# 数据库理论

# OLTP与OLAP

# 8.1 OLTP（关系型数据库)

* OLTP即联机事务处理，就是我们经常说的关系数据库，意即记录即时的增、删、改、查，就是我们经常应用的东西，这是数据库的基础
* 对于各种数据库系统环境中大家最常见的OLTP系统，其特点是并发量大，整体数据量比较多，但每次访问的数据比较少，且访问的数据比较离散，活跃数据占总体数据的比例不是太大。对于这类系统的数据库实际上是最难维护，最难以优化的，对主机整体性能要求也是最高的。因为不仅访问量很高，数据量也不小。
* 针对上面的这些特点和分析，我们可以对OLTP的得出一个大致的方向。
* 虽然系统总体数据量较大，但是系统活跃数据在数据总量中所占的比例不大，那么我们可以通过扩大内存容量来尽可能多的将活跃数据cache到内存中；
* 虽然IO访问非常频繁，但是每次访问的数据量较少且很离散，那么我们对磁盘存储的要求是IOPS(Input/Output Operations Per Second，即每秒进行读写操作的次数)表现要很好，吞吐量是次要因素；
* 并发量很高，CPU每秒所要处理的请求自然也就很多，所以CPU处理能力需要比较强劲；
* 虽然与客户端的每次交互的数据量并不是特别大，但是网络交互非常频繁，所以主机与客户端交互的网络设备对流量能力也要求不能太弱。

# 8.2 OLAP（数据分析挖掘)

* 用于数据分析的OLAP系统的主要特点就是数据量非常大，并发访问不多，但每次访问所需要检索的数据量都比较多，而且数据访问相对较为集中，没有太明显的活跃数据概念。
* OLAP即联机分析处理，是数据仓库的核心部心，所谓数据仓库是对于大量已经由OLTP形成的数据的一种分析型的数据库，用于处理商业智能、决策支持等重要的决策信息；数据仓库是在数据库应用到一定程序之后而对历史数据的加工与分析
* 基于OLAP系统的各种特点和相应的分析，针对OLAP系统硬件优化的大致策略如下：
* 数据量非常大，所以磁盘存储系统的单位容量需要尽量大一些；
* 单次访问数据量较大，而且访问数据比较集中，那么对IO系统的性能要求是需要有尽可能大的每秒IO吞吐量，所以应该选用每秒吞吐量尽可能大的磁盘；
* 虽然IO性能要求也比较高，但是并发请求较少，所以CPU处理能力较难成为性能瓶颈，所以CPU处理能力没有太苛刻的要求；
* 虽然每次请求的访问量很大，但是执行过程中的数据大都不会返回给客户端，最终返回给客户端的数据量都较小，所以和客户端交互的网络设备要求并不是太高；
* 此外，由于OLAP系统由于其每次运算过程较长，可以很好的并行化，所以一般的OLAP系统都是由多台主机构成的一个集群，而集群中主机与主机之间的数据交互量一般来说都是非常大的，所以在集群中主机之间的网络设备要求很高。

# 数据库完整性

# 8.3 实体完整性

* primary key （列级约束和表级约束)
* 定义主码之后，每当用户程序对基本表插入一条记录或对主码列进行更新操作时，DBMS将会检查
  + 1)检查主码值是否唯一：一种方法是全表扫描，耗时长；DBMS一般在主码上自动建立一个索引，通过索引查找基本表中是否已经存在新的主码值将大大提供效率。
  + 2)检查主码值是否为空

# 8.4 参照完整性

* foreign key references ...
* 当上述的不一致发生时，系统可以采用以下策略加以处理：
  + 1)拒绝执行（No action)
  + 2)级联操作（Cascade)：当删除或修改被参照表时的一个元组导致与参照表不同时，删除或修改参照表中的所有导致不一致的元组。
  + 3)设置为空值：当删除或修改被参照表时的一个元组导致与参照表不同时，则将参照表中
* 的所有造成不一致的元组的对应属性设置为空值。
* 关系查询处理和查询优化
* 关系数据库系统的查询处理
* 查询处理是RDBMS执行查询语句的过程，其任务是把用户提交给RDBMS的查询语句转换为高效的查询执行计划。
* 查询处理步骤
* 查询处理分为4个阶段：查询分析、查询检查、查询优化和查询执行。
* 1、查询分析（语法)
* 对查询语句进行扫描、词法分析、语法分析
* 2、查询检查（语义)
* 对合法的查询语句进行语义检查，即根据数据字典中有关的模式定义检查语句中的数据库对象，如关系名、属性名等是否存在和有效。然后进行安全性、完整性检查。检查通过后把SQL语句转换成等价的关系代数表达式。RDBMS一般采用查询树（语法树)来表示拓展的关系代数表达式。
* 3、查询优化
* 查询优化就是选择一个高效执行的查询处理策略。
* 分为代数优化和物理优化
  + 1)代数优化是指关系代数表达式的优化，即按照一定的规则，通过对关系代数表达式进行等价变换，改变代数表达式中操作的次序和组合，使查询执行更高效
  + 2)物理优化是指通过存取路径和底层操作算法的选择进行的优化
* 选择的依据可以是基于规则的、基于代价的、基于语义的。
* 4、查询执行
* 依据优化器得到的执行策略生成查询执行计划，由代码生成器生成执行这个查询计划的代码，然后加以执行，回送查询结果。
* 实现查询操作的算法示例

### 1、选择操作的实现

- （1) 全表扫描方法 (Table Scan)

* 对查询的基本表顺序扫描，逐一检查每个元组是否满足选择条件，把满足条件的元组作为结果输出
* 适合小表，不适合大表
  + （2)索引扫描方法 (Index Scan)
* 适合于选择条件中的属性上有索引(例如B+树索引或Hash索引)
* 通过索引先找到满足条件的元组主码或元组指针，再通过元组指针直接在查询的基本表中找到元组
  + 1)全表扫描算法
* 假设可以使用的内存为M块，全表扫描算法思想：  
  ①- 按照物理次序读Student的M块到内存  
  ②- 检查内存的每个元组t，如果满足选择条件，则输出t  
  ③- 如果student还有其他块未被处理，重复①和②  
  1）- 索引扫描算法
* [例9.1-C2] SELECT \* FROM Student WHERE Sno='201215121'  
  - 假设Sno上有索引(或Sno是散列码)  
  - 算法：  
  - 使用索引(或散列)得到Sno为‘201215121’ 元组的指针  
  - 通过元组指针在Student表中检索到该学生
* [例9.1-C3] SELECT \* FROM Student WHERE Sage>20  
  - 假设Sage 上有B+树索引  
  - 算法：  
  - 使用B+树索引找到Sage=20的索引项，以此为入口点在B+树的顺序集上得到Sage>20的所有元组指针  
  - 通过这些元组指针到student表中检索到所有年龄大于20的学生。
* [例9.1-C4] SELECT \* FROM Student WHERE Sdept='CS' AND Sage>20;  
  - 假设Sdept和Sage上都有索引  
  - 算法一：分别用索引扫描找到Sdept＝’CS’的一组元组指针和Sage>20的另一组元组指针  
  - 求这两组指针的交集  
  - 到Student表中检索  
  - 得到计算机系年龄大于20的学生  
  - 算法二：找到Sdept=’CS’的一组元组指针，  
  - 通过这些元组指针到Student表中检索  
  - 并对得到的元组检查另一些选择条件(如Sage>20)是否满足  
  - 把满足条件的元组作为结果输出。
* 当选择率较低时，基于索引的选择算法要优于全表扫描算法。但在某些情况下，例如选择率较高，或者要查找的元组均匀地分布在查找的表中，这是基于索引的选择算法性能不如全表扫描算法。因此除了对表的扫描操作，还要加上对B+树索引的扫描操作，对每一个检索码，从B+树根节点到叶子结点路径上的每个结点都要进行一次IO操作。

### 2、连接操作的实现

* 连接操作是查询处理中最耗时的操作之一
* 本节只讨论等值连接(或自然连接)最常用的实现算法
* [例9.2] SELECT \* FROM Student, SC WHERE Student.Sno=SC.Sno;

#### 1)嵌套循环算法(nested loop join)

- 对外层循环(Student表)的每一个元组(s)，检索内层循环(SC表)中的每一个元组(sc)  
- 检查这两个元组在连接属性(Sno)上是否相等  
- 如果满足连接条件，则串接后作为结果输出，直到外层循环表中的元组处理完为止。

#### 2)排序-合并算法(sort-merge join 或merge join)

- 如果连接的表没有排好序，先对Student表和SC表按连接属性Sno排序   
- 取Student表中第一个Sno，依次扫描SC表中具有相同Sno的元组   
- 当扫描到Sno不相同的第一个SC元组时，返回Student表扫描它的下一个元组，再扫描SC表中具有相同Sno的元组，把它们连接起来   
- 重复上述步骤直到Student 表扫描完

* Student表和SC表都只要扫描一遍
* 如果两个表原来无序，执行时间要加上对两个表的排序时间
* 对于大表，先排序后使用排序-合并连接算法执行连接，总的时间一般仍会减少

#### 3)索引连接(index join)算法

* 步骤：
* ① 在SC表上已经建立属性Sno的索引。
* ② 对Student中每一个元组，由Sno值通过SC的索引查找相应的SC元组。
* ③ 把这些SC元组和Student元组连接起来
* 循环执行②③，直到Student表中的元组处理完为止
* 只有表2需要索引

#### 4)Hash Join算法

- 把连接属性作为hash码，用同一个hash函数把Student表和SC表中的元组散列到hash表中。  
- 划分阶段（Build)  
- 对包含较少元组的表(如Student表)进行一遍处理  
- 把它的元组按hash函数分散到hash表的桶中  
- 试探阶段（Probe)  
- 对另一个表(SC表)进行一遍处理  
- 把SC表的元组也按同一个hash函数（hash码是连接属性)进行散列  
- 把SC元组与桶中来自Student表并与之相匹配的元组连接起来

* 将小表转为哈希表，用表1的匹配字段用哈希函数映射到哈希表
* 上面hash join算法前提：假设两个表中较小的表在第一阶段后可以完全放入内存的hash桶中
* 关系数据库系统的查询优化
* 查询优化在关系数据库系统中有着非常重要的地位
* 关系查询优化是影响关系数据库管理系统性能的关键因素
* 由于关系表达式的语义级别很高，使关系系统可以从关系表达式中分析查询语义，提供了执行查询优化的可能性
* 查询优化概述
* 关系系统的查询优化
* 是关系数据库管理系统实现的关键技术又是关系系统的优点所在
* 减轻了用户选择存取路径的负担
* 非关系系统
* 用户使用过程化的语言表达查询要求，执行何种记录级的操作，以及操作的序列是由用户来决定的
* 用户必须了解存取路径，系统要提供用户选择存取路径的手段，查询效率由用户的存取策略决定
* 如果用户做了不当的选择，系统是无法对此加以改进的
* 查询优化的优点
* 1、用户不必考虑如何最好地表达查询以获得较好的效率
* 2、系统可以比用户程序的“优化”做得更好
  + （1) 优化器可以从数据字典中获取许多统计信息，而用户程序则难以获得这些信息。
  + （2)如果数据库的物理统计信息改变了，系统可以自动对查询重新优化以选择相适应的执行计划。在非关系系统中必须重写程序，而重写程序在实际应用中往往是不太可能的。
  + （3)优化器可以考虑数百种不同的执行计划，程序员一般只能考虑有限的几种可能性。
  + （4)优化器中包括了很多复杂的优化技术，这些优化技术往往只有最好的程序员才能掌握。系统的自动优化相当于使得所有人都拥有这些优化技术。
* 关系数据库管理系统通过某种代价模型计算出各种查询执行策略的执行代价，然后选取代价最小的执行方案
* 查询优化的总目标
* 选择有效的策略
* 求得给定关系表达式的值
* 使得查询代价最小(实际上是较小)
* 一个实例
* 一个关系查询可以对应不同的执行方案，其效率可能相差非常大。
* [例9.3] 求选修了2号课程的学生姓名。
* SELECT Student.Sname FROM Student, SC
* WHERE Student.Sno=SC.Sno AND SC.Cno=’2’
* 假定学生-课程数据库中有1000个学生记录，10000个选课记录
* 选修2号课程的选课记录为50个
* 第一种情况：
* 1、计算笛卡尔积
* 算法：
  + 1)在内存中尽可能多地装入某个表(如Student表)的若干块，留出一块存放另一个表(如SC表)的元组。
  + 2)把SC中的每个元组和Student中每个元组连接，连接后的元组装满一块后就写到中间文件上
  + 3)从SC中读入一块和内存中的Student元组连接，直到SC表处理完。
  + 4)再读入若干块Student元组，读入一块SC元组
  + 5)重复上述处理过程，直到把Student表处理完
* 2、作选择操作
* 依次读入连接后的元组，按照选择条件选取满足要求的记录
* 假定内存处理时间忽略。读取中间文件花费的时间(同写中间文件一样)需读入106块。
* 若满足条件的元组假设仅50个，均可放在内存。
* 3、作投影操作
* 把第2步的结果在Sname上作投影输出，得到最终结果
* 第一种情况下执行查询的总读写数据块
* 第二种情况：
* 1、计算自然连接
  + 1)执行自然连接，读取Student和SC表的策略不变，总的读取块数仍为2100块
  + 2)自然连接的结果比第一种情况大大减少，为104个元组
  + 3)写出数据块= 103 块
* 2、读取中间文件块，执行选择运算，读取的数据块= 103 块
* 3、把第2步结果投影输出。
* 第二种情况下执行查询的总读写数据块=2100+ 103 +103
* 其执行代价大约是第一种情况的488分之一
* 第三种情况：
* 1、先对SC表作选择运算，只需读一遍SC表，存取100块，因为满足条件的元组仅50个，不必使用中间文件。
* 2、读取Student表，把读入的Student元组和内存中的SC元组作连接。也只需读一遍Student表共100块。
* 3、把连接结果投影输出
* 第三种情况总的读写数据块=100+100
* 其执行代价大约是第一种情况的万分之一，是第二种情况的20分之一   
  - 假如SC表的Cno字段上有索引  
  - 第一步就不必读取所有的SC元组而只需读取Cno=‘2’的那些元组(50个)  
  - 存取的索引块和SC中满足条件的数据块大约总共3～4块  
  - 若Student表在Sno上也有索引  
  - 不必读取所有的Student元组  
  - 因为满足条件的SC记录仅50个，涉及最多50个Student记录  
  - 读取Student表的块数也可大大减少
* 有选择和连接操作时，先做选择操作，这样参加连接的元组就可以大大减少，这是代数优化
* 在Q3中SC表的选择操作算法有全表扫描或索引扫描，经过初步估算，索引扫描方法较优。
* 对于Student和SC表的连接，利用Student表上的索引，采用索引连接代价也较小，这就是物理优化。
* 代数优化
* 关系代数表达式等价变换规则
* 代数优化策略：通过对关系代数表达式的等价变换来提高查询效率
* 关系代数表达式的等价：指用相同的关系代替两个表达式中相应的关系所得到的结果是相同的
* 两个关系表达式E1和E2是等价的，可记为E1≡E2
* 常用的代数变换规则：
* 1.连接、笛卡尔积交换律
  1. 连接、笛卡尔积的结合律
* 3.投影的串接定律
* 减少IO次数
  1. 选择的串接定律
* 合并条件
  1. 选择与投影操作的交换律
* 先选择后投影效率更高
  1. 选择与笛卡尔积的交换律
* 减少IO次数
  1. 选择与并的分配律
  2. 选择与差运算的分配律
  3. 选择对自然连接的分配律
  4. 投影与笛卡尔积的分配律
  5. 投影与并的分配律
* 查询树的启发式优化
* 典型的启发式规则
  + （1)选择运算应尽可能先做
* 在优化策略中这是最重要、最基本的一条。
  + （2)把投影运算和选择运算同时进行
* 如有若干投影和选择运算，并且它们都对同一个关系操作，则可以在扫描此关系的同时完成所有的这些运算以避免重复扫描关系。
  + （3) 把投影同其前或其后的双目运算结合起来，没有必要为了去掉某些字段而扫描一遍关系。
  + （4) 把某些选择同在它前面要执行的笛卡尔积结合起来成为一个连接运算，连接特别是等值连接运算要比同样关系上的笛卡尔积省很多时间。
  + （5) 找出公共子表达式
* 如果这种重复出现的子表达式的结果不是很大的关系，并且从外存中读入这个关系比计算该子表达式的时间少得多
* 则先计算一次公共子表达式并把结果写入中间文件是合算的。
* 当查询的是视图时，定义视图的表达式就是公共子表达式的情况
* 遵循这些启发式规则，应用9.3.1的等价变换公式来优化关系表达式的算法。
* 算法：关系表达式的优化
* 输入：一个关系表达式的查询树
* 输出：优化的查询树
* 方法：
  + （1)利用等价变换规则4把形如σF1∧F2∧…∧Fn(E)变换为
* σF1(σF2(…(σFn(E))…))。（分开选择条件)
  + （2)对每一个选择，利用等价变换规则4～9尽可能把它移到树的叶端。
* （选择先做)
  + （3)对每一个投影利用等价变换规则3，5，10，11中的一般形式尽可能把它移向树的叶端。
* 注意：
* 等价变换规则3使一些投影消失或使一些投影出现
* 规则5把一个投影分裂为两个，其中一个有可能被移向树的叶端
  + （4)利用等价变换规则3～5，把选择和投影的串接合并成单个选择、单个投影或一个选择后跟一个投影，使多个选择或投影能同时执行，或在一次扫描中全部完成
* （选择投影一起做)
  + （5)把上述得到的语法树的内节点分组。
* 每一双目运算()和它所有的直接祖先为一组(这些直接祖先是(σ，π运算)。
* 如果其后代直到叶子全是单目运算，则也将它们并入该组
* 但当双目运算是笛卡尔积(×)，而且后面不是与它组成等值连接的选择时，则不能把选择与这个双目运算组成同一组
* 示例：
  + 1)把SQL语句转换成查询树
* 为了使用关系代数表达式的优化法，假设内部表示是关
* 系代数语法树，则上面的查询树如图
  + 2)对查询树进行优化
* 利用规则4、6把选择σSC.Cno=‘2’移到叶端，图9.4查询树便转换成下图优化的查询树。这就是9.2.2节中Q3的查询树表示。
* 物理优化
* 代数优化改变查询语句中操作的次序和组合，不涉及底层的存取路径
* 对于一个查询语句有许多存取方案，它们的执行效率不同， 仅仅进行代数优化是不够的
* 物理优化就是要选择高效合理的操作算法或存取路径，求得优化的查询计划
* 物理优化方法
  + 1)基于规则的启发式优化
* 启发式规则是指那些在大多数情况下都适用，但不是在每种情况下都是最好的规则。
  + 2)基于代价估算的优化
* 优化器估算不同执行策略的代价，并选出具有最小代价的执行计划。
  + 3)两者结合的优化方法：
* 常常先使用启发式规则，选取若干较优的候选方案，减少代价估算的工作量
* 然后分别计算这些候选方案的执行代价，较快地选出最终的优化方案
* 基于启发式规则的存取路径选择优化(定性)
* 1.选择操作的启发式规则
* 对于小关系，使用全表顺序扫描，即使选择列上有索引
* 对于大关系，启发式规则有：
  + 1)对于选择条件是“主码＝值”的查询
* 查询结果最多是一个元组，可以选择主码索引
* 一般的RDBMS会自动建立主码索引
  + 2)对于选择条件是“非主属性＝值”的查询，并且选择列上有索引
* 要估算查询结果的元组数目
* 如果比例较小(<10%)可以使用索引扫描方法，否则还是使用全表顺序扫描
  + 3)对于选择条件是属性上的非等值查询或者范围查询，并且选择列上有索引
* 要估算查询结果的元组数目
* 如果比例较小(<10%)可以使用索引扫描方法，否则还是使用全表顺序扫描
  + 4)对于用AND连接的合取选择条件
* 如果有涉及这些属性的组合索引，优先采用组合索引扫描方法
* 如果某些属性上有一般的索引，可以用索引扫描方法
* 通过分别查找满足每个条件的指针，求指针的交集
* 通过索引查找满足部分条件的元组，然后在扫描这些元组时判断是否满足剩余条件
* 其他情况：使用全表顺序扫描
  + 5)对于用OR连接的析取选择条件，一般使用全表顺序扫描
* 2.连接操作的启发式规则
  + 1)如果2个表都已经按照连接属性排序：选用排序-合并算法
  + 2)如果一个表在连接属性上有索引，选用索引连接算法
  + 3)如果上面2个规则都不适用，其中一个表较小，选用Hash join算法
  + 4)可以选用嵌套循环方法，并选择其中较小的表，确切地讲是占用的块数(b)较少的表，作为外表(外循环的表)
* 理由：
* 设连接表R与S分别占用的块数为Br与Bs
* 连接操作使用的内存缓冲区块数为K
* 分配K-1块给外表
  + 如果R为外表，则嵌套循环法存取的块数为Br+BrBs/(K-1)
* 显然应该选块数小的表作为外表
* 基于代价估算的优化(定量)
* 启发式规则优化是定性的选择，适合解释执行的系统。因为解释执行的系统，优化开销包含在查询总开销之中。
* 编译执行的系统中查询优化和查询执行是分开的，因此可以采用精细复杂一些的基于代价的优化方法
* 1.统计信息
* 基于代价的优化方法要计算查询的各种不同执行方案的执行代价，它与数据库的状态密切相关
* 优化器需要的统计信息
  + （1)对每个基本表  
    ①- 该表的元组总数(N)  
    ②- 元组长度(l)  
    ③- 占用的块数(B)  
    ④- 占用的溢出块数(BO)
  + （2)对基表的每个列  
    ①- 该列不同值的个数(m)  
    ②- 列最大值  
    ③- 最小值  
    ④- 列上是否已经建立了索引  
    ⑤- 哪种索引(B+树索引、Hash索引、聚集索引)  
    ⑥- 可以计算选择率(f)
* 如果不同值的分布是均匀的，f＝1/m
* 如果不同值的分布不均匀，则要计算每个值的选择率，
* f＝具有该值的元组数/N
  + （3)对索引  
    ①- 索引的层数(L)  
    ②- 不同索引值的个数  
    ③- 索引的选择基数S(有S个元组具有某个索引值)  
    ④- 索引的叶结点数(Y)
* 2.代价估算示例
  + 1)全表扫描算法的代价估算公式
* 如果基本表大小为B块，全表扫描算法的代价 cost＝B
* 如果选择条件是“码＝值”，那么平均搜索代价 cost＝B/2 （可能是第一块也可能是最后一块)
  + 2)索引扫描算法的代价估算公式
* 如果选择条件是“码＝值”，则采用该表的主索引
* 若为B+树，层数为L，需要存取B+树中从根结点到叶结点L块，再加上基本表中该元组所在的那一块，所以cost=L+1
* 如果选择条件涉及非码属性
* 若为B+树索引，选择条件是相等比较，S是索引的选择基数(有S个元组满足条件)
* 满足条件的元组可能会保存在不同的块上，所以(最坏的情况)cost=L+S
* 如果比较条件是＞，＞＝，＜，＜＝操作
* 假设有一半的元组满足条件，就要存取一半的叶结点
* 通过索引访问一半的表存储块
* cost=L+Y/2+B/2
* 如果可以获得更准确的选择基数，可以进一步修正Y/2与B/2
  + 3)嵌套循环连接算法的代价估算公式
  + 嵌套循环连接算法的代价 cost＝Br+BrBs/(K-1)
  + 如果需要把连接结果写回磁盘 cost＝Br+Br Bs/(K-1)+(Frs*Nr*Ns)/Mrs
* 其中Frs为连接选择性(join selectivity)，表示连接结果元组数的比例
* Mrs是存放连接结果的块因子，表示每块中可以存放的结果元组数目
  + 4)排序-合并连接算法的代价估算公式
* 如果连接表已经按照连接属性排好序，则cost＝Br+Bs+(Frs*Nr*Ns)/Mrs
* 如果必须对文件排序还需要在代价函数中加上排序的代价
* 对于包含B个块的文件排序的代价大约是 (2*B)+(2*B\*log2B)
* 小结：
* 查询处理是关系数据库管理系统的核心，查询优化技术是查询处理的关键技术
* 本章主要内容
* 查询处理过程
* 查询优化
* 代数优化
* 物理优化
* 查询执行
* 比较复杂的查询，尤其是涉及连接和嵌套的查询
* 不要把优化的任务全部放在RDBMS上
* 应该找出RDBMS的优化规律，以写出适合RDBMS自动优化的SQL语句
* 对于RDBMS不能优化的查询需要重写查询语句，进行手工调整以优化性能

# 事务与数据库恢复技术

* 事务处理技术包括数据库恢复技术和并发控制技术。
* 数据库恢复机制和并发控制机制是DBMS的重要组成部分。

# 8.5 事务的基本概念

* 1.事务：是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。
* 事务和程序比较  
  - 在关系数据库中，一个事务可以是一条或多条SQL语句,也可以包含一个或多个程序。  
  - 一个程序通常包含多个事务
* 显式定义方式：
* begin transaction
* ....
* commit/rollback
* 示例：
* begin transaction
* select \* from teacher;
* update teacher
* set title=null
* where tno=‘101’;
* select \* from teacher;
* rollback;
* select \* from teacher;
* 隐式方式
* 当用户没有显式地定义事务时，DBMS按缺省规定自动划分事务
* AutoCommit事务是SQL Server默认事务方式,
* 2.事务的特性(ACID特性)  
  1.- 原子性（atomicity)：一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做；  
  2.- 一致性（consistency)：事务必须使数据库从一个一致性状态变成另一个一致性状态；  
  3.- 隔离性（isolation)：一个事务的执行不能被其他事务干扰；  
  4.- 持续性（durability)：也称永久性，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的．

# 8.6 数据库恢复概述

- 故障是不可避免的  
- 系统故障：计算机软、硬件故障  
- 人为故障：操作员的失误、恶意的破坏等。  
- 数据库的恢复  
- 把数据库从错误状态恢复到某一已知的正确状态(亦称为一致状态或完整状态)

