1. 数据库三范式（手绘ER图）

见文件：数据库设计

2. 事务四大特性（ACID）

见文件：数据库高级特性-五事务管理-事务的特性

3. 数据库隔离级别，每个级别会引发什么问题，mysql默认是哪个级别

见文件：数据库高级特性-五事务管理-事务的隔离级别/事务隔离性实现原理

4. MYSQL的两种存储引擎区别（事务、锁级别等等），各自的适用场景

(1) MyIsam ：不支持事务，不支持外键，所以访问速度快。锁机制是表锁，支持全文索引

(2) InnoDB ：支持事务、支持外键，所以对比MyISAM，InnoDB的处理效率差一些，并要占更多的磁盘空间保留数据和索引。锁机制是行锁，不支持全文索引。

(3) Memory：数据是存放在内存中的，默认哈希索引，非常适合存储临时数据，服务器关闭后，数据会丢失掉。

5. 数据库的优化（从sql语句优化和索引两个部分回答）

(1) 硬件优化，提高机器性能，增加硬件等;

(2) 优化SQL查询语句，将限定性强的where条件放前，用exists代替in操作等，SELECT子句中避免使用‘ \* ‘，用Where子句替换HAVING子句，通过内部函数提高SQL效率，用EXISTS替代IN、用NOT EXISTS替代NOT IN，用WHERE替代ORDER BY

(3) 优化索引，建立有效的索引并检查和修复缺少的统计信息等;

(4) 数据库系统文件优化，将数据文件、索引文件、日志文件放置在不同的磁盘上，提高并行度等。将大表分割成小表。

6. 索引的分类，最左前缀原则（组合索引）

见文件：MySQL性能优化-索引 3索引的类型

7. 索引失效的情况

(1) 如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)

注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引

(2) 对于多列索引，不是使用的第一部分，则不会使用索引（最左前缀原则）

(3) like查询是以%开头

(4) 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

(5) 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

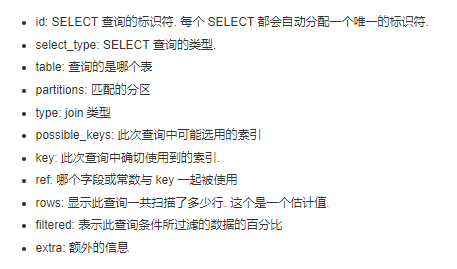
8. MySQL中 explain用法

(1) 含义：explain显示了mysql如何使用索引来处理select语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句。

用法：在 SELECT 语句前加上 Explain 就可以。

(2) 输出格式：https://segmentfault.com/a/1190000008131735





(3) type判断是全盘扫描还是索引扫描

const，system是主键索引。ref，表示最左前缀索引。range，表示索引范围查询。

9. MySQL的三种连接

(1) 内连接 inner join

(2) 外连接 left join right join full join

(3) 交叉连接（笛卡尔积）cross join

交叉联接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉联接也称作笛卡尔积。

10. 索引有B+索引和hash索引，各自的区别

见文件：MySQL性能优化-索引 5底层索引的实现

11. 聚集索引和非聚集索引区别。

(1) 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。

(2) 聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上连续，物理存储并不连续。

12. 关系型数据库和非关系型数据库区别

12.1 关系型数据库

(1) 特性：

① 关系型数据库，是指采用了关系模型来组织数据的数据库；

② 关系型数据库的最大特点就是事务的一致性；

③ 简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。

(2) 优点：

① 容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解；

② 使用方便：通用的SQL语言使得操作关系型数据库非常方便；

③ 易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率；

④ 支持SQL，可用于复杂的查询。

(3) 缺点：

① 为了维护一致性所付出的巨大代价就是其读写性能比较差；

② 固定的表结构，扩展性比较差；

③ 满足不了高并发读写需求；

④ 海量数据的读写效率很低；

12.2 非关系型数据库

(1) 特性：

① 使用键值对存储数据；

② 分布式；

③ 一般不支持ACID特性；

④ 非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

(2) 优点：

① 无需经过sql层的解析，读写性能很高；

② 基于键值对，数据没有耦合性，容易扩展；

③ 存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。

(3) 缺点：

① 不提供sql支持，学习和使用成本较高；

② 无事务处理，附加功能bi和报表等支持也不好；

13. B树、B-树、B+树、B\*树

14. 数据库慢查询

(1) 分析MySQL语句查询性能的方法除了使用 EXPLAIN 输出执行计划，还可以让MySQL记录下查询超过指定时间的语句，我们将超过指定时间的SQL语句查询称为“慢查询”。

