化概念，将其转化为更标准化的表格，减少数据依赖。

实际上，1NF是所有关系型数据库的最基本要求，你在关系型数据库管理系统（RDBMS），例如SQL Server，Oracle，MySQL中创建数据表的时候，如果数据表的设计不符合这个最基本的要求，那么操作一定是不能成功的。也就是说，只要在RDBMS中已经存在的数据表，一定是符合1NF的。

数据库设计的第二范式

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。

第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分。通常在实现来说，需要为表加上一个列，以存储各个实例的惟一标识。例如员工信息表中加上了员工编号（emp_id）列，因为每个员工的员工编号是惟一的，因此每个员工可以被惟一区分。这个惟一属性列被称为主关键字或主键、主码。

也就是说要求表中只具有一个业务主键，而且第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。什么意思呢？

有两张表：订单表，产品表

订单表ID（主键）	订单时间	产品ID
1	2018-12-12	3
1	2018-12-12	4

产品表ID	产品名称
2	娃娃
3	飞机
4	java入门

一个订单有多个产品，所以订单的主键为【订单ID】和【产品ID】组成的联合主键，这样2个组件不符合第二范式，而且产品ID和订单ID没有强关联，故，把订单表进行拆分为订单表与订单与商品的中间表

订单表ID	订单时间
1	2018-12-12

订单-商品中间表ID	订单ID	产品ID
1	1	3
2	1	4

产品表ID	产品名称
2	娃娃
3	飞机
4	java入门

数据库设计的第三范式

指每一个非主属性既不部分依赖于也不传递依赖于业务主键，也就是在第二范式的基础上消除了非主键对主键的传递依赖。例如，存在一个部门信息表，其中每个部门有部门编号（dept_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。

订单表

ID-订单编号(主键)	订单时间	产品ID	产品名称
1	2020-01-01	1	机械键盘
2	2020-01-02	2	蓝牙音响

产品ID	产品名称
1	机械键盘
2	蓝牙音响
3	蓝牙鼠标

其中

产品 ID与订单编号存在关联关系

产品名称与订单编号存在关联关系

产品ID与产品名称存在关联关系

订单表里如果如果产品ID发生改变，同一个表里产品名称也要跟着改变，这样不符合第三范式，应该把产品名称这一列从订单表中删除。

范式说明

真正的数据库范式定义上，相当难懂，比如第二范式（2NF）的定义“若某关系R属于第一范式，且每一个非主属性完全函数依赖于任何一个候选码，则关系R属于第二范式。”，这里面有着大堆专业术语的堆叠，比如“函数依赖”、“码”、“非主属性”、与“完全函数依赖”等等，而且有完备的公式定义，需要仔细研究的同学，请参考这本书：

反范式设计

什么叫反范式化设计

完全符合范式化的设计真的完美无缺吗？很明显在实际的业务查询中会大量存在着表的关联查询，而大量的表关联很多的时候非常影响查询的性能。

所谓得反范式化就是为了性能和读取效率得考虑而适当得对数据库设计范式得要求进行违反。允许存在少量得冗余，换句话来说反范式化就是使用空间来换取时间。

反范式设计-商品信息

下面是范式设计的商品信息表

商品信息和分类信息经常一起查询，所以把分类信息也放到商品表里面，冗余存放。

范式化和反范式总结

范式化设计优缺点

1、范式化的更新操作通常比反范式化要快。

2、当数据较好地范式化时，就只有很少或者没有重复数据，所以只需要修改更少的数据。

3、范式化的表通常更小，可以更好地放在内存里，所以执行操作会更快。

4、很少有多余的数据意味着检索列表数据时更少需要DISTINCT或者GROUP BY语句。在非范式化的结构中必须使用DISTINCT或者GROUPBY才能获得一份唯一的列表，但是如果是一张单独的表，很可能则只需要简单的查询这张表就行了。

范式化设计的缺点是通常需要关联。稍微复杂一些的查询语句在符合范式的表上都可能需要至少一次关联，也许更多。这不但代价昂贵，也可能使一些索引策略无效。例如，范式化可能将列存放在不同的表中，而这些列如果在一个表中本可以属于同一个索引。

反范式化设计优缺点

1、反范式设计可以减少表的关联

2、可以更好的进行索引优化。

反范式设计缺点也很明显，1、存在数据冗余及数据维护异常，2、对数据的修改需要更多的成本。

实际工作中的反范式实现

性能提升-缓存和汇总

范式化和反范式化的各有优劣，怎么选择最佳的设计?

请记住：小孩子才做选择，我们全都要；小孩才分对错，大人只看利弊。

而现实也是，完全的范式化和完全的反范式化设计都是实验室里才有的东西，在真实世界中很少会这么极端地使用。在实际应用中经常需要混用。

最常见的反范式化数据的方法是复制或者缓存，在不同的表中存储相同的特定列。
比如从父表冗余一些数据到子表的。前面我们看到的分类信息放到商品表里面进行冗余存放就是典型的例子。
缓存衍生值也是有用的。如果需要显示每个用户发了多少消息，可以每次执行一个对用户发送消息进行count的子查询来计算并显示它，也可以在user表用户中建一个消息发送数目的专门列，每当用户发新消息时更新这个值。
有需要时创建一张完全独立的汇总表或缓存表也是提升性能的好办法。“缓存表”来表示存储那些可以比较简单地从其他表获取（但是每次获取的速度比较慢）数据的表（例如，逻辑上冗余的数据)。而“汇总表”时,则保存的是使用GROUP BY语句聚合数据的表。
在使用缓存表和汇总表时，有个关键点是如何维护缓存表和汇总表中的数据，常用的有两种方式，实时维护数据和定期重建，这个取决于应用程序，不过一般来说，缓存表用实时维护数据更多点，往往在一个事务中同时更新数据本表和缓存表，汇总表则用定期重建更多，使用定时任务对汇总表进行更新。

性能提升-计数器表

计数器表在Web应用中很常见。比如网站点击数、用户的朋友数、文件下载次数等。对于高并发下的处理，首先可以创建一张独立的表存储计数器，这样可使计数器表小且快，并且可以使用一些更高级的技巧。
比如假设有一个计数器表，只有一行数据，记录网站的点击次数，网站的每次点击都会导致对计数器进行更新，问题在于，对于任何想要更新这一行的事务来说，这条记录上都有一个全局的互斥锁(mutex)。这会使得这些事务只能串行执行，会严重限制系统的并发能力。
怎么改进呢？可以将计数器保存在多行中，每次随机选择一行进行更新。在具体实现上，可以增加一个槽（slot)字段，然后预先在这张表增加100行或者更多数据，当对计数器更新时，选择一个随机的槽（slot)进行更新即可。
这种解决思路其实就是写热点的分散，在JDK的JDK1.8中新的原子类LongAdder也是这种处理方式，而我们在实际的缓冲中间件Redis等的使用、架构设计中，可以采用这种写热点的分散的方式，当然架构设计中对于写热点还有削峰填谷的处理方式，这种在MySQL的实现中也有体现，我们后面会讲到。