# 8.7 故障的种类

* 1、事务内部的故障：
  + 1)有的是可以通过事务程序本身发现的
  + 2)有的是非预期的，不能由应用程序处理（如运算溢出、死锁等)
* 以后，事务故障仅指非预期的故障
* 事务故障的恢复：UNDO撤销
* 2、系统故障
* 造成系统停止运转的任何事件，使得系统要重新启动。（硬件错误、断电等)
* 影响正在运行的所有事务，但不破坏数据库。此时内存，尤其是数据库缓冲区中的内容全部丢失，所有运行事务非正常终止。
* 恢复：
  + 1)未提交的事务：UNDO撤销所有未完成的事务
  + 2)已提交的事务，但缓冲区内容未完全写入磁盘：REDO重做所有已提交的事务
* 3、介质故障
* 系统故障称为软故障，介质故障称为硬故障（外存故障，如磁盘损坏)
* 恢复：装入数据库发生介质故障前某个时刻的数据副本
* REDO自此时开始的所有成功事务，将这些事务已提交的结果重新写入数据库
* 4、计算机病毒
* 计算机病毒是一种人为的故障或破坏，是一些恶作剧者研制的一种计算机程序。
* 可以繁殖和传播，并造成对计算机系统包括数据库的危害。
* 总之：各类故障，对数据库的影响有两种可能性  
  - 一是数据库本身被破坏  
  - 二是数据库没有被破坏，但数据可能不正确，这是由于事务的运行被非正常终止造成的。

# 8.8 恢复的实现技术

- 恢复操作的基本原理：冗余  
- 恢复机制涉及的两个关键问题  
1）- 如何建立冗余数据  
•- 数据转储（backup)  
•- 登录日志文件（logging)  
2）- 如何利用这些冗余数据实施数据库恢复

## 数据转储

- 数据转储定义：

* 转储是指DBA将整个数据库复制到其他存储介质上保存起来的过程，备用的数据称为后备副本或后援副本  
  - 如何使用  
  1）- 数据库遭到破坏后可以将后备副本重新装入  
  2）- 重装后备副本只能将数据库恢复到转储时的状态  
  - 转储方法  
  1）- 静态转储与动态转储  
  2）- 海量转储与增量转储
* 静态转储：
  + 1)定义：在系统中无事务运行时进行的转储操作。转储开始的时刻数据库处于一 致性状态，而转储不允许对数据库的任何存取、修改活动。静态转储得到的一定是一个数据一致性的副本。
  + 2)优点：实现简单
  + 3)缺点：降低了数据库的可用性
* 转储必须等待正运行的用户事务结束才能进行；新的事务必须等待转储结束才能执行
* 动态转储：
  + 1)定义：转储期间允许对数据库进行存取或修改。转储和用户事务可以并发执行。
  + 2)优点：不用等待正在运行的用户事务结束；不会影响新事务的运行。
  + 3)实现：必须把转储期间各事务对数据库的修改活动登记下来，建立日志文件 后备副本加上日志文件就能把数据库恢复到某一时刻的正确状态。
* 海量转储：
  + 1)定义：每次转储全部数据库
  + 2)特点：从恢复角度，使用海量转储得到的后备副本进行恢复更方便一些。
* 增量转储：
  + 1)定义：每次只转储上一次转储后更新过的数据
  + 2)特点：如果数据库很大，事务处理又十分频繁，则增量转储方式更实用更有效。

## 日志文件

* 1、什么是日志文件
* 日志文件(log)是用来记录事务对数据库的更新操作的文件
* 2、日志文件的格式
  + 1)以记录为单位：
* 日志文件中需要登记的内容包括：   
  ①- 各个事务的开始标记(BEGIN TRANSACTION)  
  ②- 各个事务的结束标记(COMMIT或ROLLBACK)  
  ③- 各个事务的所有更新操作
* 以上均作为日志文件中的一个日志记录
* 每个日志记录的内容：  
  ①- 事务标识（标明是哪个事务)   
  ②- 操作类型（插入、删除或修改)  
  ③- 操作对象（记录内部标识)  
  ④- 更新前数据的旧值（对插入操作而言，此项为空值)  
  ⑤- 更新后数据的新值（对删除操作而言, 此项为空值)
  + 2)以数据块为单位
* 日志记录内容包括：
* 事务标识（标明是哪个事务)
* 被更新的数据块
* 3、日志文件的作用：  
  - 进行事务故障恢复  
  - 进行系统故障恢复  
  - 协助后备副本进行介质故障恢复
  + 1)事务故障恢复和系统故障恢复必须用日志文件
  + 2)在动态转储方式中必须建立日志文件，后备副本和日志文件结合起来才能有效地恢复数据库
  + 3)静态转储方式中也可以建立日志文件（重新装入后备副本，然后利用日志文件把已完成的事务进行重做，对未完成事务进行撤销)
* 4、登记日志文件：  
  - 基本原则  
  - 登记的次序严格按并行事务执行的时间次序  
  - 必须先写日志文件，后写数据库
* 为什么要先写日志文件？
  + 1)写数据库和写日志文件是两个不同的操作，在这两个操作之间可能发生故障
  + 2)如果先写了数据库修改，而在日志文件中没有登记下这个修改，则以后就无法恢复这个修改了
  + 3)如果先写日志，但没有修改数据库，按日志文件恢复时只不过是多执行一次不必要的UNDO操作，并不会影响数据库的正确性

# 8.9 恢复策略

## 事务故障的恢复

- 事务故障：事务在运行至正常终止点前被终止  
- 恢复方法  
- 由恢复子系统应利用日志文件撤消（UNDO)此事务已对数据库进行的修改  
- 事务故障的恢复由系统自动完成，对用户是透明的，不需要用户干预  
- 事务故障的恢复步骤

* 1. 反向扫描文件日志，查找该事务的更新操作。
  2. 对该事务的更新操作执行逆操作。即将日志记录中“更新前的值” 写入数据库。  
     - 插入操作， “更新前的值”为空，则相当于做删除操作  
     - 删除操作，“更新后的值”为空，则相当于做插入操作  
     - 若是修改操作，则相当于用修改前值代替修改后值
  3. 继续反向扫描日志文件，查找该事务的其他更新操作，并做同样处理。
  4. 如此处理下去，直至读到此事务的开始标记，事务故障恢复就完成了。

## 系统故障的恢复

- 系统故障造成数据库不一致状态的原因  
- 未完成事务对数据库的更新已写入数据库  
- 已提交事务对数据库的更新还留在缓冲区没来得及写入数据库  
- 恢复方法  
- 1. Undo 故障发生时未完成的事务  
- 2. Redo 已完成的事务  
- 系统故障的恢复由系统在重新启动时自动完成，不需要用户干预  
- 系统故障的恢复步骤

* 1. 正向扫描日志文件  
     - 重做(REDO) 队列: 在故障发生前已经提交的事务  
     - 这些事务既有BEGIN TRANSACTION记录，也有COMMIT记录  
     - 撤销 (Undo)队列: 故障发生时尚未完成的事务  
     - 这些事务只有BEGIN TRANSACTION记录，无相应的COMMIT记录
  2. 对撤销(Undo)队列事务进行撤销(UNDO)处理  
     - 反向扫描日志文件，对每个UNDO事务的更新操作执行逆操作
  3. 对重做(Redo)队列事务进行重做(REDO)处理  
     - 正向扫描日志文件，对每个REDO事务重新执行登记的操作

## 介质故障的恢复

- 恢复步骤  
- 1.重装数据库  
•- 装入最新的后备副本，使数据库恢复到最近一次转储时的一致性状态。  
–- 对于静态转储的数据库副本，装入后数据库即处于一致性状态  
–- 对于动态转储的数据库副本，还须同时装入转储时刻的日志文件副本，利用恢复系统故障的方法（即REDO+UNDO)，才能将数据库恢复到一致性状态。  
- 2. 装入有关的日志文件副本，重做已完成的事务。  
•- 首先扫描日志文件，找出故障发生时已提交的事务的标识，将其记入重做队列。  
•- 然后正向扫描日志文件，对重做队列中的所有事务进行重做处理。  
- 介质故障的恢复需要DBA介入  
- DBA的工作  
- 重装最近转储的数据库副本和有关的各日志文件副本  
- 执行系统提供的恢复命令，具体的恢复操作仍由DBMS完成

# 8.10 具有检查点的数据恢复

- 利用日志技术进行数据库恢复存在两个问题  
- 搜索整个日志将耗费大量的时间  
- REDO处理：事务实际上已经执行，又重新执行，浪费了大量时间  
- 具有检查点（checkpoint)的恢复技术  
- 在日志文件中增加检查点记录（checkpoint)  
- 增加重新开始文件，并让恢复子系统在登录日志文件期间动态地维护日志  
- 检查点记录的内容  
- 建立检查点时刻所有正在执行的事务清单  
- 这些事务最近一个日志记录的地址  
- 重新开始文件的内容  
- 记录各个检查点记录在日志文件中的地址

- 动态维护日志文件的方法

* 周期性地执行如下操作：建立检查点，保存数据库状态。
* 具体步骤是：
* 1.将当前日志缓冲区中的所有日志记录写入磁盘的日志文件上
* 2.在日志文件中写入一个检查点记录
* 3.将当前数据缓冲区的所有数据记录写入磁盘的数据库中
* 4.把检查点记录在日志文件中的地址写入一个重新开始文件

- 使用检查点方法可以改善恢复效率  
- 当事务T在一个检查点之前提交  
- T对数据库所做的修改一定都已写入数据库  
- 写入时间是在这个检查点建立之前或在这个检查点建立之时  
- 在进行恢复处理时，没有必要对事务T执行REDO操作  
- 使用检查点的恢复步骤  
- 1.从重新开始文件中找到最后一个检查点记录在日志文件中的地址，由该地址在日志文件中找到最后一个检查点记录  
- 2.由该检查点记录得到检查点建立时刻所有正在执行的事务清单ACTIVE-LIST  
•- 建立两个事务队列  
–- UNDO-LIST  
–- REDO-LIST  
•- 把ACTIVE-LIST暂时放入UNDO-LIST队列，REDO队列暂为空  
- 3.从检查点开始正向扫描日志文件，直到日志文件结束  
•- 如有新开始的事务Ti，把Ti暂时放入UNDO-LIST队列  
•- 如有提交的事务Tj，把Tj从UNDO-LIST队列移到REDO-LIST队列  
- 4.对UNDO-LIST中的每个事务执行UNDO操作

* 对REDO-LIST中的每个事务执行REDO操作

# 8.11 数据库镜像

* 为避免硬盘介质出现故障影响数据库的可用性，许多DBMS提供了数据库映像（mirror)功能用于数据库恢复。
* 将整个数据库或其中的关键数据复制到另一个磁盘上，每当主数据库更新时，DBMS自动把更新后的数据复制过去，由DBMS自动保证镜像数据与主数据库的一致性。一旦出现介质故障，可由镜像磁盘继续提供使用，同时DBMS自动利用磁盘数据进行数据库的恢复，不需要关闭系统和重装数据库副本。
* 在没有出现故障时，数据库镜像还可以用于并发操作，即当一个用户对数据库加排它锁修改数据时，其他用户可以读镜像数据库上的数据，而不必等待该用户释放锁。
* 由于数据库镜像是通过复制数据实现的，频繁地赋值数据自然会降低系统运行效率。因此在实际应用中用户往往只选择对关键数据和日志文件进行镜像。
* 小结：  
  - 如果数据库只包含成功事务提交的结果，就说数据库处于一致性状态。保证数据一致性是对数据库的最基本的要求。  
  - 事务是数据库的逻辑工作单位  
  - DBMS保证系统中一切事务的原子性、一致性、隔离性和持续性  
  - DBMS必须对事务故障、系统故障和介质故障进行恢复  
  - 恢复中最经常使用的技术：数据库转储和登记日志文件  
  - 恢复的基本原理：利用存储在后备副本、日志文件和数据库镜像中的冗余数据来重建数据库  
  - 常用恢复技术  
  - 事务故障的恢复  
  - UNDO  
  - 系统故障的恢复  
  - UNDO + REDO  
  - 介质故障的恢复  
  - 重装备份并恢复到一致性状态 + REDO  
  - 提高恢复效率的技术  
  - 检查点技术  
  - 可以提高系统故障的恢复效率  
  - 可以在一定程度上提高利用动态转储备份进行介质故障恢复的效率  
  - 镜像技术  
  - 镜像技术可以改善介质故障的恢复效率

# 并发控制

* 多用户数据库：允许多个用户同时使用的数据库（订票系统)
* 不同的多事务执行方式：
* 1.串行执行：每个时刻只有一个事务运行，其他事务必须等到这个事务结束后方能运行。
* 2.交叉并发方式：
* 单处理机系统中，事务的并发执行实际上是这些并行事务的并行操作轮流交叉运行（不是真正的并发，但是提高了系统效率)
* 3.同时并发方式：
* 多处理机系统中，每个处理机可以运行一个事务，多个处理机可以同时运行多个事务，实现多个事务真正的并行运行
* 并发执行带来的问题：
* 多个事务同时存取同一数据（共享资源)
* 存取不正确的数据，破坏事务一致性和数据库一致性

# 8.12 并发控制概述

* 并发操作带来的数据不一致性包括
  + 1)丢失修改（lost update)
  + 2)不可重复读（non-repeatable read)
  + 3)读脏数据（dirty read)
* 记号:W（x)写数据x R（x)读数据x
* 并发控制机制的任务：
  + 1)对并发操作进行正确的调度
  + 2)保证事务的隔离性
  + 3)保证数据库的一致性
* 并发控制的主要技术
  + 1)封锁（locking)（主要使用的)
  + 2)时间戳（timestamp)
  + 3)乐观控制法（optimistic scheduler)
  + 4)多版本并发控制（multi-version concurrency control ，MVCC)

# 8.13 封锁

* 封锁：封锁就是事务T在对某个数据对象（例如表、记录等)操作之前，先向系统发出请求，对其加锁。加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其它的事务不能更新此数据对象
* 确切的控制由封锁的类型决定
* 基本的封锁类型有两种：排它锁（X锁，exclusive locks)、共享锁（S 锁，share locks)
* 排它锁又称写锁，对A加了排它锁之后，其他事务不能对A加 任何类型的锁（排斥读和写)
* 共享锁又称读锁，对A加了共享锁之后，其他事务只能对A加S锁，不能加X锁（只排斥写)

# 8.14 封锁协议

* 在运用X锁和S锁对数据对象加锁时，需要约定一些规则：封锁协议（Locking Protocol)
* 何时申请X锁或S锁
* 持锁时间、何时释放
* 对封锁方式制定不同的规则，就形成了各种不同的封锁协议。
* 常用的封锁协议：三级封锁协议
* 三级封锁协议在不同程度上解决了并发问题，为并发操作的正确调度提供一定的保证。
* 1、一级封锁协议
* 事务T在修改数据R之前，必须先对其加X锁，直到事务结束（commit/rollback)才释放。
* 一级封锁协议可以防止丢失修改
* 如果是读数据，不需要加锁的，所以它不能保证可重复读和不读“脏”数据。
* 2、 二级封锁协议
* 在一级封锁协议的基础（写要加X锁，事务结束释放)上，增加事务T在读入数据R之前必须先对其加S锁，读完后即可释放S锁。（读要加S锁，读完即释放)
* 二级封锁协议除了可以防止丢失修改，还可以防止读脏数据
* 由于读完数据即释放S锁，不能保证不可重复读
* 3、三级封锁协议：
* 在一级封锁协议基础上增加事务T在读取数据R之前必须先对其加S锁，直到事务结束后释放。
* 三级封锁协议除了可以防止丢失修改和读脏数据外，还防止了不可重复读
* 三级封锁协议的主要区别是什么操作需要申请锁，何时释放锁。封锁协议越高，一致性程度越高。

# 8.15 饥饿和死锁

## 饥饿

* 饥饿：事务T1封锁了数据R，事务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁之后，系统首先批准了T3的请求，T2仍然等待。 T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求……T2有可能永远等待，这就是饥饿的情形
* 避免饥饿的方法：先来先服务
* 当多个事务请求封锁同一数据对象时，按请求封锁的先后次序对这些事务排队
* 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁。

## 死锁

* 死锁：事务T1封锁了数据R1， T2封锁了数据R2。 T1又请求封锁R2，因T2已封锁了R2，于是T1等待T2释放R2上的锁。 接着T2又申请封锁R1，因T1已封锁了R1，T2也只能
* 等待T1释放R1上的锁。 这样T1在等待T2，而T2又在等待T1，T1和T2两个事务永远不能结束，形成死锁。
* 解决死锁的方法：预防、诊断和解除
* 1、死锁的预防
* 产生死锁的原因是两个或多个事务都已经封锁了一些数据对象，然后又都请求对已被其他事务封锁的数据对象加锁，从而出现死等待。
* 预防死锁发生就是破坏产生死锁的条件
* 方法
  + 1)一次封锁法：
* 要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行。
* 存在的问题：降低系统的并发度；难以实现精确确定封锁对象
  + 2)顺序封锁法：
* 预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实施封锁。
* 存在的问题：
* 维护成本：数据库系统中的封锁对象极多，并且在不断地变化
* 难以实现：很难实现确定每一个事务要封锁哪些对象
* DBMS普通采用的诊断并解除死锁的方法
* 2、死锁的诊断和解除
* 方法：超时法和事务等待图法
  + 1)超时法：如果一个事务的等待时间超过了规定的时限，就认为发生了死锁
* 优点：实现简单
* 缺点：误判死锁；时限若设置太长，死锁发生后不能及时发现。
  + 2)事务等待图法：用事务等待图动态反映所有事务的等待情况事务
* 等待图是一个有向图G=(T，U)，T为结点的集合，每个结点表示正运行的事务， U为边的集合，每条边表示事务等待的情况。若T1等待T2，则T1、T2之间划一条有向边，从T
* 1指向T2。
* 并发控制子系统周期性地（比如每隔数秒)生成事务等待图，检测事务。如果发现图中存在回路，则表示系统中出现了死锁。
* 解除死锁：并发控制子系统选择一个处理死锁代价最小的事务，将其撤销。
* 释放该事务持有的所有的锁，使其他事务能够继续运行下去。

# 8.16 并发调度的可串行性

* 什么样的调度是正确的？串行调度是正确的。
* （执行结果等价于串行调度的调度也是正确的，这样的调度称为可串行化调度。)

## 可串行化调度

* 定义：多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同，称这种调度策略为可串行化调度（serializable)。
* 可串行性是并发事务正确调度的准则。按这个准则规定，一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度。

## 冲突可串行化调度

* 判断可串行化调度的充分条件
* 冲突操作：不同的事务对同一个数据的读写和写写操作。
* 不同事务的冲突操作和同一事务的两个操作是不能交换的。
* Ri（x)和Wj（x)不可交换，Wi（x)和Wj（x)不可交换
* 冲突可串行化调度：
* 一个调度Sc在保证冲突操作的次序不变的情况下，通过交换两个事务不冲突操作的次序得到另一个调度Sc’，如果Sc’是串行的，称调度Sc为冲突可串行化的调度。
* 若一个调度是冲突可串行化，则一定是可串行化的。冲突可串行化调度是可串行化调度的充分条件而非必要条件，同样存在不满足冲突可串行化调度的可串行化调度。

# 8.17 两段锁协议

* DBMS的并发控制机制必须提供一定的手段来保证调度是可串行化的。目前DBMS普遍采用两段锁协议（TwoPhase Locking，简称2PL)的方法来显示并发调度的可串行性。
* 两段锁协议是指所有事务必须分两个阶段对数据对象进行加锁和解锁。
  + 1)在对任何数据进行读写操作以前，首先要申请并获得对该数据的锁。
  + 2)在释放一个锁之后，事务不再申请和获得其他任何的锁。
* “两段”锁的含义：事务分为两个阶段
* 第一阶段是获得封锁，也称为扩展阶段
* 事务可以申请获得任何数据对象上的任何类型的锁，但是不能释放任何锁
* 第二阶段是释放封锁，也称为收缩阶段
* 事务可以释放任何数据对象上的任何类型的锁，但是不能再申请任何锁
* 符合两段锁协议的可串行化调度示例：
* 事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件。
* 若并发事务都遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的
* 若并发事务的一个调度是可串行化的，不一定所有事务都符合两段锁协议
* 两段锁协议与防止死锁的一次封锁法
* 一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行，因此一次封锁法遵守两段锁协议
* 但是两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁

# 8.18 封锁的粒度

* 封锁对象的大小称为封锁粒度（granularity)。
* 封锁的对象可以是逻辑单元（属性值、属性值集合、元组、关系、索引项、数据库)，也可以是物理单元（页、物理记录)。
* 选择封锁粒度原则：
* 封锁粒度和系统的并发度和并发控制的开销密切相关
* 封锁的粒度越大，数据库所能够封锁的数据单元就越少，并发度就越低，系统开销也
* 越小；
* 封锁的粒度越小，并发度较高，但系统开销也就越大

## 多粒度封锁

* 如果在一个系统中同时支持多种封锁粒度供不同的事务选择，这种封锁方法称为多粒度封锁。（multiple granularity locking)
* 选择封锁粒度应该同时考虑封锁开销和并发度两个因素，适当选择封锁粒度以求得最优的效果。
* 需要处理多个关系的大量元组的用户事务：以数据库为封锁单位
* 需要处理大量元组的用户事务：以关系为封锁单元
* 只处理少量元组的用户事务：以元组为封锁单位
* 多粒度树
* 以树形结构来表示多级封锁粒度。根结点是整个数据库，表示最大的数据粒度，叶结点表示最小的数据粒度
* 多粒度封锁协议：允许多粒度树中的每个节点被独立地加锁，对一个节点加锁意味着这个节点的所有子节点也被加以同样类型的锁。因此，在多粒度封锁中一个数据对象可能以显式封锁和隐式封锁两种方式封锁。
* • 显式封锁：直接加到数据对象上的封锁
* • 隐式封锁：该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁
* • 显式封锁和隐式封锁的效果是一样的
* 系统检查封锁冲突时要检查显式封锁，还要检查隐式封锁
* 例如事务T要对关系R1加X锁，系统必须搜索其上级结点数据库、关系R1，还要搜索R1的下级结点，即R1中的每一个元组。如果其中某一个数据对象已经加了不相容锁，则T必
* 须等待。
* 对某个数据对象加锁，系统要检查该数据对象上有无显式封锁与之冲突；再检查其所有上级节点，看本事务的显式封锁是否与该数据对象上的隐式封锁（由于上级节点已加的封锁造成的)冲突；还要检查其所有下级节点，看它们的显式封锁是否与本事务的隐式封锁（将加到下级节点的封锁)冲突。
* 这种检查方法效率较低，引入一种新的锁，意向锁。有了意向锁，DBMS就无须逐个检查下一级节点的显式封锁。

## 意向锁

* 意向锁：如果对一个节点加意向锁，则可说明该节点的下层节点正在被加锁；对任一节点加锁时，必须先对它的上层节点加意向锁。
* 例如，对任一元组加锁时，必须先对它所在的数据库和关系加意向锁。
* 三种常用的意向锁：意向共享锁（Intent Share Lock，IS锁)；意向排它锁（Intent Exclusive Lock，IX锁)；共享意向排它锁（Share Intent Exclusive Lock，SIX锁)。
* 1、IS锁
* 如果对一个数据对象加IS锁，表示它的子节点拟加S锁。
* 例如：事务T1要对R1中某个元组加S锁，则要首先对关系R1和数据库加IS锁
* 2、IX锁
* 如果对一个数据对象加IX锁，表示它的子节点拟加X锁。
* 例如：事务T1要对R1中某个元组加X锁，则要首先对关系R1和数据库加IX锁
* 3、SIX锁
* 如果对一个数据对象加SIX锁，表示对它加S锁，再加IX锁，即SIX = S + IX。
* 例如：对某个表加SIX锁，则表示该事务要读整个表（所以要对该表加S锁)，同
* 时会更新个别元组（所以要对该表加IX锁)
* 意向锁的强度： 锁的强度是指它对其他锁的排斥程度。一个事务在申请封锁时以强锁代替弱锁是安全的，反之则不然。
* 具有意向锁的多粒度封锁方法
* 申请封锁时应该按自上而下的次序进行
* 释放封锁时则应该按自下而上的次序进行
* 优点：
  + 1)提高了系统并发度
  + 2)减少了加锁和解锁的开销
* 在实际的DBMS产品中得到广泛应用。

# 8.19 其他并发控制机制

* 并发控制的方法除了封锁技术外，还有时间戳方法、乐观控制法和多版本并发控制。
* 时间戳方法：给每一个事务盖上一个时标，即事务开始的时间。每个事务具有唯一的时间戳，并按照这个时间戳来解决事务的冲突操作。如果发生冲突操作，就回滚到具有较早时间戳的事务，以保证其他事务的正常执行，被回滚的事务被赋予新的时间戳被从头开始执行。
* 乐观控制法认为事务执行时很少发生冲突，所以不对事务进行特殊的管制，而是让它自由执行，事务提交前再进行正确性检查。如果检查后发现该事务执行中出现过冲突并影响了可串行性，则拒绝提交并回滚该事务。又称为验证方法
* 多版本控制是指在数据库中通过维护数据对象的多个版本信息来实现高效并发的一种策略。

# 范式（避免数据冗余和操作异常)

# 8.20 函数依赖

* A->B A和B是两个属性集，来自同一关系模式，对于同样的A属性值，B属性值也相同

# 8.21 平凡的函数依赖

* X->Y，如果Y是X的子集

# 8.22 非平凡的函数依赖

* X->Y，如果Y不是X的子集

# 8.23 部分函数依赖

* X->Y，如果存在W->Y，且W⊂X

# 8.24 传递函数依赖

* 在R(U)中，如果X→Y(非平凡函数依赖,完全函数依赖)，Y→Z， 则称Z对X传递函数依赖。
* 记为：XZ

# 8.25 super key&candidate key&primary key&主属性&非主属性

* super key：在关系中能唯一标识元素的属性集
* candidate key或key：不含有多余属性的super key
* primary key：在candidate key 中任选一个
* candidate key中X决定所有属性的函数依赖是完全函数依赖
* 包含在任何一个candidate key中的属性 ，称为主属性
* 不包含在candidate key中的属性称为非主属性

# 8.26 1NF 列不可分

* 列不可分

# 8.27 2NF 消除了非主属性对键的部分函数依赖

* 在关系T上有函数依赖集F，F+是F的闭包。
* F满足2NF，当且仅当 每个非平凡的函数依赖X->A(F+),A是单个非主属性，要求X不是任何key的真子集（有可能是super key，也有可能是非key)。