(2) 开启慢查询方法：

show variables like 'slow\_query%'; //查询慢查询开启状态以及日志存放位置

show variables like 'long\_query\_time';//查询慢查询设置的时间参数

//设置参数

set global slow\_query\_log='ON';

set global slow\_query\_log\_file='/usr/local/mysql/data/slow.log';

set global long\_query\_time=1;

//修改配置文件my.cnf，在[mysqld]下的下方加入、

[mysqld]

slow\_query\_log = ON

slow\_query\_log\_file = /usr/local/mysql/data/slow.log

long\_query\_time = 1

//重启服务

service mysqld restart

15. MySQL的主从原理（提高数据库的容灾能力）

(1) 基本介绍

MySQL 内建的复制功能是构建大型，高性能应用程序的基础。将 MySQL 的 数亿分布到到多个系统上去，这种分步的机制，是通过将 MySQL 的某一台主机的数据复制到其它主机( Slave )上，并重新执行一遍来实现的。复制过程中一个服务器充当服务器，而一个或多个其它服务器充当从服务器。主服务器将更新写入二进制日志，并维护文件的一个索引以跟踪日志循环。这些日志可以记录发送到从服务器的更新。当一个从服务器连接主服务器时，它通知主服务器从服务器在日志中读取的最后一次成功更新的位置，从服务器接收从那时起发生的任何更新，然后封锁等等主服务器通知新的更新。

请注意当你进行复制时，所有对复制中的表的更新必须在主服务器上进行。否则，你必须要小心，以避免用户对主服务器上的表进行的更新与对服务器上的表所进行的更新之间的冲突

(2) MySQL支持的复制

① 基于语句的复制。 在主服务器上执行的 SQL 语句，在从服务器上执行同样的语句。否则，你必须要小心，以避免用户对主服务器上的表进行的更新与对服务器上的表所进行的更新之间的冲突，配置：binlog\_format = 'STATEMENT'

② 基于行的复制。把改变的内容复制过去，而不是把命令在从服务器上执行一遍，从 MySQL 5.0开始支持，配置：binlog\_format = 'ROW'

③ 混合类型的复制。默认采用基于语句的复制，一旦发现基于语句的无法精确的复制时，就会采用基于行的复制,配置：binlog\_format = 'MIXED'

(3) MySQL复制解决的问题

数据分布

负载平衡

备份

高可用性和容错行

16. 数据库的锁机制

(1) 基本介绍

数据库管理系统（DBMS）中的并发控制的任务是确保在多个事务同时存取数据库中同一数据时不破坏事务的隔离性和统一性以及数据库的统一性。封锁、时间戳、乐观并发控制(乐观锁)和悲观并发控制（悲观锁）是并发控制主要采用的技术手段。

(2) 锁的分类

① 按操作划分，可分为DML锁（数据锁用来保证数据完整性）、DDL锁（数据字典锁用来保护数据库对象的结构）

② 按锁的粒度划分，可分为表级锁、行级锁、页级锁（mysql）

③ 按锁级别划分，可分为共享锁、排他锁

④ 按加锁方式划分，可分为自动锁、显示锁eeeeeee

⑤ 按使用方式划分，可分为乐观锁、悲观锁

16.1 锁的粒度（行级、表级、页级）

(1) 行级锁

行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。行级锁分为共享锁和排他锁。

① 特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

(2) 表级锁（无死锁）

表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MYISAM与INNODB都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁（排他锁）。

① 特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

(3) 页级锁

页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB支持页级锁

① 特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

16.2 MySQL常用存储引擎的锁机制

(1) MyISAM和MEMORY采用表级锁(table-level locking)

(2) BDB采用页面锁(page-level locking)或表级锁，默认为页面锁

(3) InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

16.3 Innodb中的行锁与表锁

(1) InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

(2) 行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。行级锁的缺点是：由于需要请求大量的锁资源，所以速度慢，内存消耗大。

16.4 行级锁和死锁

(1) MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。而在InnoDB中，锁是逐步获得的，就造成了死锁的可能。

(2) 在MySQL中，行级锁并不是直接锁记录，而是锁索引。索引分为主键索引和非主键索引两种，如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。 在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。

(3) 当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。

(4) 发生死锁后，InnoDB一般都可以检测到，并使一个事务释放锁回退，另一个获取锁完成事务。show engine innodb status 打印死锁记录

16.5 避免死锁

(1) 如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

(2) 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

(3) 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

16.6 共享锁和排他锁

(1) 共享锁（行级锁）

共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。获准共享锁的事务只能读数据，不能修改数据。

① 用法：SELECT ... LOCK IN SHARE MODE;