反范式设计-分库分表中的查询

例如,用户购买了商品,需要将交易记录保存下来,那么如果按照买家的纬度分表,则每个买家的交易记录都被保存在同一表中，我们可以很快、 很方便地查到某个买家的购买情况，但是某个商品被购买的交易数据很有可能分布在多张表中，查找起来比较麻烦 。

反之，按照商品维度分表，则可以很方便地查找到该商品的购买情况，但若要查找到买家的交易记录，则会比较麻烦 。

所以常见的解决方式如下。

（ 1 ）在多个分片表查询后合并数据集，这种方式的效率很低。

（ 2 ）记录两份数据，一份按照买家纬度分表，一份按照商品维度分表，，

（ 3 ）通过搜索引擎解决，但如果实时性要求很高，就需要实现实时搜索

在某电商交易平台下，可能有买家查询自己在某一时间段的订单，也可能有卖家查询自己在某一时间段的订单，如果使用了分库分表方案，则这两个需求是难以满足的，因此，通用的解决方案是，在交易生成时生成一份按照买家分片的数据副本和一份按照卖家分片的数据副本，查询时分别满足之前的两个需求，因此，查询的数据和交易的数据可能是分别存储的，并从不同的系统提供接口。

字段数据类型优化

请参考第四期MySQL的《5、Mysql索引优化实战二》。

MySQL索引辨析

范例表说明

在我们的后续课程中，会经常使用到InnoDB表order_exp，基础表结构如下：

```
CREATE TABLE `order_exp` (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的编号',
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的说明',
  `insert_time` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(0) COMMENT '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE,
  INDEX `idx_expire_time`(`expire_time`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 10819 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

数据量大概在1万多行。

```
9  select count(*) from order_exp
```

信息 结果 1 剖析 状态

count(*)
10567

同时还有个几个派生表如s1，s2，order_exp_cut，表结构基本和order_exp一致，有少许差别，数据量上也比order_exp少：

```
CREATE TABLE `order_exp_cut` (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '订单的编
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的说明',
  `insert_time` datetime(0) NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(0) COMMENT
  '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT
  '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE
```

```
CREATE TABLE `s2`  (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的编
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的
  `insert_time` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(
  '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT
  '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE,
  INDEX `idx_insert_time`(`insert_time`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 10814 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FOR
Dynamic;
```

InnoDB中的索引复习

InnoDB中的索引是按照B+树来组织的，至于什么是B+树，请参考第四期MySQL的《01、深入理解Mysql索引底层数据结构与算法》。我们知道B+树的叶子节点用来放数据的，但是放什么数据呢？索引自然是要放的，因为B+树的作用本来就是就是为了快速检索数据而提出的一种数据结构，不放索引放什么呢？但是数据库中的表，数据才是我们真正需要的数据，索引只是辅助数据，甚至于一个表可以没有自定义索引。InnoDB中的数据到底是如何组织的？

聚集索引/聚簇索引

InnoDB中使用了聚集索引，就是将表的主键用来构造一棵B+树，并且将整张表的行记录数据存放在该B+树的叶子节点中。也就是所谓的索引即数据，数据即索引。由于聚集索引是利用表的主键构建的，所以每张表只能拥有一个聚集索引。