# 8.28 3NF 消除了非主属性对键的传递函数依赖

* F满足3NF，当且仅当 每个非平凡的函数依赖X->A(F+),A是单个非主属性，要求X是T的super key。

# 8.29 BCNF 消除了主属性对键的部分函数依赖和传递函数依赖

* F满足BCNF，当且仅当 每个非平凡的函数依赖X->A(F+),A是单个属性，要求X是T的super key。
* 对于F+中 的任意一个X->A，如果A是单个属性，且A不在X中，那么X一定是T的super key。

# 反范式（减少连接，提高查询效率)

# 8.30 Pattern1：合并1对1关系

* 例：学院给老师配车，车少人多，车完全参与，人部分参与
* car  
  car*id car*name  
  1 c1  
  2 c2  
  3 c3
* teacher  
  teacher*id teacher*name  
  1 t1  
  2 t2  
  3 t3  
  4 t4
* 合并后
* car*and*teacher  
  car*id car*name teacher*id teacher*name  
  1 c1 1 t1  
  2 c2 2 t2  
  3 c3 3 t3  
  NULL NULL 4 t4
* 问题：会产生大量空值，若两边都部分参与则不能合并；
* 部分参与为大部分参与时比较适合Pattern1

# 8.31 Pattern2：1对N关系中复制非键属性以减少连接

* 两表连接时复制非键属性以减少连接
* 例：查询学生以及所在学院名，可以在学生表中不仅存储学院id，并且存储学院名
* faculty  
  fid fname  
  1 f1
* student  
  sid sname fid fname  
  1 s1 1 f1
* 维护时：
  + 1)如果在UI中，只允许用户进行选择，不能自行输入，保证输入一致性
  + 2)如果是程序员，对于类似学院名这种一般不变的代码表，在修改时直接对两张表都进行修改；如果经常变化，则可以加一个触发器。

# 8.32 Pattern3：1对N关系中复制外键以减少连接

* 把另一张表的主键复制变成外键
* 应用后：

# 8.33 Pattern4：N对N关系中复制属性，把两张表中经常需要的内容复制到中间关系表中以减少连接

* 例：

# 8.34 Pattern5：引入重复值

* 通常对于一个多值属性，值不太多，且不会经常变，可以在表中建立多个有关此属性的列
* address1 | address2 | address3 | address4

# 8.35 Pattern6：建立提取表

* 为了解决查询和更新之间不可调和的矛盾，可以将更新和查询放在两张表中，从工作表中提取查询表，专门用于查询。只适用于查询实时性不高的情况。

# 8.36 Pattern7：分表

* 水平拆分
* 垂直拆分

# MySQL使用

# MySQL特点

* MySQL是一个关系型数据库管理系统，开发者为瑞典MySQL AB公司。目前MySQL被广泛地应用在互联网行业。由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，许多互联网公司选择了MySQL作为后端数据库。2008年MySQL被Sun公司收购，2010年甲骨文成功收购Sun公司。
* MySQL数据库的优点：
* 1、多语言支持：MySQL为C、C++、Python、Java、Perl、PHP、Ruby等多种编程语言提供了API，访问和使用方便。
* 2、可以移植性好：MySQL是跨平台的。
* 3、免费开源。
* 4、高效：MySQL的核心程序采用完全的多线程编程。
* 5、支持大量数据查询和存储：MySQL可以承受大量的并发访问

# 数据类型

# 8.37 数值类型

* 整数类型：
* 实数类型：
* 定点数：DECIMAL和NUMERIC类型在MySQL中视为相同的类型。它们用于保存必须为确切精度的值。
* DECIMAL(M,D)，其中M表示十进制数字总的个数，D表示小数点后面数字的位数。  
   - 如果存储时，整数部分超出了范围（如上面的例子中，添加数值为1000.01)，MySql就会报错，不允许存这样的值。  
  - 如果存储时，小数点部分若超出范围，就分以下情况：  
  - 若四舍五入后，整数部分没有超出范围，则只警告，但能成功操作并四舍五入删除多余的小数位后保存。如999.994实际被保存为999.99。  
  - 若四舍五入后，整数部分超出范围，则MySql报错，并拒绝处理。如999.995和-999.995都会报错。
* M的默认取值为10，D默认取值为0。如果创建表时，某字段定义为decimal类型不带任何参数，等同于decimal(10,0)。带一个参数时，D取默认值。
* M的取值范围为1~65，取0时会被设为默认值，超出范围会报错。
* D的取值范围为0~30，而且必须<=M，超出范围会报错。
* 所以，很显然，当M=65，D=0时，可以取得最大和最小值。
* 浮点数类型：float，double和real。他们定义方式为：FLOAT(M,D) 、 REAL(M,D) 、 DOUBLE PRECISION(M,D)。 “(M,D)”表示该值一共显示M位整数，其中D位位于小数点后面
* FLOAT和DOUBLE中的M和D的取值默认都为0，即除了最大最小值，不限制位数。
* M取值范围为0~255。FLOAT只保证6位有效数字的准确性，所以FLOAT(M,D)中，M<=6时，数字通常是准确的。如果M和D都有明确定义，其超出范围后的处理同decimal。
* D取值范围为0~30，同时必须<=M。double只保证16位有效数字的准确性，所以DOUBLE(M,D)中，M<=16时，数字通常是准确的。如果M和D都有明确定义，其超出范围后的处理同decimal。
* 内存中，FLOAT占4-byte（1位符号位 8位表示指数 23位表示尾数)，DOUBLE占8-byte（1位符号位 11位表示指数 52位表示尾数)。
* 浮点数比定点数类型存储空间少，计算速度快，但是不够精确。
* 因为需要计算额外的空间和计算开销，所以应该尽量只在对小数进行精确计算时 才使用DECIMAL。但在数据量比较大的情况下，可以考虑使用BIGINT代替DECIMAL，将需要存储的货币单位根据小数的位数乘以相应的倍数即可。
* BIT数据类型可用来保存位字段值。BIT(M)类型允许存储M位值。M范围为1~64，默认为1。
* BIT其实就是存入二进制的值，类似010110。
* 如果存入一个BIT类型的值，位数少于M值，则左补0.
* 如果存入一个BIT类型的值，位数多于M值，MySQL的操作取决于此时有效的SQL模式：
* 如果模式未设置，MySQL将值裁剪到范围的相应端点，并保存裁减好的值。
* 如果模式设置为traditional(“严格模式”)，超出范围的值将被拒绝并提示错误，并且根据SQL标准插入会失败。
* MySQL把BIT当做字符串类型，而非数字类型。

# 8.38 字符串类型

* 字符串类型指CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM和SET。

## CHAR & VARCHAR

* CHAR和VARCHAR类型声明的长度表示你想要保存的最大字符数。例如，CHAR(30)可以占用30个字符。默认长度都为255。
* CHAR列的长度固定为创建表时声明的长度。长度可以为从0到255的任何值。当保存CHAR值时，在它们的右边填充空格以达到指定的长度。当检索到CHAR值时，尾部的空格被删除掉，所以，我们在存储时字符串右边不能有空格，即使有，查询出来后也会被删除。在存储或检索过程中不进行大小写转换。
* 所以当char类型的字段为唯一值时，添加的值是否已经存在以不包含末尾空格（可能有多个空格)的值确定，比较时会在末尾补满空格后与现已存在的值比较。
  + VARCHAR列中的值为可变长字符串。长度可以指定为0到65,535之间的值（实际可指定的最大长度与编码和其他字段有关，比如，MySql使用utf-8编码格式，大小为标准格式大小的2倍，仅有一个varchar字段时实测最大值仅21844，如果添加一个char(3)，则最大取值减少3。整体最大长度是65,532字节)。
* 同CHAR对比，VARCHAR值保存时只保存需要的字符数，另加一个字节来记录长度(如果列声明的长度超过255，则使用两个字节)。
* VARCHAR值保存时不进行填充。当值保存和检索时尾部的空格仍保留，符合标准SQL。
* 如果分配给CHAR或VARCHAR列的值超过列的最大长度，则对值进行裁剪以使其适合。如果被裁掉的字符不是空格，则会产生一条警告。如果裁剪非空格字符，则会造成错误(而不是警告)并通过使用严格SQL模式禁用值的插入。

## BINARY & VARBINARY

* BINARY和VARBINARY类型类似于CHAR和VARCHAR类型，但是不同的是，它们存储的不是字符串，而是二进制串。所以它们没有编码格式，并且排序和比较基于列值字节的数值值。
* 当保存BINARY值时，在它们右边填充0x00(零字节)值以达到指定长度。取值时不删除尾部的字节。比较时所有字节很重要（因为空格和0x00是不同的，0x00<空格)，包括ORDER BY和DISTINCT操作。比如插入'a '会变成'a \0'。
* 对于VARBINARY，插入时不填充字符，选择时不裁剪字节。比较时所有字节很重要。

## BLOB & TEXT

* BLOB是一个二进制大对象，可以容纳可变数量的数据。有4种BLOB类型：TINYBLOB、BLOB、MEDIUMBLOB和LONGBLOB。它们只是可容纳值的最大长度不同。
* 有4种TEXT类型：TINYTEXT、TEXT、MEDIUMTEXT和LONGTEXT。这些对应4种BLOB类型，有相同的最大长度和存储需求。
* BLOB列被视为二进制字符串。TEXT列被视为字符字符串，类似BINARY和CHAR。
* 在TEXT或BLOB列的存储或检索过程中，不存在大小写转换。
* 未运行在严格模式时，如果你为BLOB或TEXT列分配一个超过该列类型的最大长度的值，值被截取以保证适合。如果截掉的字符不是空格，将会产生一条警告。使用严格SQL模式，会产生错误，并且值将被拒绝而不是截取并给出警告。
* 在大多数方面，可以将BLOB列视为能够足够大的VARBINARY列。同样，可以将TEXT列视为VARCHAR列。
* BLOB和TEXT在以下几个方面不同于VARBINARY和VARCHAR：
* 当保存或检索BLOB和TEXT列的值时不删除尾部空格。(这与VARBINARY和VARCHAR列相同)。
* 比较时将用空格对TEXT进行扩充以适合比较的对象，正如CHAR和VARCHAR。
* 对于BLOB和TEXT列的索引，必须指定索引前缀的长度。对于CHAR和VARCHAR，前缀长度是可选的。
* BLOB和TEXT列不能有默认值。
* BLOB或TEXT对象的最大大小由其类型确定，但在客户端和服务器之间实际可以传递的最大值由可用内存数量和通信缓存区大小确定。你可以通过更改max*allowed*packet变量的值更改消息缓存区的大小，但必须同时修改服务器和客户端程序。
* 每个BLOB或TEXT值分别由内部分配的对象表示。
* 它们（TEXT和BLOB同)的长度：
  + Tiny：最大长度255个字符(2^8-1)
  + BLOB或TEXT：最大长度65535个字符(2^16-1)
  + Medium：最大长度16777215个字符(2^24-1)
  + LongText 最大长度4294967295个字符(2^32-1)
* 实际长度与编码有关，比如utf-8的会减半。
* 当BLOB和TEXT值太大时，InnoDB会使用专门的外部存储区域来进行存储，此时单个值在行内需要1~4个字节存储一个指针，然后在外部存储区域存储实际的值。
* MySQL会BLOB和TEXT进行排序与其他类型是不同的：它只对每个类的最前max*sort*length字节而不是整个字符串进行排序。
* MySQL不能将BLOB和TEXT列全部长度的字符串进行索引，也不能使用这些索引消除排序。
* ENUM 使用枚举代替字符串类型
* MySQL在存储枚举时非常紧凑，会根据列表值的数量压缩到一个或两个字节中。MySQL在内部将每个值在列表中的位置保存为整数，并且在表的.frm文件中保存“数组——字符串”映射关系的查找表。
* 枚举字段是按照内部存储的整数而不是定义的字符串进行排序的；
* 由于MySQL把每个枚举值都保存为整数，并且必须通过查找才能转换为字符串，所以枚举列有一定开销。在特定情况下，把CHAR/VARCHAR列与枚举列进行JOIN可能会比直接关联CHAR/VARCHAR更慢。

# 8.39 时间和日期类型

* DATE, DATETIME, 和TIMESTAMP类型 这三者其实是关联的，都用来表示日期或时间。
* 当你需要同时包含日期和时间信息的值时则使用DATETIME类型。MySQL以'YYYY-MM-DD HH:MM:SS'格式检索和显示DATETIME值。支持的范围为'1000-01-01 00:00:00'到'9999-12-31 23:59:59'。
* 当你只需要日期值而不需要时间部分时应使用DATE类型。MySQL用'YYYY-MM-DD'格式检索和显示DATE值。支持的范围是'1000-01-01'到 '9999-12-31'。
* TIMESTAMP类型同样包含日期和时间，范围从'1970-01-01 00:00:01' UTC 到'2038-01-19 03:14:07' UTC。
* TIME值的范围可以从'-838:59:59'到'838:59:59'。小时部分会因此大的原因是TIME类型不仅可以用于表示一天的时间(必须小于24小时)，还可能为某个事件过去的时间或两个事件之间的时间间隔(可以大于24小时，或者甚至为负)
* 两者的存储方式不一样
* 对于TIMESTAMP，它把客户端插入的时间从当前时区转化为UTC（世界标准时间)进行存储。查询时，将其又转化为客户端当前时区进行返回。
* 而对于DATETIME，不做任何改变，基本上是原样输入和输出。
* YEAR类型是一个单字节类型用于表示年。
* MySQL以YYYY格式检索和显示YEAR值。范围是1901到2155。

# 逻辑架构

* MySQL的特点体现在其存储引擎的架构上。
* 插件式的存储引擎架构将查询处理和其他的系统任务以及数据的存储提取相分离，这种架构可以让用户根据业务需求和实际需要选择合适的存储引擎。

# 8.40 连接层（管理客户端的连接，维护线程池)

* 最上层是一些客户端和连接服务，引入了线程池的概念；实现基于SSL的安全连接
* 每个客户端都会在服务器进程中拥有一个线程，这个连接的查询只会在这个单独的线程中执行。
* 当客户端连接到MySQL服务器时，服务器需要对其进行认证。如果使用了SSL安全套接字的方式连接，还会使用X.509证书认证。一旦客户端连接成功，服务器会继续验证该客户端是否具有执行某个特定查询的权限。

# 8.41 服务器（与具体存储引擎解耦，服务器通过API与存储引擎进行通信)

- SQL接口  
- SQL分析与优化  
- 存储过程  
- 触发器  
- 视图

* MySQL会解析查询，并创建内部数据结构（解析树)，然后对其进行各种优化，包括重写查询、决定表的读取顺序，以及选择合适的索引等。
* 优化器并不关心表使用的是什么存储引擎，但存储引擎对优化查询是有影响的。优化器会请求存储引擎提供容量或某个具体操作的开销信息，以及表数据的统计信息等。
* 对于SELECT语句，在解析查询前，服务器会先检查查询缓存，如果能够在其中找到对应的查询，服务器就不再执行查询解析、优化和执行的整个过程，而是直接返回查询缓存中的结果集。

# 8.42 存储引擎层（负责数据的存储和存取)

* 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。
* 存储引擎API包含了几十个底层函数，用于执行诸如“开启一个事务”或者“根据主键提取一行记录”等操作。但存储引擎不会去解析SQL（InnoDB是一个例外，它会解析外键定义，因为服务器没有实现该功能)，不同存储引擎之间也不会相互通信，而只是简单地响应上层服务器的请求。

# 8.43 存储层（将数据存储到文件系统上)

* 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。

# 存储引擎

* MyISAM崩溃后无法安全恢复（由于不支持事务)

# 约束

- 主键约束 ：不允许重复记录，避免数据冗余  
- 外键约束：保证本事务所关联的其他事务是存在的（主键表中的这个字段)  
- check约束：限制某一个值在某一个范围之内

* check（) （)内是关系表达式和逻辑表达式的嵌套
* 注意逻辑运算符是not and or   
  - default约束：确定默认值(可以更改) 保证事务的某个属性一定会有一个值
* 有默认值的话如果不想对其更改，可以用insert对其他字段进行赋值，跳过有默认值的字段
* 但是不能在整体insert的时候跳过这个字段

- unique约束 唯一键：唯一的值不可重复，但允许为空

* 就是该记录的这个值不会有重复的值 unique和not null 可以组合使用，顺序任意
* 注意空值可以写为null，注意空值的这个值也不能重复，只能有一条记录的这个字段可以是空值 （而oracle中可以允许多个有唯一键的记录为空值)

- not null约束 要求用户必须为该字段赋一个值，否则出错

* 如果非空的话必须赋值，不能采用部分insert的办法来跳过对这个字段的赋值
* 不写not null/null 的话默认就是允许有空值，如果没赋值的话字段的值默认是null
* null和default 关系：都允许不对某字段进行赋值，但是结果不同，一个是空值，另一个是默认值
* create table student2
* (
* stu\_id int primary key,
* stu\_name nvarchar(20) unique not null,
* stu*sal int check(stu*sal >= 1000 and stu\_sal <2000) not null,
  + stu\_sex nchar(1) default '男'
* )
* insert into student values(1,'啦啦',1800,'男') ok
* insert into student values(2,'啦啦',1800,'男') error
* insert into student values(2,'嘿嘿',2400,null) error
* insert into student values(null,'嘿嘿',1200,null) ok
* 主键和唯一键的关系：
* 不要用业务逻辑字段当做主键，应添加一个没有任何实际意义的字段（代理主键)当做主键
* 一般是主键（或者唯一键)作为其他表的外键。
* 如果业务逻辑字段的信息修改，则会影响其他表
* 查询效率低（数字、编号效率高)
* 这个业务逻辑字段修改时，因为这个主键同时充当多个其他表的外键，所以也要一并修改，十分麻烦
* 将有实际业务含义的、不能重复的、不是主键的一个字段作为唯一键

# MySQL常用函数

# 8.44 文本处理函数

* Left(x,len) – 返回串左边的字符（长度为len)
* Right(x,len)
* Length(x) – 返回串的长度
* Locate(x,sub\_x) – 找出串的一个子串
* SubString(x, from, to) – 返回字串的字符
* Lower(x)
* Upper(x)
* LTrim(x)
* RTrim(x)
* Soundex(x) – 读音（用于发音匹配)
* SELECT cust*name, cust*contact FROM customers WHERE Soundex(cust\_contact) = Soundex(‘Y Lie’);

# 8.45 日期和时间处理函数

* 日期和时间采用相应的数据类型和特殊的格式存储，以便可以快速和有效的排序或过滤，节省物理存储空间.
* 一般，应用程序不使用用来存储日期和时间的格式，因此日期和时间函数总是被用来读取、统计和处理这些函数.
* 常用日期和时间处理函数：
* AddDate() – 增加一个日期（天，周等)
* AddTime() – 增加一个时间（时，分等)
* CurDate() – 返回当前日期
* CurTime() – 返回当前时间
* Date() – 返回日期时间的日期部分
* DateDiff() – 计算两个日期之差
* Date\_Add() – 日期运算函数
* Date\_Format() – 返回一个格式化的日期或时间串
* Day() – 返回一个日期的天数部分
* DayOfWeek() – 返回日期对应的星期几
* Hour() – 返回一个时间的小时部分
* Minute() – 返回一个时间的分钟部分
* Second() – 返回一个时间的秒部分
* Month() – 返回一个日期的月部分
* Now() – 返回当前日期和时间
* Time() – 返回一个日期时间的时间部分
* Year() – 返回一个日期的年份部分
* 日期首选格式： yyyy-mm-dd; 如2005-09-01
* 检索某日期下的数据：
* SELECT cust*id, order*num FROM orders WHERE Date(order\_date) = ‘2005-09-01’;
* 检索某月或日期范围内的数据：
* SELECT cust*id, order*num FROM orders WHERE Year(order*date) = 2005 AND Month(order*date) = 9;
* – or
* SELECT cust*id, order*num FROM orders WHERE date(order\_date) BETWEEN ‘2005-09-01’ AND ‘2005-09-30’;

# 8.46 数值处理函数

* 代数、三角函数、几何运算等
* 常用数值处理函数：
* abs(); cos(); exp(); mod()（取余); Pi(); Rand(); Sin(); Sqrt(); Tan();

# 视图

* 视图是虚拟的表，与包含数据的表不同，视图只包含使用时动态检索数据的查询,主要是用于查询。

# 8.47 为什么使用视图

- 重用sql语句  
- 简化复杂的sql操作，在编写查询后，可以方便地重用它而不必知道他的基本查询细节。  
- 使用表的组成部分而不是整个表。  
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。  
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

* 注意：  
  - 在视图创建之后，可以用与表基本相同的方式利用它们。可以对视图执行select操作，过滤和排序数据，将视图联结到其他视图或表，甚至能添加和更新数据。

- 重要的是知道视图仅仅是用来查看存储在别处的数据的一种设施。视图本身不包含数据，因此它们返回的数据时从其他表中检索出来的。在添加和更改这些表中的数据时，视图将返回改变过的数据。

- 因为视图不包含数据，所以每次使用视图时，都必须处理查询执行时所需的任一检索。如果你使用多个联结和过滤创建了复杂的视图或者嵌套了视图，可能会发现性能下降得很厉害。因此，在部署使用了大量视图的应用前，应该进行测试。

# 8.48 视图的规则和限制

- 与表一样，视图必须唯一命名；  
- 可以创建任意多的视图；  
- 为了创建视图，必须具有足够的访问权限。这些限制通常由数据库管理人员授予。  
- 视图可以嵌套，可以利用从其他视图中检索数据的查询来构造一个视图。  
- Order by 可以在视图中使用，但如果从该视图检索数据select中也是含有order by，那么该视图的order by 将被覆盖。  
- 视图不能索引，也不能有关联的触发器或默认值  
- 视图可以和表一起使用

# 8.49 视图的创建

- 利用create view 语句来进行创建视图  
- 使用show create view viewname；来查看创建视图的语句  
- 用drop view viewname 来删除视图  
- 更新视图可以先drop在create，也可以使用create or replace view。

# 8.50 视图的更新

* 视图是否可以更新，要视情况而定。
* 通常情况下视图是可以更新的，可以对他们进行insert，update和delete。更新视图就是更新其基表(视图本身没有数据)。如果你对视图进行增加或者删除行，实际上就是对基表进行增加或者删除行。
  + 但是，如果MySQL不能正确的确定更新的基表数据，则不允许更新(包括插入和删除)，这就意味着视图中如果存在以下操作则不能对视图进行更新：(1)分组(使用group by 和 having )；(2)联结；(3)子查询；(4)并；(5)聚集函数;(6)dictinct;(7)导出(计算)列。【注意：基于5.0版本的规则，不排除后续变化】

# 存储过程

* 存储过程就是为了以后的使用而保存的一条或者多条MySQL语句的集合。可将视为批文件，虽然他们的作用不仅限于批处理。

# 8.51 为什么使用储存过程？

* 1.通过把处理封装在容易使用的单元中，简化复杂的操作；
* 2.由于不要求反复建立一系列处理步骤，保证了数据的完整性。如果所有开发人员和应用程序都使用同一(实验和测试)存储过程，则所使用的代码都是相同的。这一点的延伸就是防止错误。需要执行的步骤越多，出错的可能性就越大，防止错误保证了数据的一致性。
* 3.简化对变动的管理，如果表名。列名或者业务逻辑等有变化，只需要更改存储过程的代码。使用它的人员甚至不需要知道这些变化。这一点延伸就是安全性，通过存储过程限制对基数据的访问减少了数据讹误的机会。
* 4.提高性能。因为使用存储过程比使用单独的sql语句更快。
* 5.存在一些只能用在单个请求的MySQL元素和特性，存储过程可以使用他们来编写功能更强更灵活的代码
* 综上：
* 三个主要的好处：简单、安全、高性能。
* 两个缺陷：
* 1、存储过程的编写更为复杂，需要更高的技能更丰富的经验。
* 2、可能没有创建存储过程的安全访问权限。许多数据库管理员限制存储过程的 创建权限，允许使用，不允许创建。

# 8.52 执行存储过程

* Call关键字：Call接受存储过程的名字以及需要传递给他的任意参数。存储过程可以显示结果，也可以不显示结果。
* CREATE PROCEDURE productpricing()
* BEGIN
* SELECT AVG( prod\_price) as priceaverage FROM products;
* END;
* 创建名为productpricing的储存过程。如果存储过程中需要传递参数，则将他们在括号中列举出来即可。括号必须有。BEGIN和END关键字用来限制存储过程体。上述存储过程体本身是一个简单的select语句。注意这里只是创建存储过程并没有进行调用。
* 储存过程的使用：
* Call productpring()；

# 8.53 使用参数的存储过程

* 一般存储过程并不显示结果，而是把结果返回给你指定的变量上。
* 变量：内存中一个特定的位置，用来临时存储数据。
* MySQL> CREATE PROCEDURE prod(
  + out pl decimal(8,2),
  + out ph decimal(8,2),
  + out pa decimal(8,2)
* )
* begin
* select Min(prod\_price) into pl from products;
* select MAx(prod\_price) into ph from products;
* select avg(prod\_price) into pa from products;
* end;
* call PROCEDURE(@pricelow,@pricehigh,@priceaverage);
* select @pricelow;
* select @pricehigh;
* select @pricelow,@pricehigh,@priceaverage;
* 解释：
* 此存储过程接受3个参数，pl存储产品最低价，ph存储产品最高价，pa存储产品平均价。每个参数必须指定类型，使用的为十进制，关键字OUT 指出相应的参数用来从存储过程传出一个值(返回给调用者)。
* MySQL支持in(传递给存储过程)、out(从存储过程传出，这里所用)和inout(对存储过程传入和传出)类型的参数。存储过程的代码位于begin和end语句内。他们是一系列select语句，用来检索值。然后保存到相对应的变量(通过INTO关键字)。
* 存储过程的参数允许的数据类型与表中使用的类型相同。注意记录集是不被允许的类型，因此，不能通过一个参数返回多个行和列，这也是上面为什么要使用3个参数和3条select语句的原因。
* 调用：为调用此存储过程，必须指定3个变量名。如上所示。3个参数是存储过程保存结果的3个变量的名字。调用时，语句并不显示任何数据，它返回以后可以显示的变量(或在其他处理中使用)。
* 注意：所有的MySQL变量都是以@开头。
* CREATE PROCEDURE ordertotal(
* IN innumber int,
  + OUT outtotal decimal(8,2)
* )
* BEGIN
* SELECT Sum(item*price \* quantity) FROM orderitems WHERE order*num = innumber INTO outtotal;
* end //
* CALL ordertotal(20005,@total);
* select @total; // 得到20005订单的合计
* CALL ordertotal(20009,@total);
* select @total; //得到20009订单的合计