在查询语句后面增加LOCK IN SHARE MODE，Mysql会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

(2) 排他锁（行级锁）

排他锁又称写锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的封锁。获准排他锁的事务既能读数据，又能修改数据。

① 用法：SELECT ... FOR UPDATE;

在查询语句后面增加FOR UPDATE，Mysql会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

(3) 意向锁（表级锁）

① 意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁。

② 意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

③ 对于insert、update、delete，InnoDB会自动给涉及的数据加排他锁（X）；对于一般的Select语句，InnoDB不会加任何锁，事务可以通过以下语句给显示加共享锁或排他锁。

共享锁：SELECT ... LOCK IN SHARE MODE;

排他锁：SELECT ... FOR UPDATE;、

16.7 悲观锁

在关系数据库管理系统里，悲观并发控制（又名“悲观锁”，Pessimistic Concurrency Control，缩写“PCC”）是一种并发控制的方法。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作的某行数据应用了锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

(1) 悲观锁流程：

① 在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。

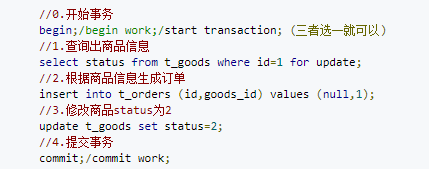
② 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。

③ 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。

④ 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常

(2) MySQL InnoDB中使用悲观锁

要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。set autocommit=0;



① MySQL InnoDB默认行级锁。行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁把整张表锁住，这点需要注意。

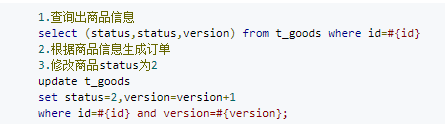
② 悲观并发控制实际上是“先取锁再访问”的保守策略，为数据处理的安全提供了保证。但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会；另外，在只读型事务处理中由于不会产生冲突，也没必要使用锁，这样做只能增加系统负载；还有会降低了并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数。

16.8 乐观锁

(1) 乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

(2) 数据版本,为数据增加的一个版本标识。当读取数据时，将版本标识的值一同读出，数据每更新一次，同时对版本标识进行更新。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的版本标识进行比对，如果数据库表当前版本号与第一次取出来的版本标识值相等，则予以更新，否则认为是过期数据。实现数据版本有两种方式，第一种是使用版本号，第二种是使用时间戳。

(3) 使用版本号时，可以在数据初始化时指定一个版本号，每次对数据的更新操作都对版本号执行+1操作。并判断当前版本号是不是该数据的最新的版本号。



(4) 优点和不足

乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

17. MVCC机制

(1) 定义：MVCC是一种多版本并发控制机制。

(2) 实现：

MVCC是通过保存数据在某个时间点的快照来实现的. 不同存储引擎的MVCC. 不同存储引擎的MVCC实现是不同的,典型的有乐观并发控制和悲观并发控制.

17.1 作用：

(1) 大多数的MYSQL事务型存储引擎,如,InnoDB，Falcon以及PBXT都不使用一种简单的行锁机制.事实上,他们都和MVCC–多版本并发控制来一起使用.

(2) 锁机制可以控制并发操作,但是其系统开销较大,而MVCC可以在大多数情况下代替行级锁,使用MVCC,能降低其系统开销.

17.2 具体实现

InnoDB的MVCC,是通过在每行记录后面保存两个隐藏的列来实现的,这两个列，分别保存了这个行的创建时间，一个保存的是行的删除时间。这里存储的并不是实际的时间值,而是系统版本号(可以理解为事务的ID)，没开始一个新的事务，系统版本号就会自动递增，事务开始时刻的系统版本号会作为事务的ID.

（在REPEATABLE READ可重复读隔离级别下,MVCC具体的操作.）

(1) INSERT - InnoDB为新插入的每一行保存当前系统版本号作为版本号.

start transaction;

insert into yang values(NULL,'yang') ;

insert into yang values(NULL,'long');

insert into yang values(NULL,'fei');

commit; //插入语句



(2) SELECT

InnoDB会根据以下两个条件检查每行记录:

① .InnoDB只会查找版本早于当前事务版本的数据行(也就是,行的系统版本号小于或等于事务的系统版本号)，这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的.。就是修改之前的。