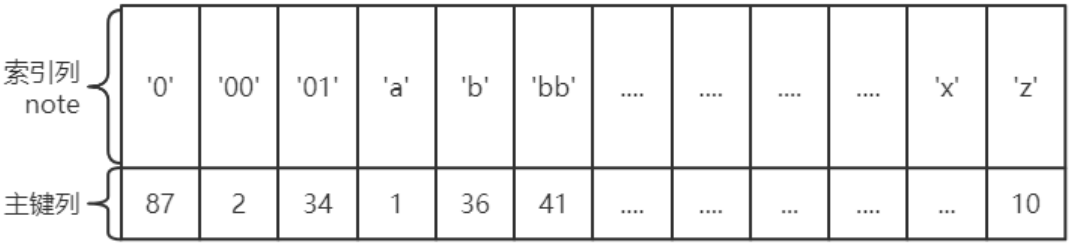
聚集索引的叶子节点就是数据页。换句话说，数据页上存放的是完整的每行记录。因此聚集索引的一个优点就是：通过过聚集索引能获取完整的整行数据。另一个优点是：对于主键的排序查找和范围查找速度非常快。

如果我们没有定义主键呢？MySQL会使用唯一性索引，没有唯一性索引，MySQL也会创建一个隐含列RowID来做主键，然后用这个主键来建立聚集索引。

辅助索引/二级索引

上边介绍的聚簇索引只能在搜索条件是主键值时才能发挥作用，因为B+树中的数据都是按照主键进行排序的,那如果我们想以别的列作为搜索条件怎么办？我们一般会建立多个索引，这些索引被称为辅助索引/二级索引。

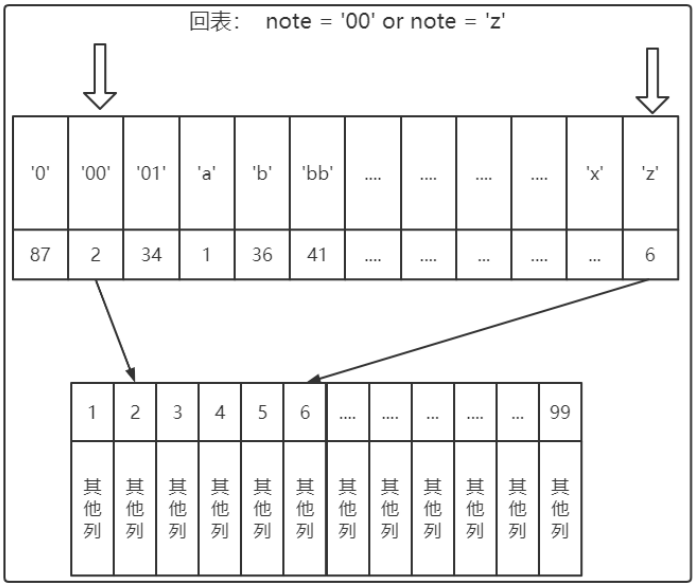
对于辅助索引(Secondary Index，也称二级索引、非聚集索引)，叶子节点并不包含行记录的全部数据。叶子节点除了包含键值以外，每个叶子节点中的索引行中还包含了相应行数据的聚集索引键。



比如辅助索引index(node)，那么叶子节点中包含的数据就包括了(主键、note)。

回表

辅助索引的存在并不影响数据在聚集索引中的组织，因此每张表上可以有多个辅助索引。当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后再通过主键索引（聚集索引）来找到一个完整的行记录。这个过程也被称为**回表**。也就是根据辅助索引的值查询一条完整的用户记录需要使用到2棵B+树----一次辅助索引，一次聚集索引。



为什么我们还需要一次回表操作呢?直接把完整的用户记录放到辅助索引d的叶子节点不就好了么？如果把完整的用户记录放到叶子节点是可以不用回表，但是太占地方了，相当于每建立一棵B+树都需要把所有的用户记录再都拷贝一遍，这就有点太浪费存储空间了。而且每次对数据的变化要在所有包含数据的索引中全部都修改一次，性能也非常低下。

很明显，回表的记录越少，性能提升就越高，需要回表的记录越多，使用二级索引的性能就越低，甚至让某些查询宁愿使用全表扫描也不使用二级索引。

那什么时候采用全表扫描的方式，什么时候使用采用二级索引 + 回表的方式去执行查询呢？这个就是查询优化器做的工作，查询优化器会事先对表中的记录计算一些统计数据，然后再利用这些统计数据根据查询的条件来计算一下需要回表的记录数，需要回表的记录数越多，就越倾向于使用全表扫描，反之倾向于使用二级索引 + 回表的方式。具体怎么算的，我们后面会详细说到。

MRR

从上文可以看出，每次从二级索引中读取到一条记录后，就会根据该记录的主键值执行回表操作。而在某个扫描区间中的二级索引记录的主键值是无序的，也就是说这些二级索引记录对应的聚簇索引记录所在的页面的页号是无序的。

每次执行回表操作时都相当于要随机读取一个聚簇索引页面，而这些随机IO带来的性能开销比较大。MySQL中提出了一个名为Disk-Sweep Multi-Range Read (MRR，多范围读取)的优化措施，即先读取一部分二级索引记录，将它们的主键值排好序之后再统一执行回表操作。

相对于每读取一条二级索引记录就立即执行回表操作，这样会节省一些IO开销。使用这个 MRR优化措施的条件比较苛刻，所以我们直接认为每读取一条二级索引记录就立即执

行回表操作。MRR的详细信息，可以查询官方文档。

联合索引/复合索引

前面我们对索引的描述，隐含了一个条件，那就是构建索引的字段只有一个，但实践工作中构建索引的完全可以是多个字段。所以，将表上的多个列组合起来进行索引我们称之为联合索引或者复合索引，比如index(a,b)就是将a,b两个列组合起来构成一个索引。

千万要注意一点，建立联合索引只会建立1棵B+树，多个列分别建立索引会分别以每个列则建立B+树，有几个列就有几个B+树，比如，index(note)、index(b)，就分别对note,b两个列各构建了一个索引。

index(note,b)在索引构建上，包含了两个意思：

1、先把各个记录按照note列进行排序。

2、在记录的note列相同的情况下，采用b列进行排序

索引列 note	'01'	'01'	'01'	'02'	'03'	'03'	....	....	....	....	'30'	'30'
索引列 b	'0'	'00'	'aa'	'00'	'00'	'bb'	....	....	....	....	'x'	'z'
主键列	87	2	34	1	36	41	....	....	...	....	...	10

自适应哈希索引

InnoDB存储引擎除了我们前面所说的各种索引，还有一种自适应哈希索引，我们知道B+树的查找次数,取决于B+树的高度,在生产环境中,B+树的高度一般为3~4层,故需要3~4次的IO查询。

所以在InnoDB存储引擎内部自己去监控索引表，如果监控到某个索引经常用，那么就认为是热数据，然后内部自己创建一个hash索引，称之为自适应哈希索引(Adaptive Hash Index,AHI)，创建以后，如果下次又查询到这个索引，那么直接通过hash算法推导出记录的地址，直接一次就能查到数据，比重复去B+tree索引中查询三四次节点的效率高了不少。

InnoDB存储引擎使用的哈希函数采用除法散列方式，其冲突机制采用链表方式。注意，对于自适应哈希索引仅是数据库自身创建并使用的，我们并不能对其进行干预。通过命令show engine innodb status\G可以看到当前自适应哈希索引的使用状况，如:

```
mysql> show engine innodb status\G
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
=====================================
2021-03-25 14:57:00 0x7f3dfc116700 INNODB MONITOR OUTPUT
=====================================
Per second averages calculated from the last 32 seconds
-----------------
BACKGROUND THREAD
-----------------
srv_master_thread loops: 134 srv_active, 0 srv_shutdown, 513170 srv_idle
srv_master_thread log flush and writes: 513304
----------
SEMAPHORES
----------
OS WAIT ARRAY INFO: reservation count 81
OS WAIT ARRAY INFO: signal count 80
RW-shared spins 0, rounds 142, OS waits 71
RW-excl spins 0, rounds 90, OS waits 2
RW-sx spins 5, rounds 150, OS waits 5
Spin rounds per wait: 142.00 RW-shared, 90.00 RW-excl, 30.00 RW-sx
------------
TRANSACTIONS
------------
---------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
---------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 1 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 1 buffer(s)
Hash table size 34673, node heap has 1 buffer(s)
Hash table size 34673, node heap has 0 buffer(s)
Hash table size 34673, node heap has 2 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
```

哈希索引只能用来搜索等值的查询,如 SELECT* FROM table WHERE index co=xxx。而对于其他查找类型,如范围查找,是不能使用哈希索引的,

因此这里会显示non- hash searches/s的统计情况。通过 hash searches: non-hash searches可以大概了解使用哈希索引后的效率。

由于AHI是由 InnoDB存储引擎控制的,因此这里的信息只供我们参考。不过我们可以通过观察 SHOW ENGINE INNODB STATUS的结果及参数 innodb_adaptive_hash_index来考虑是禁用或启动此特性,默认AHI为开启状态。

什么时候需要禁用呢？如果发现监视索引查找和维护哈希索引结构的额外开销远远超过了自适应哈希索引带来的性能提升就需要关闭这个功能。

同时在MySQL 5.7中，自适应哈希索引搜索系统被分区。每个索引都绑定到一个特定的分区，每个分区都由一个单独的 latch 锁保护。分区由 innodb_adaptive_hash_index_parts 配置选项控制 。在早期版本中，自适应哈希索引搜索系统受到单个 latch 锁的保护，这可能成为繁重工作负载下的争用点。innodb_adaptive_hash_index_parts 默认情况下，该 选项设置为8。最大设置为512。当然禁用或启动此特性和调整分区个数这个应该是DBA的工作，我们了解即可。

全文检索之倒排索引

什么是全文检索（Full-Text Search）？它是将存储于数据库中的整本书或整篇文章中的任意内容信息查找出来的技术。它可以根据需要获得全文中有关章、节、段、句、词等信息，也可以进行各种统计和分析。我们比较熟知的Elasticsearch、Solr等就是全文检索引擎，底层都是基于Apache Lucene的。

举个例子，现在我们要保存唐宋诗词，数据库中我们们会怎么设计？诗词表我们可能的设计如下：

朝代	作者	诗词年代	标题	诗词全文
唐	李白		静夜思	床前明月光，疑是地上霜。 举头望明月，低头思故乡。
宋	李清照		如梦令	常记溪亭日暮，沉醉不知归路，兴尽晚回舟，误入藕花深处。争渡，争渡，惊起一滩鸥鹭。
….	….	…	….	…….

要根据朝代或者作者寻找诗，都很简单，比如“select 诗词全文 from 诗词表 where作者=‘李白’”，如果数据很多，查询速度很慢，怎么办？我们可以在对应的查询字段上建立索引加速查询。

但是如果我们现在有个需求：要求找到包含“望”字的诗词怎么办？用“select 诗词全文 from 诗词表 where诗词全文 like ‘%望%’”，这个意味着要扫描库中的诗词全文字段，逐条比对，找出所有包含关键词“望”字的记录，。基本上，数据库中一般的SQL优化手段都是用不上的。数量少，大概性能还能接受，如果数据量稍微大点，就完全无法接受了，更何况在互联网这种海量数据的情况下呢？怎么解决这个问题呢，用倒排索引。

比如现在有：

蜀道难（唐）李白 蜀道之难难于上青天，侧身西望长咨嗟。

静夜思（唐）李白 举头望明月，低头思故乡。

春台望（唐）李隆基 暇景属三春，高台聊四望。

鹤冲天(宋)柳永 黄金榜上，偶失龙头望。明代暂遗贤，如何向？未遂风云便，争不恣狂荡。何须论得丧？才子词人，自是白衣卿相。烟花巷陌，依约丹青屏障。幸有意中人，堪寻访。且恁偎红翠，风流事，平生畅。青春都一饷。忍把浮名，换了浅斟低唱！

都有望字，于是我们可以这么保存

序号	关键字	蜀道难	静夜思	春台望	鹤冲天
1	望	有	有	有	有

如果查哪个诗词中包含上，怎么办，上述的表格可以继续填入新的记录

序号	关键字	蜀道难	静夜思	春台望	鹤冲天
1	望	有	有	有	有
2	上	有			有

其实，上述诗词的中每个字都可以作为关键字，然后建立关键字和文档之间的对应关系，也就是标识关键字被哪些文档包含。

所以，倒排索引就是，将文档中包含的关键字全部提取处理，然后再将关键字和文档之间的对应关系保存起来，最后再对关键字本身做索引排序。用户在检索某一个关键字是，先对关键字的索引进行查找，再通过关键字与文档的对应关系找到所在文档。

在存储在关系型数据库中的数据，需要我们事先分析将数据拆分为不同的字段，而在es这类的存储中，需要应用程序根据规则自动提取关键字，并形成对应关系。

这些预先提取的关键字，在全文检索领域一般被称为term（词项），文档的词项提取在es中被称为文档分析，这是全文检索很核心的过程，必须要区分哪些是词项，哪些不是，比如很多场景下，apple和apples是同一个东西，望和看其实是同一个动作。

MySQL中的全文索引

MySQL 5.6 以前的版本，只有 MyISAM 存储引擎支持全文索引。从InnoDB 1.2.x版本开始，InnoDB存储引擎开始支持全文检索，对应的MySQL版本是5.6.x系列。

注意，不管什么引擎，只有字段的数据类型为 char、varchar、text 及其系列才可以建全文索引。

不过MySQL从设计之初就是关系型数据库，存储引擎虽然支持全文检索，整体架构上对全文检索支持并不好而且限制很多，比如每张表只能有一个全文检索的索引，不支持没有单词界定符(delimiter）的语言，如中文、日语、韩语等。

所以如果有大批量或者专门的全文检索需求，还是应该选择专门的全文检索引擎，毕竟Elastic靠着全文检索起家，然后产品化、公司化后依赖全文检索不断扩充产品线和应用场景，并推出商业版本的解决方案然后融资上市，现在的市值已达100亿美元（2021/12/06 -纽约证券交易所中的市值101.5亿美元）。

具体如何使用InnoDB存储引擎的全文检索，只提供简单的使用说明，更多的详情请自行查阅相关官方文档或者书籍，我们不做任何技术支持。官方文档路径：

https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html

创建

创建表时创建全文索引