# 8.54 带有控制语句的存储过程

* CREATE PROCEDURE ordertotal(
* IN onumber INT,
* IN taxable BOOLEAN,
  + OUT ototal DECIMAL(8,2)
* )COMMENT 'Obtain order total, optionally adding tax'
* BEGIN
* -- declear variable for total
  + DECLARE total DECIMAL(8,2);
* -- declear tax percentage
* DECLARE taxrate INT DEFAULT 6;
* -- get the order total
* SELECT Sum(item*price \* quantity) FROM orderitems WHERE order*num = onumber INTO total;
* -- IS this taxable?
* IF taxable THEN
* -- yes ,so add taxrate to the total
* SELECT total+(total/100\*taxrate)INTO total;
* END IF;
* -- finally ,save to out variable
* SELECT total INTO ototal;
* END;
* 在存储过程中我们使用了DECLARE语句，他们表示定义两个局部变量，DECLARE要求指定变量名和数据类型。它也支持可选的默认值(taxrate默认6%)，因为后期我们还要判断要不要增加税，所以，我们把SELECT查询的结果存储到局部变量total中，然后在IF 和THEN的配合下，检查taxable是否为真，然后在真的情况下，我们利用另一条SELECT语句增加营业税到局部变量total中，然后我们再利用SELECT语句将total(增加税或者不增加税的结果)保存到总的ototal中。
* COMMENT关键字 上面的COMMENT是可以给出或者不给出，如果给出，将在SHOW PROCEDURE STATUS的结果中显示。

# 触发器

* 在某个表发生更改时自动处理某些语句，这就是触发器。
* 触发器是MySQL响应delete 、update 、insert 、位于begin 和end语句之间的一组语句而自动执行的一条MySQL语句。其他的语句不支持触发器。

# 8.55 创建触发器

* 在创建触发器时，需要给出4条语句（规则)：
  1. 唯一的触发器名；
  2. 触发器关联的表；
  3. 触发器应该响应的活动；
  4. 触发器何时执行(处理之前或者之后)
* Create trigger 语句创建 触发器
* CREATE TRIGGER newproduct AFTER INSERT ON products FOR EACH ROW SELECT 'Product added' INTO @info;
* CREATE TRIGGER用来创建名为newproduct的新触发器。触发器可以在一个操作发生前或者发生后执行，这里AFTER INSERT 是指此触发器在INSERT语句成功执行后执行。这个触发器还指定FOR EACH ROW ， 因此代码对每个插入行都会执行。文本Product added 将对每个插入的行显示一次。
* 注意：
* 1、触发器只有表才支持，视图，临时表都不支持触发器。
* 2、触发器是按照每个表每个事件每次地定义，每个表每个事件每次只允许一个触发器，因此，每个表最多支持六个触发器(insert，update，delete的before 和after)。
* 3、单一触发器不能与多个事件或多个表关联，所以，你需要一个对insert和update 操作执行的触发器，则应该定义两个触发器。
* 4、触发器失败：如果before 触发器失败，则MySQL将不执行请求的操作，此外，如果before触发器或者语句本身失败，MySQL则将不执行after触发器。

# 8.56 触发器类别

## INSERT触发器

* 是在insert语句执行之前或者执行之后被执行的触发器。
* 1、在insert触发器代码中，可引入一个名为new的虚拟表，访问被插入的行；
* 2、在before insert触发器中，new中的值也可以被更新(允许更改被插入的值)；
* 3、对于auto\_increment列，new在insert执行之前包含0，在insert执行之后包含新的自动生成值
* CREATE TRIGGER neworder AFTER INSERT ON orders FOR EACH ROW SELECT NEW.order\_num;
* 创建一个名为neworder的触发器，按照AFTER INSERT ON orders 执行。在插入一个新订单到orders表时，MySQL生成一个新的订单号并保存到order*num中。触发器从NEW.order*num取得这个值并返回它。此触发器必须按照AFTER INSERT执行，因为在BEFORE INSERT语句执行之前，新order\_num还没有生成。对于orders的每次插入使用这个触发器总是返回新的订单号。

## DELETE触发器

* Delete触发器在delete语句执行之前或者之后执行。
* 1、在delete触发器的代码内,可以引用一个名为OLD的虚拟表，用来访问被删除的行。
* 2、OLD中的值全为只读，不能更新。
* CREATE TRIGGER deleteorder BEFORE DELETE ON orders FOR EACH ROW
* BEGIN
* INSERT INTO archive*orders(order*num,order*date,cust*id) values (OLD.order*num,OLD.order*date,OLD.cust\_id);
* END;
* CREATE TABLE archive\_orders(
  + order*num int(11) NOT NULL AUTO*INCREMENT,
* order\_date datetime NOT NULL,
  + cust\_id int(11) NOT NULL,
* PRIMARY KEY (order\_num),
* KEY fk*orders1*customers1 (cust\_id),
* CONSTRAINT fk*orders1*customers1 FOREIGN KEY (cust\_id) REFERENCES customers
* (cust\_id)
* ) ENGINE=InnoDB AUTO\_INCREMENT=20011 DEFAULT CHARSET=utf8
* 在任意订单被删除前将执行此触发器，它使用一条INSERT 语句将OLD中的值(要被删除的订单) 保存到一个名为archive*orders的存档表中(为实际使用这个例子，我们需要用与orders相同的列创建一个名为archive*orders的表)
* 使用BEFORE DELETE触发器的优点(相对于AFTER DELETE触发器来说)为，如果由于某种原因，订单不能存档，delete本身将被放弃。
* 我们在这个触发器使用了BEGIN和END语句标记触发器体。这在此例子中并不是必须的，只是为了说明使用BEGIN END 块的好处是触发器能够容纳多条SQL 语句(在BEGIN END块中一条挨着一条)。

## UPDATE触发器

* 在update语句执行之前或者之后执行
* 1、在update触发器的代码内,可以引用一个名为OLD的虚拟表，用来访问以前(UPDATE语句之前)的值，引用一个名为NEW的虚拟表访问新更新的值。
* 2、在BEFORE UPDATE触发器中，NEW中的值可能也被用于更新(允许更改将要用于UPDATE语句中的值)
* 3、OLD中的值全为只读，不能更新。
* CREATE TRIGGER updatevendor BEFORE UPDATE ON vendors FOR EACH ROW SET NEW.vend*state = Upper(NEW.vemd*state);
* 保证州名缩写总是大写(不管UPFATE语句中是否给出了大写)，每次更新一行时，NEW.vend*state中的值(将用来更新表行的值)都用Upper(NEW.vend*state)替换。

# 8.57 总结

* 1、通常before用于数据的验证和净化(为了保证插入表中的数据确实是需要的数据) 也适用于update触发器。
* 2、与其他DBMS相比，MySQL 5中支持的触发器相当初级，未来的MySQL版本中估计会存在一些改进和增强触发器的支持。
* 3、创建触发器可能需要特殊的安全访问权限，但是触发器的执行时自动的，如果insert，update，或者delete语句能够执行，则相关的触发器也能执行。
* 4、用触发器来保证数据的一致性(大小写，格式等)。在触发器中执行这种类型的处理的优点就是它总是进行这种处理，而且透明的进行，与客户机应用无关。
* 5、触发器的一种非常有意义的使用就是创建审计跟踪。使用触发器，把更改(如果需要，甚至还有之前和之后的状态)记录到另外一个表是非常容易的。
* 6、MySQL触发器不支持call语句，无法从触发器内调用存储过程。

# MySQL索引

* b 树和 hash 索引应用场合 区别
* 主键索引和普通索引的区别
* 聚簇索引在底层怎么实现的，数据和关键字是怎么存的
* 复合索引 复合索引要把那个字段放最前，为什么
* 为啥MySQL索引要用B+树而MongoDB用B树？

# 8.58 索引使用的基本原则

* 最经常查询的列上建立聚簇索引以提高查询效率
* 一个基本表最多只建立一个聚簇索引
* 经常更新的列不宜建立聚簇索引
* 主键和唯一键会自动创建索引

# 8.59 索引分类——从数据结构角度

## B-树,B+树,B\*树

* B/B+树是一种多级索引组织方法，是适合于组织存放在外存的大型磁盘文件的一种树状索引结构。其中用得比较多的是B+树。

### 多路查找树

* m叉查找树
* 内结点：非叶节点；外结点：叶节点
* 定义：
  + 1)每个内结点至多有m个孩子和m-1个键值
  + 2)具有n个键值的结点有n+1个孩子
  + 3)有p个键值的结点：C0 K1 C1 K2 ,,, Kp Cp Ci是指针域 Ki是数据域
  + 4)键值有序（从左到右 由小到大)
  + 5)满足查找树的要求 C0所在子树的所有键值 < K1 < C1所在子树的所有键值 < ... < Cp所在子树的所有键值
* 高度与结点关系
* m叉查找树的高度为h，则其 h <= 第h行的结点数 <= mh-1

### B-树

* 平衡的m叉查找树
* 定义：B树首先是一棵多路查找树
  + 1)根节点至少有两个孩子
  + 2)所有非叶结点（除根节点)至少有 ceil(m/2) 个孩子
  + 3)所有叶结点都在同一层，叶结点总数 = 键值总数 +1
  + 因此一个结点的孩子数在 [ceil(m/2),m] 之间
* 随机查找的磁盘访问次数最多为树的高度

### B+树

* 定义：
  + 1)树中每个非叶结点最多有m个孩子
  + 2)根节点至少有2个孩子
  + 3)除根节点外，每个非叶结点至少有ceil(m/2)个孩子
  + 4)有n个孩子的结点有n-1个键值
  + 5)所有叶节点在同一层，包含了所有键值和指向相应数据对象的指针，键值升序
  + 6)每个叶节点中的孩子数允许大于m。假设叶节点可容纳的最多键值数为m1，则指向数据对象的指针数为m1，孩子数n应满足 ceil(m1/2) < n < m1
* 通常在B+树上有两个头指针，一个指向根结点（进行随机搜索)，一个指向关键字最小的叶结点（进行顺序搜索)。
* 随机查找key时每次所需要的磁盘I/O次数等于B+树的高度

### B+树与B树的比较

#### 组织方式不一样

* B+树：所有有效的索引关键字值都必须存储在叶结点中，其内部结点中的键值只用于索引项的查找定位。
* B树：有效的索引关键字值可以出现在B树的任意一个结点中。
* 因此：
* B+树：所有关键字的查找速度基本一致
* B树：依赖于查找关键字所在结点的层次

#### 叶结点不同

* B+树中叶节点间增加链表指针，提供对索引关键字的顺序扫描功能；叶节点的个数未必符合m叉查找树的要求，它依赖于键值字节数和指针字节数，为m1阶。

### 为什么B+比B树更适合实际应用中操作系统的文件索引和数据库索引

- 1) B+的磁盘读写代价更低

* B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。
  + 2) B+树的查询效率更加稳定
* 由于非叶结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
  + 3)树的遍历效率较高
* 数据库索引采用B+树的主要原因是 B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作（或者说效率太低)

### B\*树

* 与B+树的区别：
  + 1)定义了非叶子结点键值个数至少为(2/3)\*m，即块的最低使用率为2/3
  + （代替B+树的1/2)；
  + 2)为非叶结点也增加链表指针
* B\*树分配新结点的概率比B+树要低，空间使用率更高

## MySQL中的B+树适用场景

* InnoDB存储引擎使用的是B+树。
* B+树为对如下类型的查询有效：
  + 1)全值匹配：和索引中的所有列进行匹配
  + 2)匹配最左前缀：只使用索引的第一列或前几列
  + 3)匹配列前缀：只匹配某一列的值的开头部分
  + 4)匹配范围值
  + 5)精确匹配某一列并范围匹配另外一列
  + 6)覆盖索引/只访问索引的查询
* 一般来说，如果B+树可以按照某种方式查找到值，那么也可以按照这种方式用于排序。如果ORDER BY子句满足前面列出的几种查询类型，则这个索引也可以满足对应的排序需求。
* 下面是一些关于B+树索引的限制：
  + 1)如果不是按照索引的最左列开始查找，则无法使用索引
  + 2)不能跳过索引中的列
  + 3)如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找

## Hash索引

* 哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。
* 只有精确匹配索引所有列的查询才有效！
* 在MySQL中，只有Memory引擎显式支持Hash索引。
* 限制：
  + 1)哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行（无法使用覆盖索引)。不过，访问内存中的行的速度很快。
  + 2)哈希索引数据并不是按照索引值顺序存储的，所以无法进行排序
  + 3)哈希索引不支持部分索引列匹配查找。比如建立复合哈希索引(A,B)，无法仅使用A使用哈希索引去查询
  + 4)不支持范围查询，仅支持等值查询
  + 5)哈希冲突严重时，索引维护的代码很高。

## B树索引与Hash索引比较

- 1)如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；

* 如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；
  + 2)哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询)；
  + 3)哈希索引也不支持多列联合索引的最左匹配规则；
  + 4)B+树索引的关键字检索效率比较平均，在有大量重复键值情况下，哈希索引的效率是极低的，因为存在所谓的哈希碰撞问题。

# 8.60 索引分类——从物理存储角度

## 聚簇索引

* InnoDB的聚簇索引实际上在同一个结构中保存了B+树索引和数据行。
* 当表有聚簇索引时，它的数据行实际上存放在索引的叶子页中。聚簇表示数据行和相邻的键值紧紧地存储在一起。因为无法同时把数据行存储在两个不同的地方，所以一个表只能有一个聚簇索引。
* InnoDB通过主键聚簇数据。
* 每张表都会有一个聚簇索引。聚簇索引是一级索引。
* 聚簇索引一般是主键；没有主键，就是第一个唯一键；没有唯一键，就是隐藏ID。
* 聚簇索引以外的所有索引都称为二级索引。在InnoDB中，二级索引中的每条记录都包含该行的主键列，以及为二级索引指定的列。 InnoDB使用这个主键值来搜索聚簇索引中的行。
* 聚簇索引的优点：
  + 1)可以将相关数据保存在一起，只需一次IO就可以取出相邻的数据
  + 2)数据访问更快，因为索引和数据保存在同一个B+树中
  + 3)使用覆盖索引扫描的查询可以直接使用叶节点中的主键值
* 缺点：
  + 1)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
  + 2)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
  + 3)插入新行或者更新主键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，这就是一次页分裂操作。页分裂会导致表占用更多的磁盘空间。
  + 4)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
  + 5)二级索引（非聚簇索引)可能会更大， 因为在二级索引的叶子节点包含了引用行的主键值。这样的策略减少了当出现行移动或者页分裂时二级索引的维护工作。
  + 6)二级索引访问需要两次B树索引查找，而不是一次。因为二级索引中叶子节点保存的是行的主键值，要找到数据行，还需要拿主键值到聚簇索引中进行一次查找。
* 对于InnoDB，自适应哈希索引能够减少这样的重复工作。

## 非聚簇索引

# 8.61 索引分类——从逻辑角度

## 主键索引

* 索引列的值必须唯一，并且不允许有空值

## 唯一索引

* 与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同)。如果是组合索引，则列值的组合必须唯一

## 普通索引

* 最基本的索引，它没有任何限制

## 复合索引

## 全文索引

* 在相同的列上同时创建全文索引和基于值的B树索引不会有冲突，全文索引适用于MATCH AGAINST操作，而不是普通的WHERE条件操作。
* FULLTEXT索引仅可用于 MyISAM 表；他们可以从CHAR、VARCHAR或TEXT列中作为CREATE TABLE语句的一部分被创建，或是随后使用ALTER TABLE 或CREATE INDEX被添加

## 空间索引（R-Tree)

* 空间索引：空间索引是对空间数据类型的字段建立的索引，MYSQL中的空间数据类型有4种，分别是GEOMETRY、POINT、LINESTRING、POLYGON
* MyISAM表支持空间索引，可以用作地理数据存储。和B树索引不同，这类索引无须前缀查询。空间索引会从所有维度来索引数据。查询时，可以有效地使用任意维度来组合查询。MySQL中的GIS支持并不完善，做的比较好的关系数据库是PostgreSQL的PostGIS。

# 8.62 索引的特殊应用

## InnoDB AUTO\_INCREMENT

* 如果正在使用InnoDB表并且没有什么数据需要聚集，那么可以定义一个代理键作为主键，这种主键的数据应该与应用无关，最简单的方法是使用AUTO\_INCREMENT自增列。这样可以保证数据行是按顺序写入的，对于根据主键做关联操作的性能也会更好。
* 最好避免随机的聚簇索引，特别是对于IO密集型应用，比如UUID，它使得聚簇索引的插入变得完全随机，这是最坏的情况，使得数据没有任何聚集特性。
* 如果主键的值是顺序的，那么InnoDB会把每一条记录都存储在上一条记录的后面。当达到页的最大填充因子时，下一条记录就会写入新的页中。一旦数据按照这种顺序的方式加载，主键页就会近似于被顺序的记录填满，这也正是所期望的结果。
* 使用UUID作为主键的缺点：
  + 1)写入的目标页可能已经刷到磁盘上并从缓存中移除，或者还没有被加载到缓存中，InnoDB在插入之前不得不先找到并从磁盘读取目标页到内存中，这将导致大量的随机iO
  + 2)因为写入是乱序的，InnoDB不得不频繁地做页分裂操作，以便为新的行分配空间。页分裂会导致移动大量数据，一次插入最少修改三个页而不是一个页。
  + 3)由于频繁的页分裂，页会变得稀疏并被不规则填充，所以最终数据会有碎片。
* 在把这些随机值载入到聚簇索引后，也许需要做一次OPTIMIZE TABLE来重建表并优化页的填充。
* 使用InnoDB时应该尽可能地按主键顺序插入数据，并且尽可能地使用单调增加的聚簇值来插入新行。
* 顺序主键的缺点是什么？
* 对于高并发工作负载，在InnoDB中按主键顺序插入可能会造成明显的争用。主键的上界会成为热点。因为所有的插入都在这里，所以并发插入可能导致锁竞争。另一个热点可能是AUTO\_INCREMENT锁机制，可能需要重新设计表或应用。
* AUTO-INC锁是当向使用含有AUTO\_INCREMENT列的表中插入数据时需要获取的一种特殊的表级锁
* 在最简单的情况下，如果一个事务正在向表中插入值，则任何其他事务必须等待对该表执行自己的插入操作，以便第一个事务插入的行的值是连续的。
* innodb*autoinc*lock\_mode配置选项控制用于自动增量锁定的算法。 它允许您选择如何在可预测的自动递增值序列和插入操作的最大并发性之间进行权衡。
* innodb会在内存里保存一个计数器用来记录auto*increment的值，当插入一个新行数据时，就会用一个表锁来锁住这个计数器，直到插入结束。如果一行一行的插入数据则没有什么问题，但是如果大量的并发插入就废了，表锁会引起SQL堵塞，不但影响效率，而且可能会瞬间达到max*connections而崩溃。
* InnoDB提供了一个可配置的锁定机制，可以显著提高使用AUTO*INCREMENT列向表中添加行的SQL语句的可伸缩性和性能。 要对InnoDB表使用AUTO*INCREMENT机制，必须将AUTO*INCREMENT列定义为索引的一部分，以便可以对表执行相当于索引的SELECT MAX（ai*col)查找以获取最大列值。 通常，这是通过使列成为某些表索引的第一列来实现的。
* 下面介绍AUTO*INCREMENT锁定模式的行为，对不同AUTO*INCREMENT锁定模式设置的使用含义，以及InnoDB如何初始化AUTO\_INCREMENT计数器。

### 插入类型

- 1)simple inserts

* simple inserts指的是那种能够事先确定插入行数的语句，比如INSERT/REPLACE INTO 等插入单行或者多行的语句，语句中不包括嵌套子查询。此外，INSERT INTO … ON DUPLICATE KEY UPDATE这类语句也要除外。
  + 2)bulk inserts
* bulk inserts指的是事先无法确定插入行数的语句，比如INSERT/REPLACE INTO … SELECT, LOAD DATA等。
  + 3)mixed-mode inserts
* 指的是simple inserts类型中有些行指定了auto\_increment列的值，有些行没有指定，比如：
  + INSERT INTO t1 (c1,c2) VALUES (1,’a’), (NULL,’b’), (5,’c’), (NULL,’d’);
* 另外一种mixed-mode inserts是 INSERT … ON DUPLICATE KEY UPDATE这种语句，可能导致分配的auto\_increment值没有被使用。

### innodb*autoinc*lock\_mode 配置

#### innodb*autoinc*lock\_mode=0（traditional lock mode)

* 传统的auto*increment机制。这种模式下所有针对auto*increment列的插入操作都会加AUTO-INC锁，分配的值也是一个个分配，是连续的，正常情况下也不会有间隙（当然如果事务rollback了这个auto\_increment值就会浪费掉，从而造成间隙)。

#### innodb*autoinc*lock\_mode=1（consecutive lock mode)

* 这种情况下，针对bulk inserts才会采用AUTO-INC锁这种方式，而针对simple inserts，则直接通过分析语句，获得要插入的数量，然后一次性分配足够的auto\_increment id，只会将整个分配的过程锁住。。当然，如果其他事务已经持有了AUTO-INC锁，则simple inserts需要等待.
* 针对Mixed-mode inserts：直接分析语句，获得最坏情况下需要插入的数量，然后一次性分配足够的auto\_increment id，只会将整个分配的过程锁住。
* 保证同一条insert语句中新插入的auto*increment id都是连续的，语句之间是可能出现auto*increment值的空隙的。比如mixed-mode inserts以及bulk inserts中都有可能导致一些分配的auto\_increment值被浪费掉从而导致间隙。

#### innodb*autoinc*lock\_mode=2（interleaved lock mode)

* 这种模式下任何类型的inserts都不会采用AUTO-INC锁，性能最好。这种模式是来一个分配一个，而不会锁表，只会锁住分配id的过程，和innodb*autoinc*lock*mode = 1的区别在于，不会预分配多个。但是在replication中当binlog*format为statement-based时（简称SBR statement-based replication)存在问题，因为是来一个分配一个，这样当并发执行时，“Bulk inserts”在分配时会同时向其他的INSERT分配，会出现主从不一致（从库执行结果和主库执行结果不一样)，因为binlog只会记录开始的insert id。
* 可能会在同一条语句内部产生auto\_increment值间隙。

### 不同模式下间隙情况

#### simple inserts

* 针对innodb*autoinc*lock*mode=0,1,2，只有在一个有auto*increment列操作的事务出现回滚时，分配的auto\_increment的值会丢弃不再使用，从而造成间隙。

#### bulk inserts

* innodb*autoinc*lock\_mode=0,由于一直会持有AUTO-INC锁直到语句结束，生成的值都是连续的，不会产生间隙。
* innodb*autoinc*lock\_mode=1，这时候一条语句内不会产生间隙，但是语句之间可能会产生间隙。
* innodb*autoinc*lock\_mode=2，如果有并发的insert操作，那么同一条语句内都可能产生间隙。

#### mixed-mode inserts

* 这种模式下针对innodb*autoinc*lock*mode的值配置不同，结果也会不同，当然innodb*autoinc*lock*mode=0时时不会产生间隙的，而innodb*autoinc*lock*mode=1以及innodb*autoinc*lock*mode=2是会产生间隙的。
* 另外注意的一点是，在master-slave这种架构中，复制如果采用statement-based replication这种方式，则innodb*autoinc*lock*mode=0或1才是安全的。而如果是采用row-based replication或者mixed-based replication，则innodb*autoinc*lock*mode=0,1,2都是安全的。

## 覆盖索引

* 如果一个索引包含了所有需要查询字段的值，就称为覆盖索引。
* 覆盖索引的优点：
  + 1)索引条目远少于数据行大小，如果只需要读取索引，则MySQL就会极大地减少数据访问了，这对缓存的负载非常重要，因为这种情况下响应时间大部分花费在数据拷贝上。覆盖索引对IO密集型应用也有帮助，因为索引比数据更小，更容易全部放入内存中。
  + 2)因为索引是按照列值顺序存储的，对于IO密集型的范围查询会比随机从磁盘读取每一行数据的IO次数会少得多。
  + 3)InnoDB的二级索引在叶节点中保存了行的主键值，如果二级索引是覆盖索引，则可以避免对主键聚簇索引的二次查询。
* 不是所有类型的索引都可以成为覆盖索引。覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引都不存储索引列的值，所以MySQL只能使用B树索引做覆盖索引。
* 当发起一个索引覆盖查询时，在EXPLAIN的Extra列可以看到Using index的信息。
* InnoDB的二级索引的叶子节点都包含了主键的值，这意味着InnoDB的二级索引可以有效利用这些额外的主键列来覆盖查询。

## 使用索引进行排序

* MySQL有两种可以生成有序的结果：通过排序操作；按索引顺序扫描。如果EXPLAIN出来的type列的值为index，则说明MySQL使用了索引顺序扫描来做排序。
* 扫描索引本身是很快的，但如果索引不能覆盖查询所需的全部列，那就不得不每扫描一条索引记录就都回表查询一次对应的行。这基本上都是随机IO，因此按索引顺序读取数据的速度通常要比顺序地全表扫描要慢，尤其是在IO密集型的工作负载时。
* 只有当索引的列顺序和ORDER BY子句的顺序完全一致，并且所有列的排序方向（降序或升序，索引默认是升序)都一样时，MySQL才可以使用索引来对结果做排序。如果查询需要关联多张表，则只有当ORDER BY子句引用的字段全部为第一张表时，才能使用索引做排序。
* ORDER BY子句和查找型索引的限制是一样的，都需要满足索引的最左前缀的要求。
* 有一种情况下ORDER BY子句可以不满足索引的最左前缀的要求，就是前导列为常量的时候。