② 行的删除版本要么未定义，要么大于当前事务版本号（就是说删除版本是在当前版本之后的），这可以确保事务读取到的行，在事务开始之前未被删除.。

只有两条同时满足的记录，才能返回作为查询结果。

(3) DELETE

InnoDB会为删除的每一行保存当前系统的版本号(事务的ID)作为删除标识。

(4) UPDATE

InnoDB执行UPDATE，实际上是新插入了一行记录，并保存其创建时间为当前事务的ID，同时保存当前事务ID到要UPDATE的行的删除时间。（就是创建版本和删除版本都为当前的事务ID）

18. 平衡二叉树、B树、B+树、B\*树

18.1 平衡二叉树

(1) 定义：平衡二叉树是基于二分法的策略提高数据的查找速度的二叉树的数据结构。

(2) 特点：

平衡二叉树是采用二分法思维把数据按规则组装成一个树形结构的数据，用这个树形结构的数据减少无关数据的检索，大大的提升了数据检索的速度；平衡二叉树的数据结构组装过程有以下规则：

① 非叶子节点只能允许最多两个子节点存在，每一个非叶子节点数据分布规则为左边的子节点小当前节点的值，右边的子节点大于当前节点的值(这里值是基于自己的算法规则而定的，比如hash值)；（二叉搜索树）

② 平衡树的层级结构：因为平衡二叉树查询性能和树的层级（h高度）成正比、为了保证树的结构左右两端数据大致平衡降低二叉树的查询难度一般会采用一种算法机制实现节点数据结构的平衡，实现了这种算法的有比如AVL、Treap、红黑树，使用平衡二叉树能保证数据的左右两边的节点层级相差不会大于1.，通过这样避免树形结构由于删除增加变成线性链表影响查询效率，保证数据平衡的情况下查找数据的速度近于二分法查找；

(3) 小结：

① 非叶子节点最多拥有两个子节点；

② 非叶子节值大于左边子节点、小于右边子节点；（二叉搜索树）

③ 树的左右两边的层级数相差不会大于1;（平衡二叉树）

④ 没有值相等重复的节点;

18.2 B树(B-tree)（为了优化磁盘存储）

(1) 定义：

B树和平衡二叉树稍有不同的是B树属于多叉树又名平衡多路查找树（查找路径不只两个），数据库索引技术里大量使用者B树和B+树的数据结构。

(2) 规则

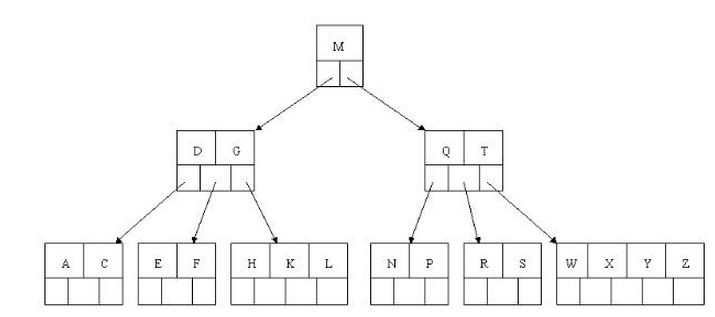
① 树种的每个节点最多拥有m个子节点且m>=2,空树除外（注：m阶代表一个树节点最多有多少个查找路径，m阶=m路,当m=2则是2叉树,m=3则是3叉）；

② 除根节点外每个节点的关键字数量大于等于ceil(m/2)-1个小于等于m-1个，非根节点关键字数必须>=2;（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2)

③ 所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应下图最后一层节点的空格子

④ 如果一个非叶节点有N个子节点，则该节点的关键字数等于N-1;

⑤ 所有节点关键字是按递增次序排列，并遵循左小右大原则；



(3) B树的查询流程： 如上图我要从上图中找到E字母，查找流程如下

① 获取根节点的关键字进行比较，当前根节点关键字为M，E要小于M（26个字母顺序），所以往找到指向左边的子节点（二分法规则，左小右大，左边放小于当前节点值的子节点、右边放大于当前节点值的子节点）；

② 拿到关键字D和G，D<E<G 所以直接找到D和G中间的节点；

③ 拿到E和F，因为E=E 所以直接返回关键字和指针信息（如果树结构里面没有包含所要查找的节点则返回null）；

(4) 特点：

B树相对于平衡二叉树的不同是，每个节点包含的关键字增多了，特别是在B树应用到数据库中的时候，数据库充分利用了磁盘块的原理（磁盘数据存储是采用块的形式存储的，每个块的大小为4K，每次IO进行数据读取时，同一个磁盘块的数据可以一次性读取出来）把节点大小限制和充分使用在磁盘快大小范围；把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