```
create table fulltext_test (
    id int(11) NOT NULL AUTO_INCREMENT,
    content text NOT NULL,
    tag varchar(255),
    PRIMARY KEY (id),
    FULLTEXT KEY content_tag_fulltext(content,tag)
) DEFAULT CHARSET=utf8;
```

在已存在的表上创建全文索引

```
create fulltext index content_tag_fulltext
    on fulltext_test(content,tag);
```

通过 SQL 语句 ALTER TABLE 创建全文索引

```
alter table fulltext_test
    add fulltext index content_tag_fulltext(content,tag);
```

使用全文索引

和常用的模糊匹配使用 like + % 不同，全文索引有自己的语法格式，使用 match 和 against 关键字，比如

```
select * from fulltext_test
    where match(content,tag) against('xxx xxx');
```

Elastic发家小故事

多年前，一个叫做 Shay Banon 的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本。

直接基于 Lucene 工作会比较困难，所以 Shay 开始抽象 Lucene 代码以便 Java 程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“ Compass”。

后来 Shay 找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写 Compass 库使其成为一个独立的服务叫做 Elasticsearch。

第一个公开版本出现在 2010 年 2 月，在那之后 Elasticsearch 已经成为 Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营 Elasticsearch 的公司就此成立，他们一边提供商业支持一边开发新功能，不过 Elasticsearch 将永远开源且对所有人可用。

总结：MySQL有哪些索引类型

从数据结构角度可分为B+树索引、哈希索引、以及FULLTEXT索引（现在MyISAM和InnoDB引擎都支持了）和R-Tree索引（用于对GIS数据类型创建SPATIAL索引）；

从物理存储角度可分为聚集索引（clustered index）、非聚集索引（non-clustered index）；

从逻辑角度可分为主键索引、普通索引，或者单列索引、多列索引、唯一索引、非唯一索引等等。

面试题：什么是密集索引和稀疏索引？

面试中还会被问到什么是密集索引和稀疏索引。

密集索引的定义：叶子节点保存的不只是键值，还保存了位于同一行记录里的其他列的信息，由于密集索引决定了表的物理排列顺序，一个表只有一个物理排列顺序，所以一个表只能创建一个密集索引。

稀疏索引：叶子节点仅保存了键位信息以及该行数据的地址，有的稀疏索引只保存了键位信息机器主键。

mysam存储引擎，不管是主键索引，唯一键索引还是普通索引都是稀疏索引，innodb存储引擎：有且只有一个密集索引。

所以，密集索引就是innodb存储引擎里的聚簇索引，稀疏索引就是innodb存储引擎里的普通二级索引。

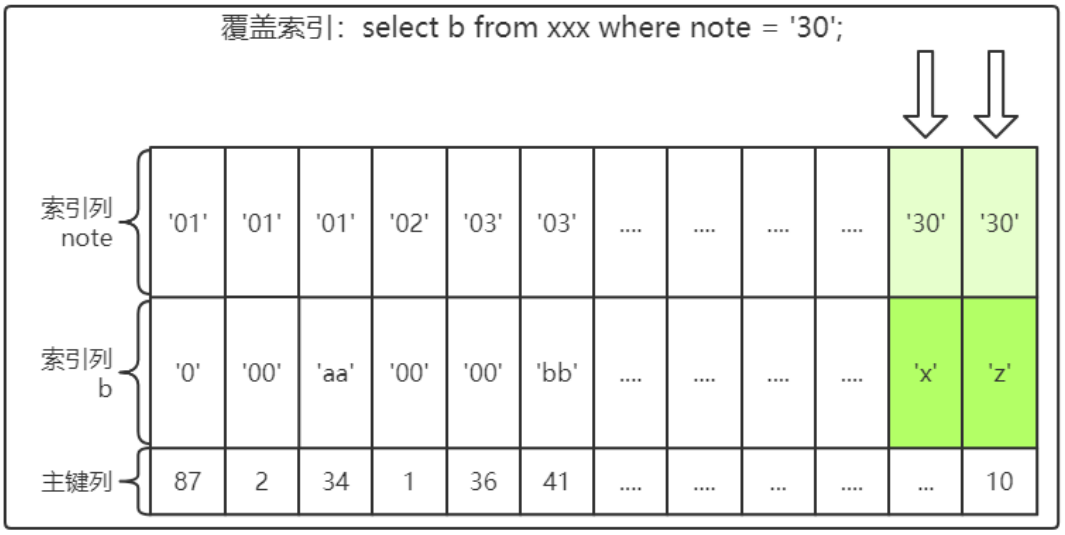
辨析覆盖索引/索引覆盖

既然多个列可以组合起来构建为联合索引，那么辅助索引自然也可以由多个列组成。

覆盖索引也是我们经常见到的名词，InnoDB存储引擎支持覆盖索引(covering index，或称索引覆盖)，即从辅助索引中就可以得到查询的记录，而不需要查询聚

集索引中的记录。使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息，故其大小要远小于聚集索引，因此可以减少大量的IO操作。所以记住，覆盖索引可以视为索引优化的一种方式，而并不是索引类型的一种。

除了覆盖索引这个概念外，在索引优化的范围内，还有前缀索引、三星索引等一系列概念，都会在我们后面的课程中学习到。



深入思考索引在查询中的使用

索引在查询中的作用到底是什么？在我们的查询中发挥着什么样的作用呢？

请记住：

1、**一个索引就是一个B+树，索引让我们的查询可以快速定位和扫描到我们需要的数据记录上，加快查询的速度。**

2、**一个select查询语句在执行过程中一般最多能使用一个二级索引来加快查询，即使在where条件中用了多个二级索引。**

索引的代价

世界上从来没有只有好处没有坏处的东西，如果你有，请你一定要告诉我，让我也感受一下。虽然索引是个好东西，在学习如何更好的使用索引之前先要了解一下使用它的代价，它在空间和时间上都会拖后腿。

空间上的代价

这个是显而易见的，每建立一个索引都要为它建立一棵B+树，每一棵B+树的每一个节点都是一个数据页，一个页默认会占用16KB的存储空间，一棵很大的B+树由许多数据页组成会占据很多的存储空间。

时间上的代价

每次对表中的数据进行增、删、改操作时，都需要去修改各个B+树索引。而且我们讲过，B+树每层节点都是按照索引列的值从小到大的顺序排序而组成了双向链表。不论是

叶子节点中的记录，还是非叶子内节点中的记录都是按照索引列的值从小到大的顺序而形成了一个单向链表。

而增、删、改操作可能会对节点和记录的排序造成破坏，所以存储引擎需要额外的时间进行一些记录移位，页面分裂、页面回收的操作来维护好节点和记录的排序。如果我们建了许多索引，每个索引对应的B+树都要进行相关的维护操作，这必然会对性能造成影响。

既然索引这么有用，我们是不是创建越多越好？既然索引有代价，我们还是别创建了吧？当然不是！按照经验，一般来说，一张表6-7个索引以下都能够取得比较好的性能权衡。

那么创建索引的时候有什么好的策略让我们充分利用索引呢？

高性能的索引创建策略

正确地创建和使用索引是实现高性能查询的基础。前面我们已经了解了索引相关的数据结构，各种类型的索引及其对应的优缺点。现在我们一起来看看如何真正地发挥这些索引的优势。

索引列的类型尽量小

我们在定义表结构的时候要显式的指定列的类型，以整数类型为例，有TTNYINT、NEDUMNT、INT、BIGTNT这么几种，它们占用的存储空间依次递增，我们这里所说的类型大小指的就是该类型表示的数据范围的大小。能表示的整数范围当然也是依次递增，如果我们想要对某个整数列建立索引的话，在表示的整数范围允许的情况下，尽量让索引列使用较小的类型，比如我们能使用INT就不要使用BIGINT，能使用NEDIUMINT就不要使用INT，这是因为:

- 数据类型越小，在查询时进行的比较操作越快（CPU层次）
- 数据类型越小，索引占用的存储空间就越少，在一个数据页内就可以放下更多的记录，从而减少磁盘/O带来的性能损耗，也就意味着可以把更多的数据页缓存在内存中，从而加快读写效率。

这个建议对于表的主键来说更加适用，因为不仅是聚簇索引中会存储主键值，其他所有的二级索引的节点处都会存储一份记录的主键值，如果主键适用更小的数据类型，也就意味着节省更多的存储空间和更高效的I/O。

利用索引选择性和前缀索引

索引的选择性/离散性

创建索引应该选择选择性/离散性高的列。索引的选择性/离散性是指，不重复的索引值（也称为基数，cardinality)和数据表的记录总数（N)的比值，范围从1/N到1之间。索

引的选择性越高则查询效率越高，因为选择性高的索引可以让MySQL在查找时过滤掉更多的行。唯一索引的选择性是1，这是最好的索引选择性，性能也是最好的。

很差的索引选择性就是列中的数据重复度很高，比如性别字段，不考虑政治正确的情况下，只有两者可能，男或女。那么我们在查询时，即使使用这个索引，从概率的角度来说，依然可能查出一半的数据出来。

比如下面这个表：

姓名	年龄	性别	区号
Peter	18	1	0731
Jack	20	1	0734
Allen	21	1	0731
Lisa	22	0	0746
Sam	23	1	0739
King	18	1	0745
James	20	1	0744
Mark	21	1	0731

哪列做为索引字段最好？当然是姓名字段，因为里面的数据没有任何重复，性别字段是最不适合做索引的，因为数据的重复度非常高。

怎么算索引的选择性/离散性？比如order_exp这个表：

```
select  COUNT(DISTINCT order_no)/count(*) cnt from order_exp;
```

```
mysql> select  COUNT(DISTINCT order_no)/count(*) cnt from order_exp;
+---------+
| cnt     |
+---------+
| 0.9676 |
+---------+
1 row in set (0.01 sec)
```