## 前缀压缩索引

* MyISAM通过前缀压缩来减少索引的大小，从而让更多的索引可以放入内存中。默认只压缩字符串，但通过参数调整也能对整数进行压缩。
* MyISAM压缩每个索引块的方法时，先完全保存索引块的第一个值，然后将其他值和第一个值进行比较得到相同前缀的字节数和剩余的不同后缀部分，把这部分存储起来即可。
* 压缩块使用更少的情况，代价是某些操作可能更慢。因为每个值的压缩前缀都依赖前面的值，所以MyISAM查找时无法在索引块使用二分查找而只能从头开始扫描。

## 冗余和重复索引

* 冗余索引：MySQL允许在相同列上创建多个索引。MySQL需要单独维护重复的索引，并且优化器在优化查询时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。
* 冗余索引和重复索引有一些不同。如果创建了索引(A,B)，又创建了索引(A)就是冗余索引，索引(A,B)也可以当做索引(A)来使用。但是如果再创建索引(B,A)，就不是冗余索引。另外，其他不同类型的索引也不会是B树索引的冗余索引。
* 冗余索引通常发生在为表添加新索引的时候。例如，有人可能会增加一个新的索引(A,B)而不是扩展已有的索引(A)，还有一种情况是将一个索引扩展为(A,PK)，对于InnoDB而言PK已经包含在二级索引中了，所以这也是冗余的。
* 大多数情况下都不需要冗余索引，应该尽量扩展已有的索引而不是创建新索引。但也有时候出于性能方面的考虑需要冗余索引，因为扩展已有的索引会导致其变得太大，从而影响其他使用该索引的查询的性能。
* 例如，现在在整数列上有一个索引，需要额外增加一个很长的VARCHAR列来扩展该索引，那性能可能会急剧下降。
* 可以使用一些工具来找出冗余和重复的索引。

## 索引重用

* 现有索引(A,B,C)，如果要使用索引，那么where中必须写为A=a and B = b and C = c。如果没有对B的筛选，还想使用索引，怎么绕过最左前缀匹配呢？
  + 假设B是一个选择性很低的列，只有b1和b2两种取值，那么查询可以写为A = a and B in(b1,b2) and C = c。

## 避免多个范围条件

* 对于范围条件查询，MySQL无法再使用范围列后面的其他索引列了，但是对于多个等值条件查询（in ...)则没有这个限制。
* 假设有索引(A,B)，查询条件为 A > a and B < b，那么此时无法同时使用A和B的复合索引，只能用到A的索引。一定要用的话可以考虑将A转为in(a1,a2...)。

## 优化limit

* 延迟关联，使用覆盖索引

# 8.63 适合建索引的情况

* 主键
* 连接中频繁使用的列
* 在某一范围内频繁搜索的列和按排列顺序频繁搜索的列

# 8.64 不适合建索引的情况

* 很少或从来不在查询中引用的列
* 只有两个或很少几个值的列
* 以bit text image 数据类型定义的列
* 数据行数很少的小表

# 8.65 索引优点

- 1)大大减少了服务器需要扫描的数据量  
- 2)帮助服务器减少排序和临时表（group by和order by都可以使用索引，因为索引有序)  
- 3)可以将随机IO变为顺序IO（覆盖索引)

# 8.66 索引缺点

* 创建索引要花费时间，占用存储空间
* 减慢数据修改速度

# 8.67 索引失效

* CREATE TABLE staffs (
* id INT PRIMARY KEY AUTO\_INCREMENT,
  + NAME VARCHAR (24) NOT NULL DEFAULT '' COMMENT '姓名',
* age INT NOT NULL DEFAULT 0 COMMENT '年龄',
* pos VARCHAR (20) NOT NULL DEFAULT '' COMMENT '职位',
* add*time TIMESTAMP NOT NULL DEFAULT CURRENT*TIMESTAMP COMMENT '入职时间'
* ) CHARSET utf8 COMMENT '员工记录表' ;
* ALTER TABLE staffs ADD INDEX idx*staffs*nameAgePos(name, age, pos);

## 1、全值匹配

* EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July';
* EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25;
* EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25 AND pos = 'dev';

## 2、最佳左前缀法则

* 如果索引了多列，要遵守最佳左前缀法则，指的是从索引的最左边的列开始并且不跳过索引中的列。
* 查询时就按照建索引的顺序进行筛选
* EXPLAIN SELECT \* FROM staffs WHERE age = 25 AND pos = 'dev';
* EXPLAIN SELECT \* FROM staffs WHERE pos = 'dev';

## 3、在索引上使用表达式

- 索引列上使用了表达式，如where substr(a, 1, 3) = 'hhh'，where a = a + 1，表达式是一大忌讳，再简单MySQL也不认。

* 有时数据量不是大到严重影响速度时，一般可以先查出来，比如先查所有有订单记录的数据，再在程序中去筛选
* 哪怕是该字段没有建立索引，但不能保证以后不在这个字段上建立索引，所以可以这么说：不要在任何字段上进行操作。

## 4、range 类型查询字段后面的索引无效

* 最后一次只用到了两个索引
* 此时可以建一个只含前两个字段的索引

## 5、尽量使用覆盖索引

## 6、使用不等于时索引失效

## 7、is (not) null 时索引失效

* 如果没有值，可以使其等于一个默认值，这样就可以利用到索引了。

## 8、like 以通配符开头会导致全表扫描

## 9、varchar 类型不加单引号索引失效

* 不加单引号会出现类型转换，此时索引失效

## 10、使用or时索引失效

* 所以要少用or

# 8.68 总结

* 假设index(a,b,c)  
  Where语句 索引是否被使用  
  where a = 3 Y,使用到a  
  where a = 3 and b = 5 Y,使用到a，b  
  where a = 3 and b = 5 and c = 4 Y,使用到a,b,c  
  where b = 3 | where b = 3 and c = 4 | where c = 4 N  
  where a = 3 and c = 5 使用到a， 但是C不可以，中间断了  
  where a = 3 and b > 4 and c = 7 使用到a和b， c在范围之后，断了  
  where a = 3 and b like 'kk%' and c = 4 同上

# MySQL查询分析工具

# 8.69 慢查询日志

* MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阈值的语句，具体指运行时间超过long*query*time值的SQL，则会被记录到慢查询日志中。long*query*time的默认值为10，意思是运行10S以上的语句。默认情况下，Mysql数据库并不启动慢查询日志，需要我们手动来设置这个参数，当然，如果不是调优需要的话，一般不建议启动该参数，因为开启慢查询日志会或多或少带来一定的性能影响。慢查询日志支持将日志记录写入文件，也支持将日志记录写入数据库表。
* slow*query*log ：是否开启慢查询日志，1表示开启，0表示关闭。
* slow-query-log-file：新版（5.6及以上版本)MySQL数据库慢查询日志存储路径。可以不设置该参数，系统则会默认给一个缺省的文件host\_name-slow.log
* long*query*time ：慢查询阈值，当查询时间多于设定的阈值时，记录日志。
* log*queries*not*using*indexes：未使用索引的查询也被记录到慢查询日志中（可选项)。
* log*output：日志存储方式。log*output='FILE'表示将日志存入文件，默认值是'FILE'。log*output='TABLE'表示将日志存入数据库，这样日志信息就会被写入到mysql.slow*log表中。MySQL数据库支持同时两种日志存储方式，配置的时候以逗号隔开即可，如：log\_output='FILE,TABLE'。日志记录到系统的专用日志表中，要比记录到文件耗费更多的系统资源，因此对于需要启用慢查询日志，又需要能够获得更高的系统性能，那么建议优先记录到文件。
* 在实际生产环境中，如果要手工分析日志，查找、分析SQL，显然是个体力活，MySQL提供了日志分析工具mysqldumpslow。
* s, 是表示按照何种方式排序
* c: 访问计数
* l: 锁定时间
* r: 返回记录
* t: 查询时间
* al:平均锁定时间
* ar:平均返回记录数
* at:平均查询时间
* -t, 是top n的意思，即为返回前面多少条的数据；
* -g, 后边可以写一个正则匹配模式，大小写不敏感的；
* 比如:
* 得到返回记录集最多的10个SQL。
* mysqldumpslow -s r -t 10 /database/mysql/mysql06\_slow.log
* 得到访问次数最多的10个SQL
* mysqldumpslow -s c -t 10 /database/mysql/mysql06\_slow.log
* 得到按照时间排序的前10条里面含有左连接的查询语句。
* mysqldumpslow -s t -t 10 -g “left join” /database/mysql/mysql06\_slow.log
* 另外建议在使用这些命令时结合 | 和more 使用 ，否则有可能出现刷屏的情况。
* mysqldumpslow -s r -t 20 /mysqldata/mysql/mysql06-slow.log | more

# 8.70 explain

* explain SQL分析 每个列代表什么含义（关于优化级别 ref 和 all，什么时候应该用到index却没用到，关于extra列出现了usetempory 和 filesort分别的原因和如何着手优化等)
* 各字段解释：

## id

- 1)id相同，表示执行顺序从上到下

* where 条件从右往左读取
  + 2)id不同，如果是子查询，id的序号会递增，id越大优先级越高，越先被执行
* primary 是主查询
* subquery是子查询
  + 3)id有相同的，也有不同的，同时存在
* id相同的可以被认为是一组，从上往下顺序执行
* 在所有组中，id值越大，优先级越高，越先执行。
* Derived：衍生的

## select\_type

## table

* 显示这一行的数据是来自哪一张表

## type（重要)

* type显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：
* system > const > eq*ref > ref > fulltext > ref*or*null > index*merge > unique*subquery > index*subquery > range > index > ALL ，
* 一般来说，得保证查询至少达到range级别，最好能达到ref。

## possible\_keys

* 显示可能应用在这张表中的索引，一个或多个。
* 查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用
* 理论上可以被用上的

## key

* 实际使用到的索引

## key\_len

## ref

* 由key*len可知t1表的idx*col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'
* 用到了多少个字段上的索引，ref就会有几个（大部分情况)
* 或者可以根据key\_len的倍数来判断使用了几个字段上的索引

## rows

* rows越少越好

## extra

* 前两个最重要：

### 1、Using fileSort

- 建立索引的作用1)查询2)排序

* 如果排序字段没有索引，那么可能会产生filesort文件排序，降低效率。

### 2、临时表Using temporary

* 如果数据量很大，使用临时表效率会很低。

### 3、Using Index

* 覆盖索引（Covering Index),一说为索引覆盖。
* 理解方式一:就是select的数据列只用从索引中就能够取得，不必读取数据行，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,换句话说查询列要被所建的索引覆盖。
* 理解方式二:索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据;当能通过读取索引就可以得到想要的数据，那就不需要读取行了。一个索引包含了(或覆盖了)满足查询结果的数据就叫做覆盖索引。
* 注意：
* 如果要使用覆盖索引，一定要注意select列表中只取出需要的列，不可select \*，
* 因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

### 4、Impossible where

## explain实例1

- 第一行（执行顺序4)：id列为1，表示是union里的第一个select，select\_type列的primary表 示该查询为外层查询，table列被标记为<derived3>，表示查询结果来自一个衍生表，其中derived3中3代表该查询衍生自第三个select查询，即id为3的select。【select d1.name......】  
- 第二行（执行顺序2)：id为3，是整个查询中第三个select的一部分。因查询包含在from中，所以为derived。【select id,name from t1 where other\_column=''】  
- 第三行（执行顺序3)：select列表中的子查询select\_type为subquery，为整个查询中的第二个select。【select id from t3】  
- 第四行（执行顺序1)：select\_type为union，说明第四个select是union里的第二个select，最先执行【select name,id from t2】  
- 第五行（执行顺序5)：代表从union的临时表中读取行的阶段，table列的<union1,4>表示用第一个和第四个select的结果进行union操作。【两个结果union操作】

## explain实例2（单表)

* 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
* );
* #查询 category*id 为 1 且 comments 大于 1 的情况下,views 最多的 article*id。
* EXPLAIN SELECT id,author*id FROM article WHERE category*id = 1 AND comments > 1 ORDER BY views DESC LIMIT 1;
* #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。
* #开始优化：

# 1.1 新建索引+删除索引

* #ALTER TABLE article ADD INDEX idx*article*ccv ( category\_id , comments, views );
* create index idx*article*ccv on article(category\_id,comments,views);
* DROP INDEX idx*article*ccv ON article

# 1.2 第2次EXPLAIN

* EXPLAIN SELECT id,author*id FROM article WHERE category*id = 1 AND comments >1 ORDER BY views DESC LIMIT 1;
* EXPLAIN SELECT id,author*id FROM article WHERE category*id = 1 AND comments =3 ORDER BY views DESC LIMIT 1
* #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。

# 1.3 删除第一次建立的索引

* DROP INDEX idx*article*ccv ON article;

# 1.4 第2次新建索引

* #ALTER TABLE article ADD INDEX idx*article*cv ( category\_id , views ) ;
* create index idx*article*cv on article(category\_id,views);

# 1.5 第3次EXPLAIN

* EXPLAIN SELECT id,author*id FROM article WHERE category*id = 1 AND comments > 1 ORDER BY views DESC LIMIT 1;
* #结论：可以看到,type 变为了 ref,Extra 中的 Using filesort 也消失了,结果非常理想。
* DROP INDEX idx*article*cv ON article;

## explain实例3（两表)

* 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)
* );
* class： book：

# 下面开始explain分析

* EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;
* #结论：type 有All

# 添加索引优化

* ALTER TABLE book ADD INDEX Y ( card);

# 第2次explain

* EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;
* #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。
* #左外连接索引建右表

# 删除旧索引 + 新建 + 第3次explain

* DROP INDEX Y ON book;
* ALTER TABLE class ADD INDEX X (card);
* EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card = book.card;

# 然后来看一个右连接查询:

* #优化较明显。这是因为 RIGHT JOIN 条件用于确定如何从左表搜索行,右边一定都有,所以左边是我们的关键点,一定需要建立索引。
* EXPLAIN SELECT \* FROM class RIGHT JOIN book ON class.card = book.card;
* DROP INDEX X ON class;
* ALTER TABLE book ADD INDEX Y (card);

# 右连接，基本无变化

* EXPLAIN SELECT \* FROM class RIGHT JOIN book ON class.card = book.card;

## explain实例3（三表)

* phone：
* ALTER TABLE phone ADD INDEX z ( card);
* ALTER TABLE book ADD INDEX Y ( card);#上一个case建过一个同样的
* EXPLAIN SELECT \* FROM class LEFT JOIN book ON class.card=book.card LEFT JOIN phone ON book.card = phone.card;

# 后 2 行的 type 都是 ref 且总 rows 优化很好,效果不错。因此索引最好设置在需要经常查询的字段中。

* ==================================================================================
* 【结论】
* Join语句的优化
* 尽可能减少Join语句中的NestedLoop的循环总次数；“永远用小结果集驱动大的结果集”。
* 优先优化NestedLoop的内层循环；
* 保证Join语句中被驱动表上Join条件字段已经被索引；
* 当无法保证被驱动表的Join条件字段被索引且内存资源充足的前提下，不要太吝惜JoinBuffer的设置；

# 8.71 show profile

## 是否支持

* show variables like 'profiling';

## 开启功能

## 查看结果

* show profiles;

## 诊断SQL

* 一般性建议

# 8.72 习题

* 【建表语句】
* create table test03(
* id int primary key not null auto\_increment,
* c1 char(10),
* c2 char(10),
* c3 char(10),
* c4 char(10),
* c5 char(10)
* );
* 【建索引】
  + create index idx*test03*c1234 on test03(c1,c2,c3,c4);
* show index from test03;
* 问题：我们创建了复合索引idx*test03*c1234 ,根据以下SQL分析下索引使用情况？
* explain select \* from test03 where c1='a1';
* explain select \* from test03 where c1='a1' and c2='a2';
* explain select \* from test03 where c1='a1' and c2='a2' and c3='a3';
* explain select \* from test03 where c1='a1' and c2='a2' and c3='a3' and c4='a4';
  + 1)
* explain select \* from test03 where c1='a1' and c2='a2' and c3='a3' and c4='a4';
* 4
  + 2)
* explain select \* from test03 where c1='a1' and c2='a2' and c4='a4' and c3='a3';
* 4
* 原因是MySQL的optimizer会进行优化，将查询语句调整为索引的顺序
  + 3)
* explain select \* from test03 where c1='a1' and c2='a2' and c3>'a3' and c4='a4';
* 3
  + 4)
* explain select \* from test03 where c1='a1' and c2='a2' and c4>'a4' and c3='a3';
* 4
* SQL的优化器也调整为1,2,3,4的顺序
  + 5)
* explain select \* from test03 where c1='a1' and c2='a2' and c4='a4' order by c3;
* c3作用在排序而不是查找
* 因为有c3的索引，所以没有出现using filesort
* 2 无filesort
  + 6)
* explain select \* from test03 where c1='a1' and c2='a2' order by c3;
* 因为有c3的索引，所以没有出现using filesort
* 2 无filesort
  + 7)
* explain select \* from test03 where c1='a1' and c2='a2' order by c4;
* 出现了filesort
* 2 有filesort
  + 8)
* 8.1 explain select \* from test03 where c1='a1' and c5='a5' order by c2,c3;
* 只用c1一个字段索引，但是c2、c3用于排序,无filesort
* 1 无filesort
* 8.2 explain select \* from test03 where c1='a1' and c5='a5' order by c3,c2;
* 出现了filesort，我们建的索引是1234，它没有按照顺序来，3 2 颠倒了
* 1 有filesort
  + 9)
* explain select \* from test03 where c1='a1' and c2='a2' order by c2,c3;
* 10)
* 10.1 explain select \* from test03 where c1='a1' and c2='a2' and c5='a5' order by c2,c3; 用c1、c2两个字段索引，但是c2、c3用于排序,无filesort
* 10.2 explain select \* from test03 where c1='a1' and c2='a2' and c5='a5' order by c3,c2;
* 本例有常量c2的情况，和8.2对比
* where中添加了了c2，此时c2对应的是常量，所以order by c3,c2 真正起作用的只有c3
* 10.3 explain select \* from test03 where c1='a1' and c5='a5' order by c3,c2;
* 1 有filesort
  + 11)
* explain select \* from test03 where c1='a1' and c4='a4' group by c2,c3;
* 1 无filesort
* group by 也会默认进行排序
  + 12)
* explain select \* from test03 where c1='a1' and c4='a4' group by c3,c2;
* Using where; Using temporary; Using filesort
* 1 有filesort
* 因为group by 顺序与索引顺序不同，所以会产生临时表，并排序
* 分组之前会先排序

# MySQL性能优化

* 性能优化可以理解为在一定工作负载下尽可能地降低响应时间。
* 性能优化 不等于 提升QPS，这其实仅仅是吞吐量的优化。吞吐量的提升可以看做性能优化的副产品。因为每条查询执行时间变短，因此可以让服务器每秒执行更多的查询。
* 如果目标是降低响应时间，那么就需要测量时间花在什么地方。没有测量就没有调优。
* 一旦掌握并实践面向响应时间的优化方法，就会发现需要不断地对系统进行性能剖析（profiling)。性能剖析分为两个步骤：测量任务所花费的时间；对结果进行统计和排序，把重要的任务排在前面。
* MySQL的profile将最重要的任务展示在前面，但有时候没显示出来的信息也很重要。比如：
* 值得优化的查询：一些只占总响应时间比重很小的查询是不值得优化的。
* 异常情况：某些任务即使没有出现在profile输出的前面也需要优化。比如非常影响用户体验的某些任务，即使执行次数较少。

# MySQL查询优化

* 从效果上第一条影响最大，后面越来越小。
* ① SQL语句及索引的优化
* ② 数据库表结构的优化
* ③ 系统配置的优化
* ④ 硬件的优化

# 8.73 慢查询基础：优化数据访问

* 查询性能低下最基本的原因是访问的数据太多。某些查询可能不可避免地需要筛选大量数据，但这并不常见。大部分性能低下的查询都可以通过减少访问的数据量的方式进行优化。对于低效的查询，我们发现提供下面两个步骤来分析总是很有效。
* 1、确认应用程序是否在检索大量超过需要的数据，通常是访问了太多的行，但有时候也可能是访问了太多的列。
* 2、确认MySQL服务器层是否在分析大量超过需要的数据行。

## 是否向数据库请求了不需要的数据

- 1)查询不需要的记录：尽量使用LIMIT来获取所需的数据，而非取出全部数据然后在Application中获取某些行。  
- 2)多表关联时返回全部列  
- 3)总是取出全部列  
- 4)重复查询相同的数据：使用缓存

## MySQL是否在扫描额外的记录

* 对于MySQL，最简单的衡量查询开销的三个指标如下：
  + 1)响应时间
  + 2)扫描的行数
  + 3)返回的行数

### 响应时间

* 响应时间是两个部分的和：服务时间和排队时间。服务时间是指数据库处理这个查询真正花了多少时间。排队时间是指服务器因为等待某些资源而没有真正执行查询的时间——等待IO或等待锁。
* 在不同类型的应用压力下，响应时间并没有什么一致的规律或者公式。响应时间既可能是一个问题的结果也可能是一个问题的原因。
* 当你看到一个查询的响应时间的时候，首先需要问问自己，这个响应时间是否是一个合理的值。可以采用快速上限估计法来估算查询的响应时间：了解这个查询需要哪些索引以及它的执行计划是什么，然后计算大概需要多少个顺序和随机IO，再用其乘以在具体硬件条件下一次IO的消耗时间，最后把这些消耗都加起来，就可以获得一个大概参考值来判断当前响应时间是不是一个合理的值。

### 扫描的行数和返回的行数

* 分析查询时，查看该查询扫描的行数是非常有帮助的。这在一定程度上能够说明该查询找到需要的数据的效率高不高。
* 对于找出那些糟糕的查询，这个指标可能还不够完美，因为并不是所有的行的访问代价都是相同的。较短的行的访问速度更快，内存中的行也比磁盘中的行的访问速度要快得多。
* 理想情况下扫描的行数和返回的行数应该是相同的。一般扫描的行数对返回的行数的比率很小，一般在1:1和1:10之间。

### 扫描的行数和访问类型

* 在评估查询开销的时候，需要考虑一下从表中找到某一行数据的成本。MySQL有好几种访问方式可以查找并返回一行结果。有些访问方式可能需要扫描很多行才能返回一行结果，也有些访问方式可能无需扫描就能返回结果。
* 在EXPLAIN语句中的type列反映了访问类型。访问类型有全表扫描、索引扫描、范围扫描、唯一索引、常数引用等，速度由慢到快。
* 如果查询没有办法找到合适的访问类型，那么解决的最好方法通常就是增加一个合适的索引。
* 一般MytSQL能够使用如下三种方式应用WHERE条件，从好到坏依次为：
  + 1)在索引中使用WHERE条件来过滤不匹配的记录，这是在存储引擎层完成的
  + 2)使用覆盖索引（Extra中Using index)来返回记录，直接从索引中过滤不需要的记录并返回命中的结果，这是在MySQL服务器层完成的，但无需再回表查询记录
  + 3)从数据表中返回数据，然后过滤不满足条件的记录（Extra中Using Where)，这是在MySQL服务器层完成的，MySQL需要先从数据表读出记然后过滤。
* MySQL不会告诉我们生成结果实际上需要扫描多少行数据（例如关联查询结果返回的一条记录通过是由多条记录组成)，而只会告诉我们生成结果时一共扫描了多少行数据。扫描的行数中大部分都很可能是被WHERE条件过滤掉的，对最终的结果集没有贡献。
* 如果发现查询需要扫描大量的数据但只返回少数的行，那么通过可以尝试下面的技巧去优化它：
  + 1)使用覆盖索引
  + 2)改变表结构，例如使用单独的汇总表
  + 3)重写这个复杂的查询，让MySQL优化器能够以更优化的方式执行这个查询

# 8.74 重构查询的方式

## 一个复杂查询还是多个简单查询

* 传统实现中总是强调数据库层完成尽可能多的工作，因为以前认为网络通信、查询解析和优化是一件代价很高的事情。但是这样的想法对MySQL并不适用，MySQL从设计上让连接和断开连接都很轻量级，在返回一个小的查询结果方面很高效。
* MySQL内部每秒能够扫描内存中上百万行数据，相比之下，MySQL响应数据给客户端就慢得多了。在其他条件都相同时，使用尽可能少的查询当然是很好的。但是有时候，将一个大查询分解为多个小查询是很有必要的。

## 切分查询

* 有时候对于一个大查询我们需要分而治之，将大查询切分成小查询，每个查询功能完全一样，只完成一小部分，每次只返回一部分查询结果。
* 比如删除旧的数据，分批删除效率会高很多。

## 分解关联查询

* 很多高性能的应用都会对关联查询进行分解。可以对每一张表进行一次单表查询，然后将结果在应用程序中进行关联。
* 这样做的好处有：
  + 1)让缓存的效率更高。许多应用程序可以方便地缓存单表查询对应的结果对象。对MySQL的查询缓存来说，如果关联中的某张表发生了变化，那么就无法使用查询缓存了，而拆分后，如果某张表很少改变，那么基于该表的查询就可以重复利用缓存了。
  + 2)将查询分解后，执行单个查询可以减少锁的竞争
  + 3)在应用层做关联，可以更容易对数据库进行拆分
  + 4)查询本身效率也可能会有所提升。比如使用IN来代替JOIN，可以让MySQL按照ID顺序进行查询。
  + 5)可以减少冗余记录的查询。在数据库中做关联查询可能需要重复地访问一部分数据。
  + 6)相当于在应用中实现了哈希关联，而不是使用MySQL的嵌套循环关联。

# 8.75 优化特定类型的查询

## JOIN 优化

- 1)确保ON或者Using子句上的列有索引，在创建索引的时候就要考虑到关联的顺序。当表A和表B用列c关联时，如果优化器的关联顺序是B、A，那么就不需要在B表的对应列上建索引。一般情况下只需要在关联顺序的第二个表的相应列上创建索引。  
- 2)确保任何的GROUP BY和ORDER BY中的表达式只涉及到一个表中的列，这样MySQL才有可能使用索引来优化这个过程

## 小表驱动大表

* 两张表连接，类似于二重循环
* 外层的表应该是小表，内层的应该是大表
* 虽然总的遍历次数是一样的，但是频繁切换数据表是影响效率的（IO次数)，应该尽可能减少切换表的次数。
* A in B：
* for b in B:
* for a in A:
* if a == b:
* putIntoResultSet()
* A exists B：
* for a in A:
* for b in B:
* if a == b:
* putIntoResultSet()
* 所以，如果A是小表，B是大表时
* 如果用in，那么是B in A
* 如果用exists，那么是A exists B

## order by优化

### 尽量使用index方式排序，遵照索引的最佳左前缀

* CREATE TABLE tblA(
* #id int primary key not null auto\_increment,
* age INT,
* birth TIMESTAMP NOT NULL
* );
* CREATE INDEX idx*A*ageBirth ON tblA(age,birth);
* 排序时使用的字段的顺序最好与index建立的顺序相同
* 如果字段顺序不同，那么也会出现filesort
* 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。
* order a,b
* where a = xxx order by b

### 非索引列的filesort算法

* 问题：
* 在sort*buffer中单路排序比双路排序要多占用很多空间，因为单路排序是把所有字段都取出, 所以有可能取出的数据的总大小超出了sort*buffer的容量，导致每次只能取sort*buffer容量大小的数据，进行排序（创建tmp文件，多路合并)，排完再取sort*buffer容量大小，再排……从而多次I/O。
* 本来想省一次I/O操作，反而导致了大量的I/O操作，反而得不偿失。

### 优化策略

* 分组时也是需要order by

#### 1. Order by时select \* 是一个大忌

* 只取出需要的字段， 这点非常重要。在这里的影响是：
* 1.1 当Query的字段大小总和小于max*length*for*sort*data 而且排序字段不是 TEXT|BLOB 类型时，会用改进后的算法——单路排序， 否则用老算法——多路排序。
* 1.2 两种算法的数据都有可能超出sort*buffer的容量，超出之后，会创建tmp文件进行合并排序，导致多次I/O，但是用单路排序算法的风险会更大一些,所以要提高sort*buffer\_size。

#### 2. 尝试提高 sort*buffer*size

* 不管用哪种算法，提高这个参数都会提高效率，当然，要根据系统的能力去提高，因为这个参数是针对每个进程的

#### 3. 尝试提高 max*length*for*sort*data

* 提高这个参数， 会增加用改进算法的概率。但是如果设的太高，数据总容量超出sort*buffer*size的概率就增大，明显症状是高的磁盘I/O活动和低的处理器使用率.