18.3 B+树（只有到叶子节点才能查询到数据）

(1) 定义：B+树是B树的一个升级版，相对于B树来说B+树更充分的利用了节点的空间，让查询速度更加稳定，其速度完全接近于二分法查找。

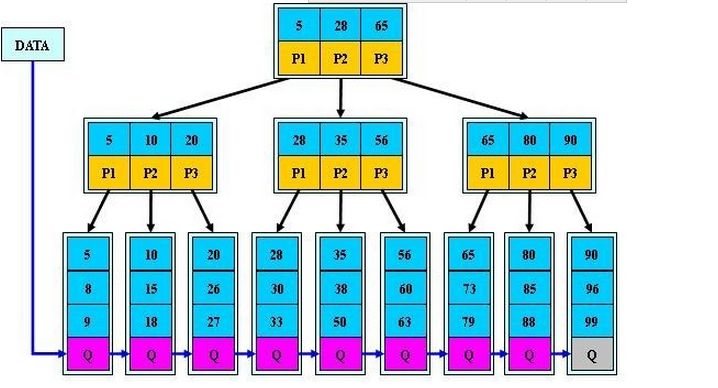
(2) B树和B+树的区别

① B+跟B树不同。B+树的非叶子节点不保存关键字记录的指针，这样使得B+树每个节点所能保存的关键字大大增加；

② B+树叶子节点保存了父节点的所有关键字和关键字记录的指针，每个叶子节点的关键字从小到大链接；

③ B+树的根节点关键字数量和其子节点个数相等;

④ B+的非叶子节点只进行数据索引，不会存实际的关键字记录的指针，所有数据地址必须要到叶子节点才能获取到，所以每次数据查询的次数都一样；



(3) 特点：

在B树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定。

18.4 B\*树（B\*树是B+树的变种）

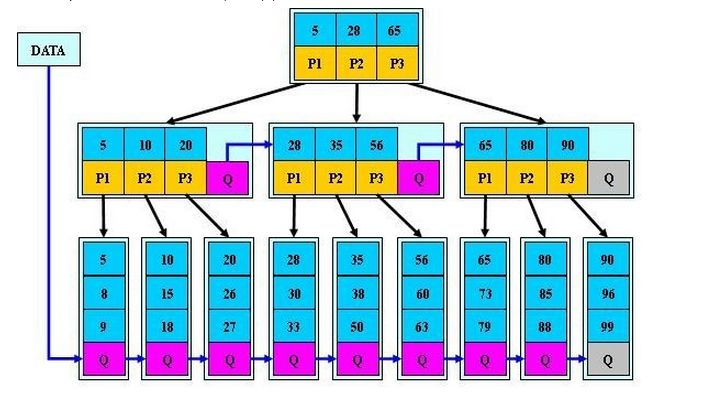
(1) B+和B\*之间的区别

① 首先是关键字个数限制问题，B+树初始化的关键字初始化个数是cei(m/2)，b\*树的初始化个数为（cei(2/3\*m)）

② B+树节点满时就会分裂，而B\*树节点满时会检查兄弟节点是否满（因为每个节点都有指向兄弟的指针），如果兄弟节点未满则向兄弟节点转移关键字，如果兄弟节点已满，则从当前节点和兄弟节点各拿出1/3的数据创建一个新的节点出来；

(3) 特点：

在B+树的基础上因其初始化的容量变大，使得节点空间使用率更高，而又存有兄弟节点的指针，可以向兄弟节点转移关键字的特性使得B\*树额分解次数变得更少；



18.5 总结

(1) 从平衡二叉树、B树、B+树、B\*树总体来看它们的贯彻的思想是相同的，都是采用二分法和数据平衡策略来提升查找数据的速度；

(2) 不同点是他们一个一个在演变的过程中通过IO从磁盘读取数据的原理进行一步步的演变，每一次演变都是为了让节点的空间更合理的运用起来，从而使树的层级减少达到快速查找数据的目的；

19. 怎么防止SQL 的注入

(1) 采用预编译语句集，它内置了处理SQL注入的能力，只要使用它的setXXX方法传值即可。

(2) jsp中调用该函数检查是否包函非法字符

(3) 采用正则表达式过滤传入的参数

20. 大数据的分库分表

20.1 垂直拆分：指按功能模块拆分，比如分成订单表，用户表。

20.2 水平拆分：将数据进行拆分保存到不同的表当中，这些表的结构完全相同。

(1) 顺序拆分：例如订单表可以按订单的日期按年份才分，容易造成数据不均衡。

(2) hash取模拆分：数据分布均匀。

21. 数据库的高并发访问

(1) 集群

(2) 主从复制