```
select  COUNT(DISTINCT order_status)/count(*) cnt from order_exp;
```

```
mysql> select  COUNT(DISTINCT order_status)/count(*) cnt from order_exp;
+---------+
| cnt     |
+---------+
| 0.0001 |
+---------+
1 row in set (0.00 sec)
```

很明显，order_no列上的索引就比order_status列上的索引的选择性就要好，原因很简单，因为order_status列中的值只有-1,0,1三种。

前缀索引

有时候需要索引很长的字符列，这会让索引变得大且慢。一个策略是前面提到过的模拟哈希索引。

模拟哈希索引：

order_exp表中order_note字段很长，想把它作为一个索引，我们可以增加一个order_not_hash字段来存储order_note的哈希值，然后在order_not_hash上建立索引，

相对于之前的索引速度会有明显提升，一个是对完整的 order_note做索引，而后者则是用整数哈希值做索引，显然数字的比较比字符串的匹配要高效得多。

但是缺陷也很明显：

1、需要额外维护order_not_hash字段；

2、哈希算法的选择决定了哈希冲突的概率，不良的哈希算法会导致重复值很多；

3、不支持范围查找。

还可以做些什么改进呢？还可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。但这样也会降低索引的选择性。一般情况下我们需要保证某个列前缀的选择性也是足够高的，以满足查询性能。（尤其对于BLOB、TEXT或者很长的VARCHAR类型的列，应该使用前缀索引，因为MySQL不允许索引这些列的完整长度）。

诀窍在于要选择足够长的前缀以保证较高的选择性，同时又不能太长（以便节约空间）。前缀应该足够长，以使得前缀索引的选择性接近于索引整个列。

按照《阿里最新Java编程规范泰山版》中《(二) 索引规约》中的说法：

3. 【强制】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度。

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分度会高达 90% 以上，可以使用 count(distinct left(列名, 索引长度))/count(*)的区分度来确定。

中建议，这个前缀的长度为20比较合适，但是其实后面的“count(distinct left(列名,索引长度))/count(*)的区分度来确定”会更合适点。在工程实践中具体如何做呢？

可以这样：

```
SELECT COUNT(DISTINCT LEFT(order_note,3))/COUNT(*) AS sel3,
COUNT(DISTINCT LEFT(order_note,4))/COUNT(*)AS sel4,
COUNT(DISTINCT LEFT(order_note,5))/COUNT(*) AS sel5,
COUNT(DISTINCT LEFT(order_note, 6))/COUNT(*) As sel6,
COUNT(DISTINCT LEFT(order_note, 7))/COUNT(*) As sel7,
COUNT(DISTINCT LEFT(order_note, 8))/COUNT(*) As sel8,
COUNT(DISTINCT LEFT(order_note, 9))/COUNT(*) As sel9,
COUNT(DISTINCT LEFT(order_note, 10))/COUNT(*) As sel10,
COUNT(DISTINCT LEFT(order_note, 11))/COUNT(*) As sel11,
COUNT(DISTINCT LEFT(order_note, 12))/COUNT(*) As sel12,
COUNT(DISTINCT LEFT(order_note, 13))/COUNT(*) As sel13,
COUNT(DISTINCT LEFT(order_note, 14))/COUNT(*) As sel14,
COUNT(DISTINCT LEFT(order_note, 15))/COUNT(*) As sel15,
COUNT(DISTINCT order_note)/COUNT(*) As total
FROM order_exp;
```

```
mysql> SELECT COUNT(DISTINCT LEFT(order_note,3))/COUNT(*) AS sel3,
    -> COUNT(DISTINCT LEFT(order_note,4))/COUNT(*)AS sel4,
    -> COUNT(DISTINCT LEFT(order_note,5))/COUNT(*) AS sel5,
    -> COUNT(DISTINCT LEFT(order_note, 6))/COUNT(*) As sel6,
    -> COUNT(DISTINCT LEFT(order_note, 7))/COUNT(*) As sel7,
    -> COUNT(DISTINCT LEFT(order_note, 8))/COUNT(*) As sel8,
    -> COUNT(DISTINCT LEFT(order_note, 9))/COUNT(*) As sel9,
    -> COUNT(DISTINCT LEFT(order_note, 10))/COUNT(*) As sel10,
    -> COUNT(DISTINCT LEFT(order_note, 11))/COUNT(*) As sel11,
    -> COUNT(DISTINCT LEFT(order_note, 12))/COUNT(*) As sel12,
    -> COUNT(DISTINCT LEFT(order_note, 13))/COUNT(*) As sel13,
    -> COUNT(DISTINCT LEFT(order_note, 14))/COUNT(*) As sel14,
    -> COUNT(DISTINCT LEFT(order_note, 15))/COUNT(*) As sel15,
    -> COUNT(DISTINCT LEFT(order_note, 16))/COUNT(*) As sel16,
    -> COUNT(DISTINCT order_note, 15)/COUNT(*) As total
    -> FROM order_exp;
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
| sel3   | sel4   | sel5   | sel6   | sel7   | sel8   | sel9   | sel10  | sel11  | sel12  | sel13  | sel14  | sel15  | sel16  | total  |
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
| 0.0008 | 0.0008 | 0.0008 | 0.0015 | 0.0107 | 0.0844 | 0.1628 | 0.3455 | 0.4723 | 0.6834 | 0.8564 | 0.9197 | 0.9592 | 0.9657 | 0.9676 |
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
1 row in set (0.14 sec)
```

可以看见，从第10个开始选择性的增加值很高，随着前缀字符的越来越多，选择度也在不断上升，但是增长到第15时，已经和第14没太大差别了，选择性提升的幅度已经很小了，都非常接近整个列的选择性了。

那么针对这个字段做前缀索引的话，从第13到第15都是不错的选择，甚至第12也不是不能考虑。

在上面的示例中，已经找到了合适的前缀长度，如何创建前缀索引:

```
ALTER TABLE order_exp ADD KEY (order_note(14));
```

建立前缀索引后查询语句并不需要更改：

```
select * from order_exp where order_note = 'xxxx' ;
```

前缀索引是一种能使索引更小、更快的有效办法，但另一方面也有其缺点MySQL无法使用前缀索引做ORDER BY和GROUP BY，也无法使用前缀索引做覆盖扫描。

有时候后缀索引 (suffix index)也有用途（例如，找到某个域名的所有电子邮件地址)。MySQL原生并不支持反向索引，但是可以把字符串反转后存储，并基于此建立前缀索引。可以通过触发器或者应用程序自行处理来维护索引。

只为用于搜索、排序或分组的列创建索引

也就是说，只为出现在WHERE 子句中的列、连接子句中的连接列创建索引，而出现在查询列表中的列一般就没必要建立索引了，除非是需要使用覆盖索引。又或者为出现在ORDER BY或GROUP BY子句中的列创建索引，这句话什么意思呢？比如：