### 总结

## group by 优化

* 当无法使用索引时，GROUP BY使用两种策略来完成：使用临时表或者文件排序来做分组。
* 当不遵照最佳左前缀，order by会出现filesort，而group by会出现临时表和filesort

## limit 优化

* 当偏移量非常大的时候，比如limit 1000,20 这样的查询，这时MySQL需要查询10020条记录然后只返回最后20条，这样的代价非常高。要优化这种查询，要么在页面中限制分页数量，要么优化大偏移的性能。
* 一个简单的办法是使用覆盖索引（延迟关联)
* 如果使用书签记录上次取数据的位置，那么下次就可以直接从该书签记录的位置开始扫描。
* 假设主键递增：
* 通过判断id的范围来分页
* select id,my*sn from big*data where id>5000000 limit 10;
* 也得到了分页的数据，但是我们发现如果id不是顺序的，也就是如果有数据删除过的话，那么这样分页数据就会不正确，这个是有缺陷的。

## UNION优化

* MySQL总是通过创建并填充临时表的方式来执行UNION查询。因此很多优化策略在UNION查询中都没法很好使用。经常需要手工将WHERE、LIMIT、ORDER BY等子句下推到UNION的各个子查询中，以便优化器可以充分利用这些条件进行优化。

# MySQL实现层次模型

# 8.76 邻接模型

* 属性：id,id\_parent,other fields
* 兄弟结点无序
* 特点：
  + 1)DML节点效率高，查询性能最高
  + 2)只支持单父节点
  + 3)递归实现
  + 4)删除子树较难

# 8.77 物化路径模型

* 属性：materialized\_path,other fields
  + PathID(1,1.1,1.2,1.1.1)
* 使用层次式的路径，明确地标识出来，一般用字符串存储路径，允许兄弟节点有序
* 特点：
  + 1)查询编写容易
  + 2)计算由路径导出的层次不方便
  + 3)会产生重复记录问题
  + 4)查询性能中等

# 8.78 嵌套集合模型

* 属性：left*num,right*num,other fields
* 每一个节点都有一个left*num和一个right*num。某节点的后代的left*num和right*num都会在该节点的left*num和right*num范围内。
* 特点：
  + 1)查找子节点容易，但是无法实现缩排
  + 2)适合DFS
  + 3)DML开销大，查询性能最低
* 分区分库分表
* 分区（针对表)
* 简介
* 数据表的物理存储拆分为多个文件
* 分区表是一个独立的逻辑表，其底层由多个物理子表组成。实现分区的代码实际上是对一组底层表的句柄对象（Handler Object)的封装。对分区表的请求，都会通过句柄对象转化成对存储引擎的接口调用。所以分区对于SQL层来说是一个完全封装底层实现的黑盒子，对应用是透明的。
* MySQL实现分区表的方式——对底层表的封装——意味着索引也是按照分区的子表定义的，而没有全局索引。
* MySQL在创建表时使用PARTITION BY子句定义每个分区存放的数据。在执行查询的时候，优化器会根据分区定义过滤那些没有我们需要数据的分区，这样查询就无须扫描所有分区——只需要查找包含需要数据的分区即可。
* 分区的一个目的是将数据按照一个较粗的粒度分在不同的表中。这样做可以将相关的数据存放在一起，另外，如果想一次批量删除整个分区的数据也会变得很方便。
* 分区非常适合在以下场景：
  + 1)表非常大以至于无法全部放在内存中，或者只在表的最后部分有热点数据，其他均为历史数据
  + 2)分区表的数据更容易维护。（批量删除数据->清除整个分区)
  + 3)分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备
  + 4)可以使用分区表来避免某些特殊的瓶颈。、比如InnoDB的单个索引的互斥访问，ext3文件系统的inode锁竞争。
  + 5)还可以备份和恢复独立的分区
* 分区表也有一些限制：
  + 1)一个表最多只能有1024个分区
  + 2)如果分区字段有主键或者唯一索引，那么所有主键列和唯一索引列都必须包含进来
  + 3)分区表中无法使用外键索引

## 原理

* 存储引擎管理分区的各个底层表和管理普通表一样，所有的底层表都必须使用相同的存储引擎，分区表的索引只是在各个底层表上各自加一个完全相同的索引。从存储引擎的角度，底层表和一个普通表没有任何不同。

### select

* 分区层先打开并锁住所有的底层表，优化器先判断是否可以过滤部分分区，然后再调用对应的存储引擎接口访问各个分区的数据。

### insert

* 当写入一条记录时，分区层先打开并锁住所有的底层表，然后确定哪个分区接收这条记录，再将记录写入对应底层表。

### delete

* 当删除一条记录时，分区层先打开并锁住所有的底层表，然后确定数据对应的分区，最后对相应底层表进行删除操作。

### update

* 当更新一条记录时，分区层先打开并锁住所有的底层表，MySQL先确定需要更新的记录在哪个分区，然后取出数据并更新，再判断更新后的数据应该放在哪个分区，最后对底层表进行写入操作，并对原数据所在的底层表进行删除操作。
* 虽然每个操作都会先打开并锁住所有的底层表，但这并不是分区表在处理过程中是锁住全表的，如果存储引擎能够自己实现行级锁，则会在分区层释放对应表锁，比如InnoDB，这个加锁和解锁的过程与普通InnoDB上的查询类似
* 分区类型

### Range分区

#### 原理

* MySQL将会根据指定的拆分策略，把数据放在不同的表文件上。相当于在文件上被拆成了小块.但是,对外给客户的感觉还是一张表，是透明的。

#### 案例

* CREATE TABLE tbl\_new(
* id INT NOT NULL PRIMARY KEY,
* title VARCHAR(20) NOT NULL DEFAULT ''
* )ENGINE MYISAM CHARSET utf8
* PARTITION BY RANGE(id)(
* PARTITION t0 VALUES LESS THAN(10),
* PARTITION t1 VALUES LESS THAN(20),
* PARTITION t2 VALUES LESS THAN(MAXVALUE)
* );
* 0~10放在t0
* 10~20放在t1
* 20放在t2
* 如果要查询id在20以上的，那么会直接去t2分区查找
* 如果插入的记录的id在20以上，那么会插入到t2分区
* 物理文件：
* 可以看出，普通的InnoDB引擎的表是一个frm和一个ibd文件
* 分区之后的MyIasm引擎的表有一个frm和par文件，此外每个分区还有一个myi和myd文件。
* frm：表的结构信息
* par：表的分区信息
* myi：表的索引信息
* myd：表的数据信息
* range的字段未必一定是id

### List分区

#### 原理

* MySQL中的LIST分区在很多方面类似于RANGE分区。和按照RANGE分区一样，每个分区必须明确定义。它们的主要区别在于，
* LIST分区中每个分区的定义和选择是基于某列的值从属于一个值列表集中的一个值，
* 而RANGE分区是从属于一个连续区间值的集合。

#### 案例

* create table area(
* id INT NOT NULL PRIMARY KEY,
* region varchar(20)
* )engine myisam charset utf8;
* insert into area values(1,'bj');
* insert into area values(2,'sh');
* insert into area values(3,'gz');
* insert into area values(4,'sz');
* 这个area的值是确定的
* create table user (
* uid int not null,
* userName varchar(20),
* area\_id int
* )engine myisam charset utf8
* partition by list(area\_id) (
  + partition bj values in (1),
  + partition sh values in (2),
  + partition gz values in (3),
  + partition sz values in (4)
* );
* User：

### 其他分区

#### Hash分区

#### Key分区

#### 子分区

## 分区表如何应用于大数据量

* 数据量超大时，肯定不能去全表扫描，并且B树索引也无法起作用，除非是覆盖索引。这正是分区要做的事情。理解分区时可以将其当做索引的最初形态，以代价非常小的方式定位到需要的数据在哪一片区域。在这篇区域中，可以做顺序扫描，可以建索引，还可以将数据都缓存到内存。
* 为了保证大数据量的可扩展性，一般有下面两个策略：
  + 1)全量扫描数据，不要加任何索引
* 可以使用简单的分区方式存放表，不要任何索引，根据分区的规则大致定位需要的数据位置。只要能够使用WHERE条件，将需要的数据限制在少数分区中，则效率是很高的。
  + 2)索引数据，并分离热点
* 如果数据有明显的热点，而且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独放在一个分区中，让这个分区的数据能够有机会都缓存在内存中。这样查询就可以只访问一个很小的分区表，能够使用索引，也能够有效地使用缓存。
* 分区表的陷阱

### NULL值会使分区过滤无效

* 分区的表达式的值可以是NULL；第一个分区是一个特殊分区，如果表达式的值为NULL或非法制，记录都会被存放到第一个分区。WHERE查询时即使看起来可以过滤到只有一个分区，但实际会检查两个分区，即第一个分区。最好是设置分区的列为NOT NULL。

### 分区列和索引列不匹配

* 如果定义的索引列和分区列不匹配，会导致索引无法进行分区过滤。
* 假设在列a上定义了索引，而在列b上进行分区。因为每个分区都有其独立的索引，所以扫描b上的索引就需要扫描每一个分区内对应的索引。

### 选择分区的成本可能很高

* 尤其是范围分区，对于回答“这一行属于哪个分区”、“这些符合查询条件的行在哪些分区”这样的问题的成本可能会非常高。其他的分区类型，比如键分区和哈希分区，就没有这样的问题。
* 在批量插入时问题尤其严重。

### 其他限制

- 1)每个分区都必须使用同样的存储引擎  
- 2)分区函数中可以使用的函数和表达式也有一些限制  
- 3)某些存储引擎不支持分区  
- 4)对应MyISAM表，使用分区表时需要打开更多的文件描述符。有可能出现茶瓯go文件描述符限制的问题。

## 查询优化

* 对于访问分区表来说，很重要的一点是要在WHERE条件中加入分区列，有时候即使看似多余的也要带上，这样就可以让优化器能够过滤无须访问的分区。
* 使用EXPLAIN PARTITION可以观察优化器是否执行而来分区过滤。
* 分库（针对库)
* 简介
* 一个库里表太多了，导致了海量数据，系统性能下降，把原本存储于一个库的表拆分存储到多个库上，通常是将表按照功能模块、关系密切程度划分出来，部署到不同库上。
* 将一个数据库里的表拆分到多个数据库（主机)中，形成数据库集群
* 比如分为一个静态信息库（基本没有写入)和一个业务相关的库（频繁写入)
* 为什么要分库
* 数据库集群环境后都是多台slave,基本满足了读取操作;
* 但是写入或者说大数据、频繁的写入操作对master性能影响就比较大，
* 这个时候，单库并不能解决大规模并发写入的问题。
* 优点
* 减少增量数据写入时的锁对查询的影响。
* 由于单表数量下降，常见的查询操作由于减少了需要扫描的记录，使得单表单次查询所需的检索行数变少，减少了磁盘IO，时延变短。
* 但是它无法解决单表数据量太大的问题。
* 分表（针对表)
* 简介
* 水平拆分（行)
* 类似于Range分区
* 一张表有很多数据时，将数据分到多张表中
* MySQL单表的容量不超过500W（300W就需要拆分)，否则建议水平拆分
* 垂直拆分（列)
* 比如有些表会有大量的属性
* 将一些相关的属性拆分到一张单独的表
* 垂直分表，
* 通常是按照业务功能的使用频次，把主要的、热门的字段放在一起做为主要表；
* 然后把不常用的，按照各自的业务属性进行聚集，拆分到不同的次要表中；主要表和次要表的关系一般都是一对一的。
* 冷数据放到主要表中，热数据放到次要表中
* 使用
* 切分策略：需要DBA参与研究
* 导航路由：查询时，怎么导航到哪一张表
* 开源方案
* MySQL Fabric（官方)
* 网址：http://www.MySQL.com/products/enterprise/fabric.html
* MySQL Fabric 是一个用于管理 MySQL 服务器群的可扩展框架。该框架实现了两个特性 — 高可用性 (HA) 以及使用数据分片的横向扩展
* 官方推荐，但是2014年左右才推出，是真正的分表，不是代理的(不同于MySQL-proxy)。
* 未来很有前景，目前属于测试阶段还没大规模运用于生产,期待它的升级。
* Atlas（奇虎360)
* Atlas是由 Qihoo 360, Web平台部基础架构团队开发维护的一个基于MySQL协议的数据中间层项目。
* 它在MySQL官方推出的MySQL-Proxy 0.8.2版本的基础上，修改了大量bug，添加了很多功能特性。目前该项目在360公司内部得到了广泛应用，很多MySQL业务已经接入了Atlas平台，每天承载的读写请求数达几十亿条。
* 主要功能：
  + 读写分离
  + 从库负载均衡
  + IP过滤
  + SQL语句黑白名单
  + 自动分表，只支持单库多表，不支持分布式分表，同理，该功能我们可以用分库来代替，多库多表搞不定。
* 网址： https://github.com/Qihoo360/Atlas
* TDDL（阿里)
* 江湖外号：头都大了
* 淘宝根据自己的业务特点开发了TDDL（Taobao Distributed Data Layer )框架，主要解决了分库分表对应用的透明化以及异构数据库之间的数据复制，它是一个基于集中式配置的 jdbc datasource实现，具有主备，读写分离，动态数据库配置等功能。
* TDDL所处的位置（tddl通用数据访问层，部署在客户端的jar包，用于将用户的SQL路由到指定的数据库中)：
* 淘宝很早就对数据进行过分库的处理， 上层系统连接多个数据库，中间有一个叫做DBRoute的路由来对数据进行统一访问。DBRoute对数据进行多库的操作、数据的整合，让上层系统像操作 一个数据库一样操作多个库。但是随着数据量的增长，对于库表的分法有了更高的要求，例如，你的商品数据到了百亿级别的时候，任何一个库都无法存放了，于是 分成2个、4个、8个、16个、32个……直到1024个、2048个。好，分成这么多，数据能够存放了，那怎么查询它？这时候，数据查询的中间件就要能够承担这个重任了，它对上层来说，必须像查询一个数据库一样来查询数据，还要像查询一个数据库一样快（每条查询在几毫秒内完成)，TDDL就承担了这样一 个工作。在外面有些系统也用DAL（数据访问层) 这个概念来命名这个中间件。
* 系出名门，淘宝诞生。功能强大，阿里开源（部分)
* 主要优点：
* 1.数据库主备和动态切换
* 2.带权重的读写分离
* 3.单线程读重试
* 4.集中式数据源信息管理和动态变更
* 5.剥离的稳定jboss数据源
* 6.支持MySQL和oracle数据库
* 7.基于jdbc规范，很容易扩展支持实现jdbc规范的数据源
* 8.无server,client-jar形式存在，应用直连数据库
* 9.读写次数,并发度流程控制，动态变更
* 10.可分析的日志打印,日志流控，动态变更
* TDDL必须要依赖diamond配置中心（diamond是淘宝内部使用的一个管理持久配置的系统，目前淘宝内部绝大多数系统的配置，由diamond来进行统一管理，同时diamond也已开源)。
* TDDL动态数据源使用示例说明：http://rdc.taobao.com/team/jm/archives/1645
* diamond简介和快速使用：http://jm.taobao.org/tag/diamond%E4%B8%93%E9%A2%98/
* TDDL源码：https://github.com/alibaba/tb\_tddl
* TDDL复杂度相对较高。当前公布的文档较少，只开源动态数据源，分表分库部分还未开源，还需要依赖diamond，不推荐使用。
* MySQL Proxy（官方)
* 官网提供的，小巧精干型的，但是能力有限，对于大数据量的分库分表无能为力，适合中小型的互联网应用，基本上MySQL-proxy - master/slave就可以构成一个简单版的读写分离和负载均衡
* 总结
* 分库分表演变过程
* 单库多表--->读写分离主从复制--->垂直分库，每个库又可以带着slave--->继续垂直分库，极端情况单库单表--->分区(变相的水平拆分表，只不过是单库的)--->水平分表后再放入多个数据库里，进行分布式部署

- 单库多表  
- 读写分离主从复制  
- 垂直分库（每个库又可以带salve)  
- 继续垂直分库，理论上可以到极端情况，单库单表  
- 分区（partition是变相的水平拆分，只不过是单库内进行)  
- 终于到水平分表，后续放入多个数据库里，进行分布式部署，终极method。  
- 但是理论上OK，实际上中间的各种通信、调度、维护和编码要求，更加高。

* 分库分表后的难题

- 分布式事务的问题，数据的完整性和一致性问题。  
- 数据操作维度问题：用户、交易、订单各个不同的维度，用户查询维度、产品数据分析维度的不同对比分析角度。  
- 跨库联合查询的问题，可能需要两次查询  
- 跨节点的count、order by、group by以及聚合函数问题，可能需要分别在各个节点上得到结果后在应用程序端进行合并  
- 额外的数据管理负担，如：访问数据表的导航定位  
- 额外的数据运算压力，如：需要在多个节点执行，然后再合并计算  
- 程序编码开发难度提升，没有太好的框架解决，更多依赖业务看如何分，如何合，是个难题。

* 不到最后一步不要轻易水平分表

# 主从复制

# 8.79 复制概述

* 复制解决的基本问题是让一台服务器的数据与其他服务器保持同步。一台主库的数据可以同步到多台备库上，备库本身也可以被配置成另外一台服务器的主库。
* MySQL支持两种复制方式：基于行的复制和基于语句的复制。这两种方式都是通过在主库上记录binlog，在备库重放日志的方式来实现异步的数据复制。这意味着，在同一时间点备库上的数据可能与主库存在不一致，并且保证主备之间的延迟。
* 复制通常不会增加主库的开销，主要是启用binlog带来的开销，但出于备份或及时从崩溃中恢复的目的，这点开销也是必要的。除此之外，每个备库也会对主库增加一些负载（网络IO)，尤其当备库请求从主库读取旧的binlog时，可能会造成更高的IO开销。
* 通过复制可以将读操作指向备库来获得更好的读扩展，但对于写操作，除非设计得当，否则并不适合通过复制来扩展写操作。

## 复制解决的问题

### 数据分布

* 可以在不同地址位置来分布数据备份，例如不同的数据中心，即使在不稳定的网络环境下，远程复制也可以工作。

### 负载均衡

* 可以将读操作分布到多个服务器上，实现对读密集型应用的优化，并且实现方便。

### 备份

* 对于备份来说，复制是一项很有意义的技术补充，但复制既不是备份，也不能取代备份。

### 高可用和故障切换

* 复制能够帮助应用程序避免MySQL单点失败，一个包含复制的设计良好的故障切换系统能够显著地缩短宕机实现。

## 复制如何工作

* 1、在主库上把数据更改记录在binlog中（这些记录称为二进制日志事件)
* 2、备库将主库上的日志复制到自己的中继日志中
* 3、备库读取中继日志中的事件，将其重放到备库数据之上

# 8.80 复制原理

## 基于语句的复制

* 主库会记录那些造成数据更改的查询，当备库读取并重放这些事件时，实际上只是把主库上执行过的SQL再执行一遍。
  + 优点：1)实现简单2)binlog中的事件更加紧凑
* 问题：
  + 1)同一条SQL在主库和备库上执行的时间可能稍微或很不相同，因此在传输的binlog中，除了SQL，还有一些元数据，比如时间戳
  + 2)一些无法被正确复制的SQL，存储过程、触发器
  + 3)更新必须是串行的，这需要更多的锁

## 基于行的复制

* 会将实际数据记录在binlog中。
  + 好处：1)可以正确地复制每一行，一些语句可以被更加有效地复制
  + 2)复制更加高效（但也视情况而定)

## 比较

* 理论上基于行的复制方式整体上更有效，并且在实际应用中也适用于大多数场景。

# 8.81 复制拓扑

* 可以在任意个主库和备库之间建立复制，只有一个限制：每一个备库只能有一个主库。
* 每个备库必须有一个唯一的服务器ID；一个主库可以有多个备库；如果打开了log*slave*updates选项，一个备库可以把其主库上的数据变化传播到其他备库。
  + 1)一主库多备库
  + 2)主动-主动模式下的主主复制 / 双主复制
* 每一个都被配置成对方的主库和备库。
* 这种配置最大的问题是如何解决冲突，比如两台服务器同时修改一行记录，或同时往两台服务器上向一个包含AUTO\_INCREMENT列的表里插入数据。
* 使用起来非常麻烦
  + 3)主动-被动模式下的主主复制
* 主要区别在于其中的一台服务器是只读的被动服务器。
* 这种方式使得反复切换主动和被动服务器非常方便，因为服务器的配置是对称的，这使得故障转移和故障恢复很容易。
* 设置主动-被动的主主复制在某种意义上类似于创建一个热备份，但是可以使用这个备份来提高性能。比如用它来执行读操作、备份、离线维护以及升级。真正的热备份是做不了这些事情的，然而，你不会获得比单台服务器更好的写性能。
  + 4)拥有备库的主主结构
* 为每个主库都增加一个备库，增加了冗余，可以将读查询分配到备库上。
  + 5)环形复制：环形结构可以有三个或更多的主库，每个服务器都是它之前的服务器的备库，是在它之后的服务器的主库。
* 环形结构没有双主结构的一些优点，比如对称配置和简单的故障转移，并完全依赖于环上的每一个可用结点，这大大增加了整个系统失效的几率。
* 可以为每个节点增加备库的方式来减少环形复制的风险。

# 8.82 复制和容量规划

## 为什么复制无法扩展写操作

* 糟糕的服务容量比例的根本原因是不能像分发读操作那样把写操作等同地分发到更多服务器上。
* 分区是唯一可以扩展写入的方法（分库分表？)

## 备库什么时候开始延迟

* 为了预测在将来的某个时间点会发生什么，可以人为地之制造延迟，然后看多久备库能赶上主库。

## 规划冗余容量

* 在构建一个大型应用时，有意让服务器不被充分使用，这应该是一种聪明并且划算的方式，尤其是在使用复制的时候，有多余容量的服务器可以更好地处理负载尖峰，也有更多能力处理慢速查询和维护工作（OPTIMIZE TABLE)，并且能更好跟上复制。

# 8.83 复制管理和维护

## 监控复制

* 在主库上，可以使用SHOW MASTER STATUS 来查看当前主库的binlog位置和配置，还可以查看主库当前有哪些binlog是在磁盘上的。

## 测量备库延迟

* 虽然SHOW SLAVE STATUS输出的Seconds*behind*master列理论上显示了备库的延时，但由于各种各样的原因，并不总是准确的：
  + 1)Seconds*behind*master是通过将备库服务器当前时间戳与binlog中事件的时间戳相对比得到的，所以只有在执行事件时才能报告延迟
  + 2)如果备库复制线程没有运行，就会报延迟为NULL
  + 3)一些错误（网络不稳定)可能中断复制/停止复制线程，但Seconds*behind*master将显示为0而不是显示错误
  + 4)即使备库线程正在运行，备库有时候无法计算延时
  + 5)一个大事务可能会导致延迟波动。
* 可以使用一些其他方法来监控备库延迟，比如heartbeat record，这是一个在主库上每秒更新一次的时间戳。为了计算延时，可以直接用备库当前的时间戳减去心跳记录的值。

## 确定主备是否一致

* 主备经常会因为MySQL的bug、网络中断、服务器崩溃导致不一致。
* 应该经常性地检查主备是否一致。可以使用一些工具去计算表和数据的checksum。

## 从主库重新同步备库

* 传统方法是关闭备库，然后重新从主库复制一份数据。但当数据量很大时，如果能够找出并修复不一致的数据，要比从其他服务器上重新复制数据要有效的多。
* 最简单的办法是使用mysqldump转储受影响的数据并重新导入。如果数据没有发生变化，这种方法会很好，可以在主库上将表锁住然后转储，再等待备库赶上主库，然后将数据导入到备库中。缺点是在一个繁忙的服务器上可能行不通，另外在备库上通过非复制的方式改变数据可能不够安全。

## 改变主库

* 只需在备库中简单地使用CHANGE MASTER TO命令，并指定合适的值。整个过程最难的是获取新主库上合适的binlog的位置，这样备库才可以从和老主库相同的逻辑位置开始复制。

# 8.84 复制的问题和解决方案

## 数据损坏或丢失

* 意外关闭服务器时可能会遇到的情况：
  + 1)主库意外关闭：在崩溃前没有将最后几个binlog事件刷新到磁盘中，备库IO线程因此一直处于读不到尚未写入磁盘的事件的状态中。当主库重新启动时，备库将重连到主库逼格再次尝试去读该事件，但主库会告诉备库没有这个binlog偏移量。
* 解决方法是指定备库从下一个binlog的开头读日志，但一些日志事件将永久地丢失。可以通过在主库开启sync\_binlog来避免事件丢失。
  + 2)备库意外关闭：如果使用的都是InnoDB表，可以在重启后观察MySQL错误日志。InnoDB在恢复过程中会打印出它的恢复点的binlog坐标。可以使用这个值来决定备库指向胡库的偏移量。

## 使用非事务型表

* 基于语句的复制通过能够很好地处理非事务表。但是当对非事务型表的更新发生错误时，就可能导致主库和备库的数据不一致。如果使用的是MyIASM表，在关闭MySQL之前需要确保已经运行了STOP SLAVE，否则服务器在关闭时会kill所有正在运行的查询。事务型存储引擎则没有这个问题，如果使用的是事务型表，失败的更新会在主库上回滚并且不会记录到binlog中。

## 混合事务型和非事务型表

* 如果使用的是事务型存储引擎，只有在事务提交时才会查询记录到binlog中。因此如果事务回滚，MySQL就不会记录这条查询，也就不会在备库重放。
* 但是如果混合使用事务型和非事务型表，并且发生了一次回滚，MySQL能够回滚事务型表的更新，但非事务型表就会被永久地更新了。
* 防止该问题的唯一办法是避免混合使用事务型和非事务型表。如果遇到这个问题，唯一的解决办法是忽略错误，并重新同步相关的表。
* 基于行的复制不会受这个问题影响。

## 不确定语句

* 当使用基于语句的复制模式时，如果通过不确定的方式更改数据可能会导致主备不一致。
* 基于行的复制则没有上述限制。

## 主库和备库使用不同的存储引擎

* 当使用基于语句的复制方式时，如果备库使用了不同的存储引擎，则可能造成一条查询在主库和备库上的执行结果不同。

## 备库发生数据改变

* 基于语句的复制方式前提是确保备库上有和主库相同的数据，因此不应该允许对备库数据的任何更改。
* 唯一的解决办法是重新从主库同步数据。

## 不唯一的服务器ID

## 未定义的服务器ID

## 对未复制数据的依赖性

## 丢失的临时表

## 不复制所有的更新

## InnoDB加锁读引起的锁争用

## 过大的复制延迟

## 来自主库过大的包

## 受限制的复制带宽

## 磁盘空间不足

## 复制的局限性

# 高可用解决方案

* scale-up 向上扩展/垂直扩展：购买更多性能更强的硬件。容易达到性能瓶颈。
* scale-out 向外扩展：复制、拆分、数据分片
* 比如按业务分库；单表分区

# 8.85 脑裂问题

* 在心跳失效的时候，就发生了脑裂（split-brain)。
* （ 一种常见的脑裂情况可以描述如下)比如正常情况下，（集群中的)NodeA 和 NodeB 会通过心跳检测以确认对方存在，在通过心跳检测确认不到对方存在时，就接管对应的（共享) resource 。如果突然间，NodeA 和 NodeB 之间的心跳不存在了（如网络断开)，而 NodeA 和 NodeB 事实上却都处于 Active 状态，此时 NodeA 要接管 NodeB 的 resource ，同时 NodeB 要接管 NodeA 的 resource ，这时就是脑裂（split-brain)。
* 脑裂（split-brain)会 引起数据的不完整性 ，并且可能会对服务造成严重影响 。
* 起数据的不完整性主要是指，集群中节点（在脑裂期间)同时访问同一共享资源，而此时并没有锁机制来控制针对该数据访问，那么就存在数据的不完整性的可能。
* 对付 HA 系统“裂脑”的对策 ，目前我所了解的大概有以下几条：
  + 1)添加冗余的心跳线。例如双“心跳线”。尽量减少“脑裂”发生机会。
  + 2)启用磁盘锁。正在服务一方锁住共享磁盘，“脑裂”发生时，让对方完全“抢不走”共享磁盘资源。但使用锁磁盘也会有一个不小的问题，如果占用共享盘的一方不主动解锁，另一方就永远得不到共享磁盘。现实中假如服务节点突然死机或崩溃，就不可能执行解锁命令。后备节点也就接管不了共享资源和应用服务。于是有人在 HA 系统中设计了“智能”锁。即正在服务的一方只在发现“心跳线”全部断开（察觉不到对端)时才启用磁盘锁。平时就不上锁了。
  + 3)设置仲裁机制。例如设置参考 IP（如网关IP)Monitor，当心跳线完全断开时，2 个节点都各自 ping 一下 参考 IP ，不通则表明断点就出在本端，不仅“心跳线”断了、对外提供“服务”的本端网络链路也路断了，即使启动（或继续)应用服务也没有用了，那就主动放弃竞争，让能够 ping 通参考 IP 的一端去起服务。更保险一些，ping 不通参考 IP 的一方干脆就自我重启，以彻底释放有可能还占用着的那些共享资源。

# 8.86 解决方案

* LVS+Keepalived+MySQL（有脑裂问题？但似乎很多人推荐这个)
* DRBD+Heartbeat+MySQL（有一台机器空余？Heartbeat切换时间较长？有脑裂问题？)MySQL Proxy（不够成熟与稳定？使用了Lua？是不是用了他做分表则可以不用更改客户端逻辑？)
* MySQL Cluster （社区版不支持INNODB引擎？商用案例不足？)
* MySQL + MHA （如果配上异步复制，似乎是不错的选择，又和问题？)
* MySQL + MMM （似乎反映有很多问题，未实践过，谁能给个说法)

# 8.87 MHA

* MHA（Master High Availability)目前在MySQL高可用方面是一个相对成熟的解决方案，它由日本DeNA公司youshimaton（现就职于Facebook公司)开发，是一套优秀的作为MySQL高可用性环境下故障切换和主从提升的高可用软件。在MySQL故障切换过程中，MHA能做到在0~30秒之内自动完成数据库的故障切换操作，并且在进行故障切换的过程中，MHA能在最大程度上保证数据的一致性，以达到真正意义上的高可用。
* 该软件由两部分组成： MHA Manager（管理节点)和MHA Node（数据节点)。MHA Manager可以单独部署在一台独立的机器上管理多个master-slave集群，也可以部署在一台slave节点上。MHA Node运行在每台MySQL服务器上，MHA Manager会定时探测集群中的master节点，当master出现故障时，它可以自动将最新数据的slave提升为新的master，然后将所有其他的slave重新指向新的master。整个故障转移过程对应用程序完全透明。
* 在MHA自动故障切换过程中，MHA试图从宕机的主服务器上保存二进制日志，最大程度的保证数据的不丢失，但这并不总是可行的。例如，如果主服务器硬件故障或无法通过ssh访问，MHA没法保存二进制日志，只进行故障转移而丢失了最新的数据。使用MySQL 5.5的半同步复制，可以大大降低数据丢失的风险。MHA可以与半同步复制结合起来。如果只有一个slave已经收到了最新的二进制日志，MHA可以将最新的二进制日志应用于其他所有的slave服务器上，因此可以保证所有节点的数据一致性。
* 目前MHA主要支持一主多从的架构，要搭建MHA,要求一个复制集群中必须最少有三台数据库服务器，一主二从，即一台充当master，一台充当备用master，另外一台充当从库，因为至少需要三台服务器，

- （1)从宕机崩溃的master保存二进制日志事件（binlog events);  
- （2)识别含有最新更新的slave；  
- （3)应用差异的中继日志（relay log)到其他的slave；  
- （4)应用从master保存的二进制日志事件（binlog events)；  
- （5)提升一个slave为新的master；  
- （6)使其他的slave连接新的master进行复制；

# 8.88 MMM

* MMM(Master-Master replication managerfor Mysql，Mysql主主复制管理器)是一套灵活的脚本程序，基于perl实现，用来对mysql replication进行监控和故障迁移，并能管理mysql Master-Master复制的配置(同一时间只有一个节点是可写的)。
* mmm\_mond：监控进程，负责所有的监控工作，决定和处理所有节点角色活动。此脚本需要在监管机上运行。
* mmm\_agentd：运行在每个mysql服务器上的代理进程，完成监控的探针工作和执行简单的远端服务设置。此脚本需要在被监管机上运行。
* mmm*control：一个简单的脚本，提供管理mmm*mond进程的命令。
* mysql-mmm的监管端会提供多个虚拟IP（VIP)，包括一个可写VIP，多个可读VIP，通过监管的管理，这些IP会绑定在可用mysql之上，当某一台mysql宕机时，监管会将VIP迁移至其他mysql。
* 在整个监管过程中，需要在mysql中添加相关授权用户，以便让mysql可以支持监理机的维护。授权的用户包括一个mmm*monitor用户和一个mmm*agent用户，如果想使用mmm的备份工具则还要添加一个mmm\_tools用户。
* 优点：高可用性，扩展性好，出现故障自动切换，对于主主同步，在同一时间只提供一台数据库写操作，保证的数据的一致性。
* 缺点：Monitor节点是单点，可以结合Keepalived实现高可用。

# 压力测试

* sysbench是一个模块化的、跨平台、多线程基准测试工具，主要用于评估测试各种不同系统参数下的数据库负载情况。关于这个项目的详细介绍请看：http://sysbench.sourceforge.net。
* 它主要包括以下几种方式的测试：
* 1、cpu性能
* 2、磁盘io性能
* 3、调度程序性能
* 4、内存分配及传输速度
* 5、POSIX线程性能
* 6、数据库性能(OLTP基准测试)
* 目前sysbench主要支持 MySQL,pgsql,oracle 这3种数据库。
* 一、安装
* 首先，在 http://sourceforge.net/projects/sysbench 下载源码包。
* 接下来，按照以下步骤安装：
* tar zxf sysbench-0.4.8.tar.gz
* cd sysbench-0.4.8
* ./configure && make && make install
* strip /usr/local/bin/sysbench
* 以上方法适用于 MySQL 安装在标准默认目录下的情况，如果 MySQL 并不是安装在标准目录下的话，那么就需要自己指定 MySQL 的路径了。比如我的 MySQL 喜欢自己安装在 /usr/local/mysql 下，则按照以下方法编译：
* ./configure --with-mysql-includes=/usr/local/mysql/include --with-mysql-libs=/usr/local/mysql/lib && make && make install
* 当然了，用上面的参数编译的话，就要确保你的 MySQL lib目录下有对应的 so 文件，如果没有，可以自己下载 devel 或者 share 包来安装。
* 另外，如果想要让 sysbench 支持 pgsql/oracle 的话，就需要在编译的时候加上参数
* --with-pgsql
* 或者
* --with-oracle
* 这2个参数默认是关闭的，只有 MySQL 是默认支持的。
* 二、开始测试
* 编译成功之后，就要开始测试各种性能了，测试的方法官网网站上也提到一些，但涉及到 OLTP 测试的部分却不够准确。在这里我大致提一下：
* 1、cpu性能测试
* sysbench --test=cpu --cpu-max-prime=20000 run
* cpu测试主要是进行素数的加法运算，在上面的例子中，指定了最大的素数为 20000，自己可以根据机器cpu的性能来适当调整数值。
* 2、线程测试
* sysbench --test=threads --num-threads=64 --thread-yields=100 --thread-locks=2 run
* 3、磁盘IO性能测试
* sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw prepare
* sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw run
* sysbench --test=fileio --num-threads=16 --file-total-size=3G --file-test-mode=rndrw cleanup
* 上述参数指定了最大创建16个线程，创建的文件总大小为3G，文件读写模式为随机读。
* 4、内存测试
* sysbench --test=memory --memory-block-size=8k --memory-total-size=4G run
* 上述参数指定了本次测试整个过程是在内存中传输 4G 的数据量，每个 block 大小为 8K。
* 5、OLTP测试
* sysbench --test=oltp --mysql-table-engine=myisam --oltp-table-size=1000000 \
* --mysql-socket=/tmp/mysql.sock --mysql-user=test --mysql-host=localhost \
* --mysql-password=test prepare
* 上述参数指定了本次测试的表存储引擎类型为 myisam，这里需要注意的是，官方网站上的参数有一处有误，即 --mysql-table-engine，官方网站上写的是 --mysql-table-type，这个应该是没有及时更新导致的。另外，指定了表最大记录数为 1000000，其他参数就很好理解了，主要是指定登录方式。测试 OLTP 时，可以自己先创建数据库 sbtest，或者自己用参数 --mysql-db 来指定其他数据库。--mysql-table-engine 还可以指定为 innodb 等 MySQL 支持的表存储引擎类型。

# 容灾备份

# 8.89 为什么要备份

- 1)灾难恢复  
- 2)人们改变刑法  
- 3)审计  
- 4)测试

# 8.90 设计备份方案

* 建议：
  + 1)在生产实践中，对于大数据量来说，物理备份还必需的：逻辑备份太慢并受到资源限制，从逻辑备份中恢复需要很长时间。基于快照的备份是最好的选择。对于较小的数据库，逻辑备份可以很好地胜任。
  + 2)保留多个备份集
  + 3)定期从逻辑备份/物理备份中抽取数据进行恢复测试
  + 4)保存binlog以用于基于故障时间点的回复
  + 5)完全不借助备份工具本身来监控本分和备份的过程，需要另外验证备份是否正常
  + 6)通过演练整个恢复过程来测试备份和恢复，测算恢复所需要的资源

## 在线备份还是离线备份

* 如果可能，关闭MySQL做备份是最简单安全的，也是所有获取一致性副本中最好的，而且损坏或不一致的风险最小。
* 由于一致性的需要，对服务器进行在线备份仍然会有明显的服务中断。
* 最大的权衡是备份时间和备份负载，可以牺牲其一以增强另外一个。

## 逻辑备份还是物理备份

* 有两种主要的方法来备份MySQL数据：逻辑备份（导出)和直接复制原始文件的物理备份。
* 逻辑备份有如下优点：
  + 1)是可以用编辑器或grep、sed查看和操作的普通文件，当需要回复数据或者只查看数据时都非常有帮助
  + 2)恢复非常简单
  + 3)可以通过网络来备份和恢复
  + 4)可以在不能访问底层文件系统的系统中使用
  + 5)非常灵活，因为mysqldump可以接受很多选项。
  + 6)与存储引擎无关。
  + 7)有助于避免数据损坏。
* 缺点：
  + 1)必须由数据库服务器完成生成逻辑备份的工作，要是要更多的CPU周期
  + 2)在某些场景下比数据库文件本身更大
  + 3)无法保证导出后再还原出来的一定是同样的数据
  + 4)效率较低
* 最大的缺点是从MySQL中导出数据和通过SQL语句将其加载回去的开销。如果使用逻辑备份，测试恢复需要的时间将非常重要。
* 物理备份有如下好处：
  + 1)只需要将需要的文件复制到其他地方即可完成备份，不需要额外的工作
  + 2)恢复更加简单，并且更快
* 缺点：
  + 1)原始文件通常比相应的逻辑备份要大得多
  + 2)物理备份不总是可以跨平台、操作系统和MySQL版本。
* 简单地说，逻辑备份不易出错，物理备份更加简单高效

## 备份什么

* 非显著数据，比如binlog和redo log
* 代码
* 复制配置
* 服务器配置
* 选定的操作系统文件

## 存储引擎和一致性

* 数据一致性：InnoDB的MVCC可以帮到我们。开始一个事务，转储一组相关的表，然后提交事务。只要在服务器上使用RR事务隔离级别，并且没有任何DDL，就一定会有完美的一致性，以及基于时间点的数据快照，且在备份过程中不会阻塞任何后续的工作。也可以用mysqldump来获得InnoDB表的一致性逻辑备份。可能会导致一个非常长的事务，在某些负载下会导致开销大到不可接受。
* 文件一致性：InnoDB如果检测到数据不一致或损坏，会记录错误日志乃至让服务器崩溃。

# 8.91 管理和备份binlog

* 服务器的binlog是备份的最重要因素之一，它们对于基于时间点的恢复是必需的，并且通常比数据要小，所以更容易进行频繁的备份。如果有某个时间点的数据备份和所有从那时以后的binlog，就可以重放自从上次全备以来的binlog并前滚所有的变更。

# 8.92 备份数据

## 生成逻辑备份

### SQL导出

* SQL导出是很多人所熟悉的，因为它们是mysqldump默认的方式。
* 缺点：
  + 1)Schema和数据存储在一起
  + 2)巨大的SQL语句
  + 3)单个巨大的文件

### 符号分隔文件备份

* 可以使用SQL命令SELECT INTO OUTFILE以符号分隔文件格式创建数据的逻辑备份（mysqldumo的—tab选项)。符号分隔文件包含以ASCII展示的原始数据，没有SQL、注释和列名。
* 比起SQL导出文件，符号分隔文件要更紧凑且更易于用命令行工具操作，最大的优点是备份和还原速度更快。
* LOAD DATA INFILE方法可以加载数据到表中。

## 文件系统快照

* 文件系统快照是一种非常好的在线备份方法，支持快照的文件系统能够瞬间创建用来备份的内容一致的进行。支持快照的文件系统和设备包括FreeBSD的文件系统、ZFS文件系统、LVM、以及许多的SAN系统和文件存储解决方案。

### LVM快照

# 8.93 从备份中恢复

# SQL

* 分页
* 带日期
* 要加上nextkey锁，语句该怎么写
* 各种join
* like%..%为什么会扫描全表？遵循什么原则？
* sql语句各种条件的执行顺序，如select， where， order by， group by

# 8.94 执行顺序

* for human：
* for machine：SQL解析器执行顺序
* from on join where group by having select distinct order by limit

# 8.95 连接

* 7种：  
  - 内连接  
  - 左外连接  
  - 右外连接  
  - 全外连接（内连接+左外连接+右外连接)  
  - 左外连接 – 内连接  
  - 右外连接 – 内连接  
  - 全外连接 – 内连接
* 实际的SQL：
* emp:8
* dept:5

## 笛卡尔积/交叉连接

* 40

## 内连接

* 7

## 左外连接

* 8

## 右外连接

* 8

## 全外连接

* MySQL不支持，一种替代做法是：

## 左外连接 – 内连接

* 1

## 右外连接 – 内连接

* 1

## 全外连接 – 内连接

* 2

# MySQL底层实现

* 查询处理与查询优化过程
* MySQL的查询流程大致是：
* MySQL客户端通过协议与MySQL服务器建立连接，发送查询语句，先检查查询缓存，如果命中，直接返回结果，否则进行语句解析
* 有一系列预处理，比如检查语句是否写正确了，然后是查询优化（比如是否使用索引扫描，如果是一个不可能的条件，则提前终止)，生成查询计划，然后查询引擎启动，开始执行查询，从底层存储引擎调用API获取数据，最后返回给客户端。怎么存数据、怎么取数据，都与存储引擎有关。
* MySQL客户端-->MySQL服务器-->缓存-->查询检查-->查询优化-->执行查询

# 8.96 查询执行的基础

* 1、客户端发送一条查询给服务器
* 2、服务器先检查查询缓存，如果命中了缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段
* 3、服务器进行SQL解析、预处理，再由优化器生成对应的执行计划
* 4、MySQL根据优化器生成的执行计划，调用存储引擎的API来执行查询
* 5、将结果返回给客户端

## MySQL C/S通信协议

* MySQL客户端和服务器之间的通信协议是半双工的，这意味着，在任何一个时刻，要么是由服务器向客户端发送数据，要么是由客户端向服务器发送数据，这两个动作不能同时发生。所以，我们无法也无需将一个消息切成小块独立来发送。
* 这种协议让MySQL通信简单快速，但是也从很多地方限制了MySQL，一个明显的限制是，这意味着没法进行流量控制。一旦一端开始发送消息，另一端要接收完整个消息才能响应它。
* 客户端用一个单独的数据包将查询传给服务器，这也是为什么当查询的语句很长的时候，参宿max*allowed*packet就特别重要的。一旦客户端发送了请求，它能做的事情就只是等待结果了。
* 相反的，一般服务器响应给用户的数据通常很多，由多个数据包组成。当服务器开始响应客户端请求时，客户端必须完整地接收整个返回结果，而不能简单地只取前面几条结果，然后让服务器停止发送数据。这种情况下，客户端若接受完整的记过，然后取前面几条需要的记过，或者接收完几条结果后就粗暴地断开连接，都不是好主意，这也是在必要的时候一定要在查询中加上LIMIT限制的原因。
* 多数连接MySQL的库函数都可以获得全部结果集并缓存到内存里，还可以逐行获取需要的数据。默认一般是获得全部结果集并缓存到内存中。MySQL通常需要等所有的数据都已经发送给客户端才能释放这条查询所占用的资源，所以接收全部结果并缓存通过可以减少服务器的压力，让查询能够早点结束，早点释放相应的资源。

## 查询缓存

* 在解析一个查询语句之前，如果查询缓存是打开的，那么MySQL会优先检查这个查询是否命中查询缓存中的数据。这个检查是通过一个对大小写敏感的哈希查找实现的。查询和缓存中的查询即使只有一个字节不同，那也不会匹配查询结果，这种情况下查询就会进入下一阶段的处理。
* 如果当前的查询恰好命中了查询缓存，那么在返回查询结果之前MySQL会检查一次用户权限。这仍然是无须解析查询SQL语句的，因为在查询缓存中已经存放了当前查询需要访问的表信息。如果权限没有问题，MySQL会跳过所有其他阶段，直接从缓存中拿到结果并返回给客户端。这种情况下，查询不会被解析，不用生成执行计划，不会被执行。

## 查询优化处理

### 语法解析器和预处理

* 首先，MySQL通过对关键字将SQL语句进行解析，并生成一棵对应的解析树。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树是否合法，下一步预处理器会验证权限。

### 查询优化器

* 现在语法数被认为是合法的了，将由优化器将其转化为执行计划。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。
* MySQL使用基于成本的优化器，它将尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。
* 成本的最小单位是随机读取一个4K数据页的成本。
* 可以通过查询当前会话的Last*query*cost的值来得知MySQL计算的当前查询的成本。
* 成本是根据一系列的统计信息计算得来的：每个表或者索引的页面个数、索引的基数（索引中不同值的数量)、索引和数据行的长度、索引分布情况。优化器在评估成本的时候并不考虑任何层面的缓存，它假设读取任何数据都需要一次磁盘IO。
* 与很多种原因会导致MySQL优化器选择错误的执行计划：
  + 1)统计信息不准确
  + 2)执行计划中的成本估算不等于实际执行的成本
  + 3)MySQL并不按照执行时间最短来选择的，而是基于成本
  + 4)不考虑其他并发执行的查询
  + 5)并不是任何时候都是基于成本的优化，有时也会基于一些固定的规则
  + 6)不会考虑不受其控制的操作的成本（存储过程、用户自定义函数的成本)
* MySQL的查询优化器是使用了很多优化策略来生成一个最优的执行计划。优化策略可以简单地分为两种，一种是静态优化，一种是动态优化。
* 静态优化可以直接对解析树进行分析，并完成优化。静态优化在第一次完成后就一直有效，即使使用不同的参数重复执行查询也不会发送变化，可以认为这是一种编译时优化。
* 相反，动态优化则和查询的上下文有关，需要在每此查询的时候都重新评估，可以认为这是运行时优化。
* MySQL能够处理的优化类型：
  + 1)重新定义关联表的顺序
  + 2)将外连接转为内连接
  + 3)使用等价变换规则
  + 4)优化count、min、max
  + 5)预估并转化为常数表达式
  + 6)覆盖索引扫描
  + 7)子查询优化
  + 8)提前终止查询（比如limit)
  + 9)等值传播
* 10)列表IN()的比较：MySQL会将IN()列表中的数据先进行排序，然后通过二分查找的方式来确定列表中的值是否满足条件，这是一个O(logn)复杂度的操作，等价地转为OR查询的复杂度为O(n)。

### 数据和索引的统计信息

* 在服务器层有查询优化器，却没有保存数据和索引的统计信息。统计信息由存储引擎实现，不同的存储引擎可能会存储不同的统计信息。
* 因为服务器层没有任何统计信息，所以MySQL查询优化器在生成查询的执行计划时，需要向存储引擎获取相应的统计信息。

### MySQL如何执行关联查询

* MySQL认为任何一个查询都是一次关联，并不仅仅是一个查询需要用到两个表匹配才叫关联。
* 当前MySQL关联执行的策略很简单，MySQL对任何关联都执行嵌套循环关联操作。
* 不过不是所有的查询都可以转换成上面的形式，比如全外连接，这大概也是MySQL并不支持全外连接的原因。

### 执行计划

* 和很多其他关系数据库不同，MySQL并不会生成查询字节码来执行查询。MySQL生成查询的一棵指令树，然后通过存储引擎执行完成这颗指令树并返回结果。
* MySQL总是对一张表开始嵌套循环、回溯完成所有表关联。
* 比如四表关联：

### 关联查询优化器

* MySQL优化器最重要的一部分就是关联查询优化，它决定了多个表关联时的顺序。通过多表关联的时候，可以有多种不同的关联顺序来获得相同的执行结果。关联查询优化器通过评估不同顺序时的成本来选择一个代价最小的关联顺序。
* 比如嵌套循环关联时将小表（或者说读取的数据页较小的表)放在最外层。

### 排序优化

* 当不能通过索引生成排序结果的时候，MySQL需要自己进行排序，如果数据量小则在内存中进行，如果数据量大则需要使用磁盘，MySQL将这个过程统一称为filesort文件排序。
* 如果需要排序的数据量小于排序缓冲区，MySQL使用内存进行快速排序操作。如果内存不够排序，那么MySQL会先将数据分块，对每个独立的块使用快速排序，并将各个块的排序结果存放在磁盘上，然后将各个排好序的块进行合并，最后返回排序结果。
* MySQL有两种排序算法：
  + 1)两次传输排序：读取行指针和需要排序的字段，对其进行排序，然后再根据排序结果读取锁需要的数据行。缺点是会产生大量随机IO，数据传输成本高。
  + 2)单次传输排序：先读取查询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果。缺点是会占用大量的空间。
* 当查询需要所有列的总长度不超过max*length*for*sort*data时，MySQL使用单次传输排序。
* 在关联查询时如果需要排序，MySQL会分两种情况来处理这样的文件排序。如果ORDER BY子句中的所有列都来自关联的第一个表，那么MySQL在关联处理第一个表的时候就会进行文件排序。如果是这样，那么EXPLAIN时会显示Extra字段有“Using filesort”。除此之外的其他情况，MySQL都会先将关联的结果存放到一个临时表中，然后在所有的关联都结束后，再进行文件排序。此时EXPLAIN会显示Extra字段有“Using temporary；Using filesort”。如果有LIMIT的话，LIMIT也会在排序之后应用，所以即使需要返回较少的数据，临时表和需要排序的数据量仍然会非常大。