```
SELECT * FROM order_exp ORDER BY insert_time, order_status,expire_time;
```

查询的结果集需要先按照insert_time值排序，如果记录的insert_time值相同，则需要按照order_status来排序，如果order_status的值相同，则需要按照expire_time排序。

回顾一下联合索引的存储结构，u_idx_day_status索引本身就是按照上述规则排好序的，所以直接从索引中提取数据，然后进行回表操作取出该索引中不包含的列就好了。

当然ORDER BY的子句后边的列的顺序也必须按照索引列的顺序给出，如果给出ORDER BY order_status,expire_time, insert_time的顺序，那也是用不了B+树索引的，原因不用再说了吧。

```
SELECT insert_time, order_status,expire_time,count(*) FROM order_exp GROUP BY insert_time, order_status,expire_time;
```

这个查询语句相当于做了3次分组操作：

先把记录按照insert_time值进行分组，所有insert_time值相同的记录划分为一组。

将每个insert_time值相同的分组里的记录再按照order_status的值进行分组，将order_status值相同的记录放到一个小分组里。

再将上一步中产生的小分组按照expire_time的值分成更小的分组。

然后针对最后的分组进行统计，如果没有索引的话，这个分组过程全部需要在内存里实现，而如果有了索引的话，恰巧这个分组顺序又和我们的u_idx_day_status索引中的索引列的顺序是一致的，而我们的B+树索引又是按照索引列排好序的，这不正好么，所以可以直接使用B+树索引进行分组。和使用B+树索引进行排序是一个道理，分组列的顺序也需要和索引列的顺序一致。

合理设计多列索引

很多人对多列索引的理解都不够。一个常见的错误就是，为每个列创建独立的索引，或者按照错误的顺序创建多列索引。

我们遇到的最容易引起困惑的问题就是索引列的顺序。正确的顺序依赖于使用该索引的查询，并且同时需要考虑如何更好地满足排序和分组的需要。反复强调过，在一个多列B-Tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列，等等。所以，索引可以按照升序或者降序进行扫描，以满足精确符合列顺序的ORDER BY、GROUP BY和DISTINCT等子句的查询需求。

所以多列索引的列顺序至关重要。对于如何选择索引的列顺序有一个经验法则：将选择性最高的列放到索引最前列。当不需要考虑排序和分组时，将选择性最高的列放在前面通常是很好的。这时候索引的作用只是用于优化WHERE条件的查找。在这种情况下，这样设计的索引确实能够最快地过滤出需要的行，对于在WHERE子句中只使用了索引部分前缀列的查询来说选择性也更高。

然而，性能不只是依赖于索引列的选择性，也和查询条件的有关。可能需要根据那些运行频率最高的查询来调整索引列的顺序，比如排序和分组，让这种情况下索引的选择性最高。

同时，在优化性能的时候，可能需要使用相同的列但顺序不同的索引来满足不同类型的查询需求。

尽可能设计三星索引

三星索引概念

对于一个查询而言，一个三星索引，可能是其最好的索引。

如果查询使用三星索引，一次查询通常只需要进行一次磁盘随机读以及一次窄索引片的扫描，因此其相应时间通常比使用一个普通索引的响应时间少几个数量级。

三星索引概念是在《Rrelational Database Index Design and the optimizers》 一书（这本书也是《高性能MySQL》作者强烈推荐的一本书）中提出来的。原文如下：

The index earns one star if it places relevant rows adjacent to each other,
a second star if its rows are sorted in the order the query needs,
and a final star if it contains all the columns needed for the query.

索引将相关的记录放到一起则获得一星；

如果索引中的数据顺序和查找中的排列顺序一致则获得二星；

如果索引中的列包含了查询中需要的全部列则获得三星。

二星（排序星）：

在满足一星的情况下，当查询需要排序，group by、 order by，如果查询所需的顺序与索引是一致的（索引本身是有序的），是不是就可以不用再另外排序了，一般来说排序可是影响性能的关键因素。

三星（宽索引星）：

在满足了二星的情况下，如果索引中所包含了这个查询所需的所有列（包括 where 子句和 select 子句中所需的列，也就是覆盖索引），这样一来，查询就不再需要回表了，减少了查询的步骤和IO请求次数，性能几乎可以提升一倍。

一星按照原文稍微有点难以理解，其实它的意思就是：如果一个查询相关的索引行是相邻的或者至少相距足够靠近的话，必须扫描的索引片宽度就会缩至最短，也就是说，让索引片尽量变窄，也就是我们所说的索引的扫描范围越小越好。

这三颗星，哪颗最重要？第三颗星。因为将一个列排除在索引之外可能会导致很多磁盘随机读（回表操作）。第一和第二颗星重要性差不多，可以理解为第三颗星比重是50%，第一颗星为27%，第二颗星为23%，所以在大部分的情况下，会先考虑第一颗星，但会根据业务情况调整这两颗星的优先度。

达成三星索引

现在有表

```
create table customer(
cno int,
lname varchar(10),
fname varchar(10),
sex int,
weight int,
city varchar(10));
```

建立索引

```
create index idx_cust on customer(city,lname,fname,cno);
```

对于下面的SQL而言，这是个三星索引

```
 select cno,fname from customer where lname =’ xx’  and city =’ yy’  order by fname;
```

来评估下：

第一颗星：所有等值谓词的列，是组合索引的开头的列，可以把索引片缩得很窄，符合。

第二颗星：order by的fname字段在组合索引中且是索引自动排序好的，符合。

第三颗星：select中的cno字段、fname字段在组合索引中存在，符合。

达不成三星索引

现在有表

```
CREATE TABLE `test` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) DEFAULT NULL,
  `sex` int(11) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `c_date` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  ) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8;
```

SQL语句如下：

```
select user_name,sex,age from test where user_name like ’test%’  and sex =1 ORDER BY age
```

如果我们建立索引(user_name,sex,age)：

第三颗星，满足

第一颗星，满足

第二颗星，不满足，user_name 采用了范围匹配，sex 是过滤列，此时age 列无法保证有序的。

上述我们看到，此时索引(user_name,sex,age)并不能满足三星索引中的第二颗星（排序）。

于是我们改改，建立索引(sex, age，user_name)：

第一颗星，不满足，只可以匹配到sex，sex选择性很差，意味着是一个宽索引片，

第二颗星，满足，等值sex 的情况下，age是有序的，

第三颗星，满足，select查询的列都在索引列中，

对于索引(sex,age，user_name)我们可以看到，此时无法满足第一颗星，窄索引片的需求。

以上2个索引，都是无法同时满足三星索引设计中的三个需求的，我们只能尽力满足2个。而在多数情况下，能够满足2颗星，已经能缩小很大的查询范围了，具体最终要保留那一颗星（排序星 or 窄索引片星），这个就需要看查询者自己的着重点了，无法给出标准答案。

主键尽量是很少改变的列

我们知道，行是按照聚集索引物理排序的，如果主键频繁改变(update)，物理顺序会改变，MySQL要不断调整B+树，并且中间可能会产生页面的分裂和合并等等，会导致性能会急剧降低。

处理冗余和重复索引

MySQL允许在相同列上创建多个索引，无论是有意的还是无意的。MySQL需要单独维护重复的索引，并且优化器在优化查询的时候也需要逐个地进行考虑，这会影响性能。重复索引是指在相同的列上按照相同的顺序创建的相同类型的索引。应该避免这样创建重复索引，发现以后也应该立即移除。

有时会在不经意间创建了重复索引，例如下面的代码:

```
CREATE TABLE test (
ID INT NOT NULL PRIMARY KEY,
A INT NOT NULL,
B INT NOT NULL,
UNIQUE(ID),
INDEX(ID)
) ENGINE=InnoDB;
```

这里创建了一个主键，又加上唯一限制，然后再加上索引以供查询使用。事实上，MySQL的唯一限制和主键限制都是通过索引实现的，因此，上面的写法实际上在相同的列上创建了三个重复的索引。通常并没有理由这样做，除非是在同一列上创建不同类型的索引来满足不同的查询需求。

冗余索引和重复索引有一些不同。如果创建了索引(A B)，再创建索引(A)就是冗余索引，因为这只是前一个索引的前缀索引。因此索引(AB)也可以当作索引(A)来使用（这种冗余只是对B-Tree索引来说的)。但是如果再创建索引 (B,A)，则不是冗余索引，索引(B)也不是，因为B不是索引(A,B)的最左前缀列。

已有的索引(A)，扩展为(A，ID)，其中ID是主键，对于InnoDB来说主键列已经包含在二级索引中了，所以这也是冗余的。

解决冗余索引和重复索引的方法很简单，删除这些索引就可以，但首先要做的是找出这样的索引。可以通过写一些复杂的访问INFORMATION_SCHEMA表的查询来找。

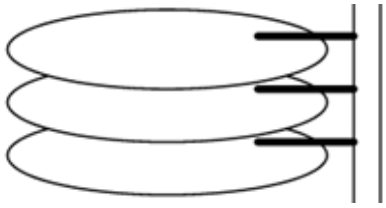
删除未使用的索引

除了冗余索引和重复索引，可能还会有一些服务器永远不用的索引。这样的索引完全是累赘，建议考虑删除。

补充资料：磁盘和B+树

为什么关系型数据库都选择了B+树，这个和磁盘的特性有着非常大的关系。

如果我们简化一下，可以这么看



一个磁盘由大小相同且同轴的圆形盘片组成，磁盘可以转动（各个磁盘必须同步转动）。在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。磁头不能转动，但是可以沿磁盘半径方向运动。

盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元也是最小读写单元。现在磁盘扇区一般是512个字节~4k个字节。

磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、扇区号。

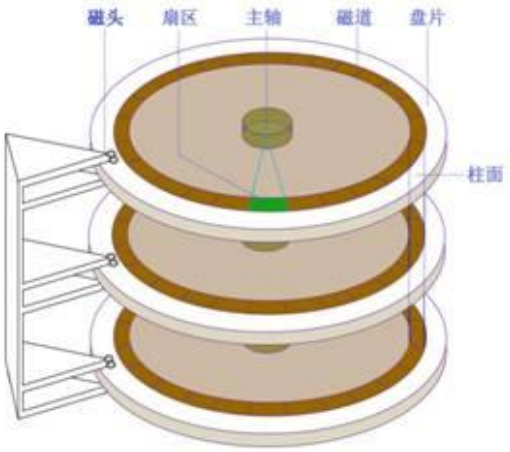
读/写磁盘上某一指定数据需要下面步骤：

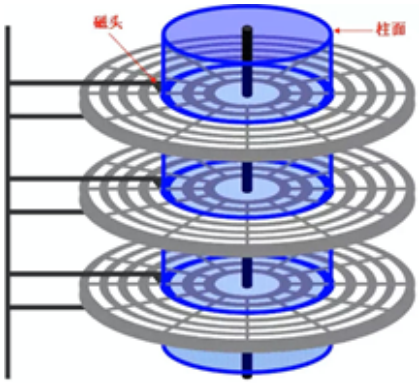
(1) 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找。

(2)所有磁头都定位到磁道上后，这时根据盘面号来确定指定盘面上的具体磁道。

(3) 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。





可以看见，磁盘读取依靠的是机械运动，分为寻道时间、旋转延迟、传输时间三个部分，这三个部分耗时相加就是一次磁盘IO的时间，一般大概9ms左右。寻道时间（seek）是将读写磁头移动至正确的磁道上所需要的时间，这部分时间代价最高；旋转延迟时间（rotation）是磁盘旋转将目标扇区移动到读写磁头下方所需的时间，取决于磁盘转速；数据传输时间（transfer）是完成传输数据所需要的时间，取决于接口的数据传输率，在纳秒级，远小于前两部分消耗时间。磁盘读取时间成本是访问内存的几百倍到几万倍之间。

为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存，这个称之为**预读**。这样做的理论依据是计算机科学中著名的局部性原理：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

大家可以试运行下面这段代码：

```
public static void main(String[] args) {
    int[][] arr = new int[10000][10000];
    int sum = 0;
    long startTime = System.currentTimeMillis();
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[0].length; j++) {
            /*按行访问数组*/
            sum += arr[i][j];
        }
```

```
    }
    System.out.println("按行耗时：" + (System.currentTimeMillis() - startTime) + "ms");
    sum = 0;
    startTime = System.currentTimeMillis();
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr[0].length; j++) {
            /*按列访问数组*/
            sum += arr[j][i];
        }
    }
    System.out.println("按列耗时：" + (System.currentTimeMillis() - startTime) + "ms");
}
```

```
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
按行耗时：44ms
按列耗时：1578ms
```

就能看到局部性原理对程序性能的影响。

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），一般来说，磁盘的顺序读的效率是随机读的40到400倍都有可能，顺序写是随机写的10到100倍（SSD盘则差距要小的多，顺序读写的效率是随机读写的7到10倍，但是有评测表明机械硬盘的顺序写性能稍优于SSD。总的来说Mysql数据库如果由硬盘由机械的换成SSD的，性能会有很大的提升），因此对于具有局部性的程序来说，预读可以提高I/O效率。

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，页大小通常为4k当然也有16K的，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。

按照磁盘的这种性质，如果是一个页存放一个B+树的节点，自然是可以存放很多的数据的，比如InnoDB里，默认定义的B+树的节点大小是16KB，这就是说，假如一个Key是8个字节，那么一个节点可以存放大约1000个Key，意味着B+数可以有1000个分叉。同时InnoDB每一次磁盘I/O，读取的都是 16KB的整数倍的数据。也就是说InnoDB在节点的读写上是可以充分利用磁盘顺序IO的高速读写特性。

同时按照B+树逻辑结构来说，在叶子节点一层，所有记录的主键按照从小到大的顺序排列，并且形成了一个双向链表。同一层的非叶子节点也互相串联，形成了一个双向链表。那么在实际读写的时候，很大的概率相邻的节点会放在相邻的页上，又可以充分利用磁盘顺序IO的高速读写特性。所以我们对MySQL优化的一大方向就是**尽可能的多让数据顺序读写，少让数据随机读写**。