## 查询执行引擎

* MySQL的查询执行引擎会根据执行计划来完成整个查询，执行计划是一个数据结构。
* 相对于查询优化阶段，查询执行阶段不是那么复杂：MySQL只是简单地根据执行计划给出的指令逐步执行。在根据执行计划逐步执行的过程中，有大量的操作需要通过调用存储引擎实现的接口来完成，这些接口就是我们成为“handler API”的皆苦，实际上，MySQL在优化阶段就为每个表都创建了一个handler实例，优化器根据这些实例的接口可以获取表的相关信息。
* 存储引擎接口有着非常丰富的功能，但底层接口却只有几十个，这些接口像搭积木一样能够完成查询的大部分操作。

## 返回结果给客户端

* 查询执行的最后一个阶段是将结果返回给客户端，即使查询不需要返回结果集给客户端，MySQL仍然会返回这个查询的一些信息，如该查询影响到的行数，
* 如果查询可以被缓存，那么MySQL在这个阶段也会将结果存放在查询缓存中。
* MySQL将结果集返回客户端是一个增量、逐步返回的过程。这样处理有两个好处：服务器无需存储太多的结果，也不会因为要返回太多结果而消耗太多内存。另外这样的处理也让MySQL客户端第一时间获得返回的结果。
* 结果集中的每一行都会以一个满足MySQL C/S通信协议的封包发送，再通过TCP协议进行传输，在传输过程中，可能对MySQL的封包进行缓存然后批量传输。

# 8.97 MySQL查询优化器的局限性

## 关联子查询

* MySQL的子查询实现的非常糟糕，最糟糕的一类查询是WHERE中包含IN()的子查询。
* 并不是所有关联子查询性能都很差。建议通过一些测试来判断使用哪种写法速度会更快。

## UNION的限制

* 有时候MySQL无法将限制条件从外层下推到内层，这使得原本能够限制部分返回结果的条件无法应用到内层查询的优化上。

## 索引合并优化

* 当WHERE子句中包含多个复杂条件的时候，MySQL能够访问单个表的多个索引以合并和交叉过滤的方式来定位需要查找的行。

## 并行执行

* MySQL无法利用多核特性来并行执行查询。

## 松散索引扫描

* MySQL不支持松散索引扫描，也就无法不连续的方式扫描一个索引。通常，MySQL的索引扫描需要先定义一个起点和终点，即使需要的数据只是这段索引中很少数的几个，MySQL仍需要扫描这段索引中的每一个条目。

## 最大值和最小值优化

## 在同一个表上查询和更新

# 存储实现

* 每个数据库对应一个子目录，每张表对应子目录下的一个与表同名的.frm文件，它保存了表的定义。
* 如果是MyIASM引擎，那么表数据存放在.myd文件，表索引存放在.myi文件。
* 如果是InnoDB引擎，那么表数据和索引文件都放在.ibd文件。

# InnoDB 简介

* 从MySQL5.5.8开始，InnoDB存储引擎是默认的存储引擎。InnoDB存储引擎将数据放在一个逻辑的表空间中，这个表空间就像黑盒一样由InnoDB存储引擎自身进行管理。
* InnoDB通过MVCC来获得高并发性，并且实现了SQL标准的四种隔离级别，默认为可重复读。同时，使用一种被称为next-key lock的策略来避免幻读。除此之外，InnoDB存储引擎还提供了插入缓冲、两次写、自适应哈希索引、预读等高性能和高可用的概念。
* 对于表中数据的存储，InnoDB存储引擎采用了聚集的方式，因此每张表的存储都是按主键的顺序进行存放。如果没有显式地在表定义时指定主键，InnoDB存储引擎会为每一行生成一个6字节的ROWID，并以此作为主键。

# InnoDB 体系结构

* innoDB存储引擎有多个内存块，可以认为这些内存块组成了一个大的内存池，负责：
  + 1)维护所有线程需要访问的多个内部数据结构
  + 2)缓存磁盘上的数据，方便快速读取，同时在对磁盘文件的数据修改之前在这里缓存
  + 3)redo log缓冲。
* 后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外将已修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常的情况下InnoDB能恢复到正常运行状态。

# 8.98 组件

## 后台线程

* 1、Master Thread：负责将缓冲池只能怪的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲、UNDO页的回收、
* 2、IO Thread
* InnoDB大量使用AIO来处理写IO请求，这样可以极大提高数据库的性能。而IO Thread的工作是负责这些IO 请求的回调处理。
* 3、Purge（清除) Thread
* 事务被提交后，其所使用的undo log可能不再需要，因此需要PurgeThread来回收已经使用并分配的undo页。

## 内存

* 1、缓冲池：InnoDB存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。缓冲池简单来说就是一块内存取与，通过内存的速度来弥补磁盘速度较慢对数据库性能的影响。
* 在数据库中进行读取页的操作，首先将从磁盘读到的页存放在缓存池中，这个过程称为将页“FIX”在缓冲池中，下一次再读相同的页时，首先判断该页是否在缓冲池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。
* 对于磁盘中页的修改操作，则首先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。回写是通过一种称为Checkpoint的机制实现的。
* 在InnoDB存储引擎中，缓冲池中页的大小默认为16KB，使用LRU算法对缓冲池进行管理，并且有一定优化，新读取到的页，不是进入首部，而是放入到LRU列表的midpoint位置。
* 2、redo log缓冲
* InnoDB存储引擎首先将redo log信息先放入到这个缓冲区，然后按一定频率将其刷新到redo log文件。默认为8MB大小。
* 在下列三种情况下会将redo log buffer中的内存刷新到redo log文件中：
  + 1)Master Thread每一秒将刷新一次
  + 2)每个事务提交时刷新
  + 3)redo log buffer剩余空间小于一半时，刷新

## Checkpoint

# 8.99 事务日志 redo log（保证事务持久性 物理日志)

* 事务日志即redo log。
* InnoDB使用日志来减少提交事务时的开销。因为日志中已经记录了事务，就无须在每个事务提交时把缓冲区中的脏块刷新到磁盘中。事务修改的数据和索引通常会映射到表空间的随机位置，所以刷新这些变更到磁盘需要很多随机IO。
* InnoDB用日志把随机IO变成顺序IO，一旦日志安全写到磁盘，事务就持久化了，即使变更还没写到数据文件。如果一些糟糕的事情发生，InnoDB可以重放日志并且恢复已经提交的事务。
* 当然，InnoDB最后还是要把变更写到数据文件，因为日志有固定大小。InnoDB的日志是环形方式写的，当写到此值的尾部，会重新跳转到开头继续写，但不会覆盖还没应用到数据文件的日志记录。
* InnoDB使用一个后台线程智能地刷新这些变更到数据文件。这个线程可以批量组合写入，使得数据写入更顺序，以提高效率。实际上，事务日志把数据文件的随机IO转换成几乎顺序的日志文件和数据文件IO，把刷新操作转移到后台使查询可以更快完成，并且缓和查询高峰时IO系统的压力。
* InnoDB变更任何数据，会写一条变更记录到内存日志缓冲区。在缓冲满的时候、事务提交的时候、或者每一秒钟，InnoDB都会刷新缓冲区的内容到磁盘日志文件——无论上述三个条件哪个先达到。
* 把日志缓冲写到日志文件和把日志刷新到持久化存储是有区别的，在大部分操作系统中，把缓冲写到日志只是简单地把数据从InnoDB的内存缓冲区转移到了操作系统的缓冲区，并没有真正的持久化。
* 当数据库对数据做修改的时候，需要把数据页从磁盘读到buffer pool中，然后在buffer pool中进行修改，那么这个时候buffer pool中的数据页就与磁盘上的数据页内容不一致，称buffer pool的数据页为dirty page 脏数据，如果这个时候发生非正常的DB服务重启，那么这些数据还在内存，并没有同步到磁盘文件中（注意，同步到磁盘文件是个随机IO，较慢)，也就是会发生数据丢失，如果这个时候，能够在有一个文件，当buffer pool 中的data page变更结束后，把相应修改记录记录到这个文件（注意，记录日志是顺序IO)，那么当DB服务发生crash的情况，恢复DB的时候，也可以根据这个文件的记录内容，重新应用到磁盘文件，数据保持一致。
* Innodb将所有对页面的修改操作写入一个专门的文件（顺序IO，很快)。redo log在磁盘上作为一个独立的文件存在，即Innodb的log文件。

# 8.100 逻辑存储结构

* InnoDB把数据保存在表空间内，本质上由一个或多个磁盘文件组成的虚拟文件系统。InnoDB的表空间不只是存储表和索引，它还保存了undo log、插入缓冲、双写缓冲，以及其他内部数据结构。
* 表空间又由段、区、页组成。页有时也称为块。

## 表空间

* 表空间可以看做InnoDB存储引擎逻辑结构的最高层，所有的数据都存放在表空间中。在默认情况下InnoDB存储引擎中有一个共享表空间ibdata1，即所有数据都存放在这个表空间里。如果用户启用了参数innodb*file*per\_table，则每张表内的数据可以单独放在一个表空间中，存放的只是数据、索引和插入缓冲bitmap页，其他数据，如回滚信息、插入缓冲索引页、系统事务信息、二次写缓冲等还是存放在原来的共享表空间中。

### 独立表空间 | 共享表空间

* 独立表空间：每个表都会生成以独立的文件方式来存储，每个表都一个.frm的描述文件，还有一个.ibd文件。其中这个文件包括了单独一个表的数据及索引内容，默认情况下它的存储在mysql指定的目录下。
* 独立表空间优缺点：
* 优点：
* 每个表都有自己独立的表空间；每个表的数据和索引都会存储在各个独立的表空间中；可以实现 单表 在不同的数据进行迁移；表空间可以回收（除了drop table操作，表空不能自己回收)；drop table 操作自动回收表空间，如果对统计分析或是日值表，删除大量数据后可以通过 ：alter table tablename engin=innodb进行回缩不用的空间；对于使用inodb-plugin的innodb使用truncate table会使用空间收缩；对于使用独立表空间，不管怎么删除 ，表空间的碎片都不会太严重。
* 缺点：
* 单表增加过大，如超过100G。对于单表增长过大的问题，如果使用共享表空间可以把文件分开，但有同样有一个问题，如果访问的范围过大同样会访问多个文件，一样会比较慢。对于独立表空间也有一个解决办法是：使用分区表，也可以把那个大的表空间移动到别的空间上然后做一个连接。其实从性能上出发，当一个表超过100个G有可能响应也是较慢了，对于独立表空间还容易发现问题早做处理。
* 共享表空间：某一个数据库所有的表数据、索引保存在一个单独的表空间中，而这个表空间可以由很多个文件组成，一张表可以跨多个文件存在。默认这个共享表空间的文件路径在data目录下，默认的文件名为 bata1,初始化为10M。
* 共享表空间优缺点
* 优点：可以将表空间分成多个文件存放在各个磁盘上（表空间文件大小不受表大小的限制，如一个表可以分布在不同的文件上)，数据和文件放在一起方便管理。
* 缺点：所有的数据和索引存放到一个文件中，将来会是一个很大的文件，虽然可以把一个大文件分成多个小文件，但是多个表及索引在表空间中混合存储，这样对一个表做了大量删除操作后表空间将有大量的空隙，特别是对统计分析、日志系统这类应用最不适合用共享表空间。
* 共享表空间分配后不能回缩：当出现临时建索引或是创建一个临时表的操作表空间扩大后，就是删除相关的表也没办法回缩那部分空间了
* 如何开启独立表空间？
* 查看是否开启独立表空间：
* mysql> show variables like '%per\_table';
* +-----------------------+-------+
* | Variable\_name | Value |
* +-----------------------+-------+
* | innodb*file*per\_table | OFF |
* +-----------------------+-------+
* 设置开启：
* 在my.cnf文件中[mysqld] 节点下添加innodb*file*per\_table=1

## 段

* 表空间是由各个段组成的，比如数据段、索引段、回滚段等。
* 数据段即为B+树的叶子节点，索引段即为B+树的非索引节点。

## 区

* 区是由连续页组成的空间，每个区大小为1MB。默认情况下一页大小为16KB，一个区中有64个连续的页。

## 页

* 在InnoDB存储引擎中，默认每个页的大小为16KB，也可以进行重新设置。
* 在InnoDB存储引擎中，常见的页类型有：
  + 1)数据页
  + 2)undo页
  + 3)系统页
  + 4)事务数据页
  + 5)插入缓冲bitmap页
  + 6)插入缓冲空闲列表页
* 等等

## 行

* 每个页允许存放16KB/2 – 200 行的记录，即7992行记录。

# 8.101 InnoDB 特性

## 两次写 Double Write

* InnoDB使用双写缓冲来避免页没写完整所导致的数据损坏。当一个磁盘写操作不能完整地完成时，不完整的页写入就可能发生。
* 双写缓冲是表空间的一个特殊的保留区域，在一些连续的块中足够保存100个页。本质上是一个最近写回的页面的备份拷贝。当InnoDB从缓冲池刷新页面到磁盘时，首先把他们刷新到双写缓冲中，然后再把它们写到其所属的数据区域中，这样可以保证每个页面的写入都是原子并且持久的。
* 如果有一个不完整的页写到了双写缓冲，原始的页依然会在磁盘上它的真实位置。当InnoDB恢复时，它将用原始页面替换掉双写缓冲中的损坏页面。然而，如果双写缓冲成功写入，但写到页的真实位置失败了，InnoDB在恢复时将使用双写缓冲中的拷贝来替换。InnoDB知道什么时候页面损坏了，因为每个页面在尾部都有校验值。校验值是最后写到页面的冬休，所以如果页面的内容跟校验值不匹配，说明这个页面是损坏的。因此，在恢复的时候，InnoDB只需要读取双写缓冲中每个页面并且验证校验值，如果一个页面的校验值不对，就从它的原始位置读取这个页面。

## 插入缓冲 Insert Buffer

## 自适应哈希索引

## 异步IO

## 刷新邻接页

# InnoDB 数据组织方式与索引分类

* InnoDB存储引擎的数据组织方式，是聚簇索引表：完整的记录，存储在主键索引中，通过主键索引，就可以获取记录所有的列。
* 当在表上定义PRIMARY KEY时，InnoDB将它用作聚簇索引。尽量为每个表定义一个主键。如果没有逻辑的唯一且非空的列或一组列，则添加一个新的AUTO\_INCREMENT列，其值将自动填入。
* 如果没有为表定义一个PRIMARY KEY，那么MySQL将定位第一个UNIQUE索引，其中所有的键列都是非NULL，InnoDB将它用作聚簇索引。
* 如果该表没有PRIMARY KEY或合适的UNIQUE索引，则InnoDB会在包含行ID值的合成列内部生成一个名为GEN*CLUST*INDEX的隐藏聚簇索引。这些行按照InnoDB分配给这个表中的行的ID进行排序。行ID是一个6字节的字段，随着新行的插入而单调递增。因此，由行ID排序的行在物理上处于插入顺序。
* 因此，每张表都会有一个聚簇索引。聚簇索引是一级索引。
* 聚簇索引以外的所有索引都称为二级索引。在InnoDB中，二级索引中的每条记录（叶子)都包含该行的主键列，以及为二级索引指定的列。 InnoDB使用这个主键值来搜索聚簇索引中的行。
* 聚簇索引一般是主键；没有主键，就是第一个唯一键；没有唯一键，就是隐藏ID。

# 锁与事务实现原理

* 概述
* 所有的存储引擎都以自己的方式实现了锁机制，服务器层完全不了解存储引擎中的锁实现。但服务器层也会使用各种有效的表锁来实现不同的目的。
* 对于MySQL而言，事务机制更多是靠底层的存储引擎实现的，在服务器层面只有表锁。支持事务的InnoDB存储引擎实现了行锁、gap锁、next-key锁。
* 分类
* 按操作类型分
* 读锁和写锁
* 按数据操作粒度分
* 表锁和行锁
* MySQL对锁提供了多种选择。每种MySQL存储引擎都可以实现自己的锁策略和锁粒度。在存储引擎的设计中，锁管理是一个非常重要的决定。将锁粒度固定在某个级别，可以为某些特定的应用场景提供更好的性能，但同时也会失去对另外一些应用场景的良好支持。好在MySQL支持多个存储引擎的架构，所以不需要单一的通用解决方案。
* MyISAM表锁

## 特点

* 偏向MyISAM存储引擎，开销小，加锁快，无死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低
* 在特定的场景中，表锁也可能有良好的性能。比如，READ LOCAL表锁支持某些类型的并发写操作；另外，写锁也比读锁有更高的优先级，因此一个写锁请求可以会被插入到读锁队列的前面。
* 尽管存储引擎可以管理自己的锁，服务器还是会使用各种有效的表锁来实现不同的目的。比如服务器在ALTER TABLE时使用表锁，而忽略存储引擎的锁机制。
* MyISAM在读表前自动对表加读锁，在写表前自动对表加写锁。
* 案例
* mylock：
* 手动增加表锁：
* lock table table1 read/write , table2 read/write ,...
* 显示加过锁的表：
* show open tables;
* 释放表锁：
* unlock tables;
* In\_use为1表示已经被加锁。
* 加读锁
* 左边是用户1，先给mylock加了读锁；右边是用户2，尝试给mylock加写锁，无法获得，处于阻塞状态。
* 左边是用户1，给mylock加了读锁，由于读锁是共享锁，所以用户1和用户2都可以查询。

session*1 session*2  
获得表mylock的READ锁定  
 连接终端  
当前session可以查询该表记录

其他session也可以查询该表的记录

当前session不能查询其它没有锁定的表

其他session可以查询或者更新未锁定的表

当前session中插入或者更新读锁锁定的表都会提示错误：  
 其他session插入或者更新锁定表会一直等待获得锁：

释放锁  
 Session2获得锁，插入操作完成：

* 用户A给表A加了读锁之后，只能读表A，不能写表A（报错)，也不能读写其他表（报错)。
* 此时用户B可以读表A，可以读写其他表，但是写表A时会出现阻塞（未报错)，直至用户A释放表A的锁之后才解除阻塞，执行命令。
* 加写锁

session*1 session*2  
获得表mylock的WRITE锁定  
 连接终端  
当前session对锁定表的查询+更新+插入操作都可以执行：

其他session对锁定表的查询被阻塞，需要等待锁被释放：

释放锁  
 Session2获得锁，查询返回：

* 总结
* MyISAM在执行查询语句（SELECT)前，会自动给涉及的所有表加读锁，在执行增删改操作前，会自动给涉及的表加写锁。
* MySQL的表级锁有两种模式：  
  锁类型 可否兼容 读锁 写锁  
  读锁 是 是 否  
  写锁 是 否 否
* 结论：
* 结合上表，所以对MyISAM表进行操作，会有以下情况：
* 1、对MyISAM表的读操作（加读锁)，不会阻塞其他进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其它进程的写操作。
* 2、对MyISAM表的写操作（加写锁)，会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其它进程的读写操作。
* 简而言之，就是读锁会阻塞写，但是不会堵塞读。而写锁则会把读和写都堵塞。
* 分析
* 【看看哪些表被加锁了】
* MySQL>show open tables;
* 【如何分析表锁定】
* 可以通过检查table*locks*waited和table*locks*immediate状态变量来分析系统上的表锁定：
* SQL：show status like 'table%';
* 这里有两个状态变量记录MySQL内部表级锁定的情况，两个变量说明如下：
* Table*locks*immediate：产生表级锁的次数，表示可以立即获取锁的查询次数，每立即获取锁值加1 ；
  + Table*locks*waited：出现表级锁定争用而发生等待的次数(不能立即获取锁的次数，每等待一次锁值加1)，此值高则说明存在着较严重的表级锁争用情况；
* 此外，Myisam的读写锁调度是写优先，这也是myisam不适合做写为主表的引擎。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永远阻塞。
* InnoDB行锁
* 特点  
  - 锁粒度小，并发度高；开销大，加锁慢，会出现死锁  
  - 支持事务
* 分析
* 【如何分析行锁定】
* 通过检查InnoDB*row*lock状态变量来分析系统上的行锁的争夺情况
* MySQL>show status like 'innodb*row*lock%';
* 对各个状态量的说明如下：
* Innodb*row*lock*current*waits：当前正在等待锁定的数量；
* Innodb*row*lock\_time：从系统启动到现在锁定总时间长度；
* Innodb*row*lock*time*avg：每次等待所花平均时间；
* Innodb*row*lock*time*max：从系统启动到现在等待最长的一次所花的时间；
* Innodb*row*lock\_waits：系统启动后到现在总共等待的次数；
* 对于这5个状态变量，比较重要的主要是
* Innodb*row*lock*time*avg（等待平均时长)，
* Innodb*row*lock\_waits（等待总次数)
* Innodb*row*lock\_time（等待总时长)这三项。
* 尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。
* 优化建议

# 8.102 事务隔离级别

* 锁出现的问题：  
  - 脏读（读-DB结果不一致)：在一个事务中，读取其他事务未提交的数据，其他事务回滚后，导致读到的数据与数据库中的数据不一致；   
  - 不可重复读（读-读结果不一致)：一个事务中多次读取相同记录结果不一致（另一事务对该记录进行增改删)；  
  - 幻读（读-写，用写来验证读，结果不一致)：一个事务中读取某个范围内的记录，另一个事务在该范围内插入新的记录，虽然直接查询读取不到，但在插入同PK（同另一个事务插入记录的PK)时会冲突，并且更新范围记录时会同时更新另一个事务新插入的记录。插入同PK和更新范围记录虽然是写，但是在写之前也是要读的，所以也算在读到不同的记录里面了。
* 事务隔离级别：  
  - 读未提交（都不能避免) 事务中的数据即使没有提交，也会对其他事务可见；  
  - 读已提交（可避免脏读，提交读，可以立即读到其他事务提交的数据)：一个事务从开始直接提交之前，所做的任何修改对其他事务都是不可见的；  
  - 可重复读（可避免脏读、不可重复读，快照读，一致性读)：一个事务中多次读取相同的记录，结果是一致的；
* 如果使用select ... for update、lock in share mode才会避免幻读，在第二次读的时候便可读到其他事务更新的数据（相当于破坏了可重复读，但是不会出现幻影)。
* InnoDB使用MVCC来实现可重复读（也可实现读已提交)，但没有解决幻读问题；
* 另外，InnoDB提供了这样的机制：在默认的可重复读的隔离级别里，可以使用加锁读去查询最新的数据。这个加锁读使用到的机制就是next-key locks。  
  - 串行化（都可避免)：

## 幻读示例1——插入同PK

Session*1 Session*2  
开启事务（隔离级别为可重复读) 开启事务（隔离级别为可重复读)  
范围查询表 插入PK为n的记录，并提交  
再次范围查询表，同上次查询结果一致，没有看到PK为n的记录（体现了可重复读)   
插入PK为n的记录，报错：Duplicate key for key ‘PRIMARY’（出现了幻读，因为根据上次查询的结果，本不应该存在PK为n的记录的)   
提交

## 幻读示例2——范围更新

Session*1 Session*2  
开启事务（隔离级别为可重复读) 开启事务（隔离级别为可重复读)  
范围查询表，共n条记录 在Session*1的范围内插入记录，并提交*  
*再次范围查询表，同上次查询结果一致，没有看到Session*2插入的新的记录（体现了可重复读)   
该范围内记录全部更新，最终更新了n+1条记录，包括Session\_2插入的新的记录（出现了幻读，因为根据之前查询的结果只有n条记录的)   
提交

## 解决幻读示例3——排他锁

Session*1 Session*2  
开启事务（隔离级别为可重复读) 开启事务（隔离级别为可重复读)  
范围查询表，共n条记录，并使用select...for update 在Session*1的范围内插入记录，阻塞。*  
*再次范围查询表，同上次查询结果一致，没有看到Session*2插入的新的记录（体现了可重复读)   
整体更新该范围内的记录，最终更新了n条记录（没有幻读)   
提交 解除阻塞

# 8.103 事务隔离级别的实现

## 读未提交

* 无锁

## 读已提交

* MVCC

## 可重复读

* MVCC只工作在REPEATABLE READ和READ COMMITED隔离级别下。
* MVCC最大的作用是: 实现了非阻塞的读操作,写操作也只锁定了必要的行.

## 可序列化

* 读加共享锁，写加排他锁，读写互斥。使用的悲观锁的理论。

# 8.104 MVCC

* MVCC 在MySQL 中的实现依赖的是 undo log 与 read view。

## undo log（保证事务原子性->事务回滚)

* undo log是为回滚而用，具体内容就是copy事务前的数据库内容（行)到undo buffer，在适合的时间把undo buffer中的内容刷新到磁盘。undo buffer与redo buffer一样，也是环形缓冲，但当缓冲满的时候，undo buffer中的内容会也会被刷新到磁盘；
* 与redo log不同的是，磁盘上不存在单独的undo log文件。 Undo记录默认记录在系统表空间（ibdata)中，从MySQL 5.6开始，Undo使用的表空间可以分离为独立的Undo log文件。
* 在Innodb当中，INSERT操作在事务提交前只对当前事务可见，Undo log在事务提交后即会被删除，因为新插入的数据没有历史版本，所以无需维护Undo log。而对于UPDATE、DELETE，则需要维护多版本信息。
* 在InnoDB当中，UPDATE和DELETE操作产生的Undo log都属于同一类型：update\_undo。（update可以视为insert新数据到原位置，delete旧数据，undo log暂时保留旧数据)
* UNDO内部由多个回滚段组成，即 Rollback segment，一共有128个，保存在ibdata系统表空间中，分别从resg slot0 - resg slot127，每一个resg slot，也就是每一个回滚段，内部由1024个undo segment 组成。
* 回滚段（rollback segment)分配如下：
* slot 0 ，预留给系统表空间；
* slot 1- 32，预留给临时表空间，每次数据库重启的时候，都会重建临时表空间；
* slot33-127，如果有独立表空间，则预留给UNDO独立表空间；如果没有，则预留给系统表空间；
* 回滚段中除去32个提供给临时表事务使用，剩下的 128-32=96个回滚段，可执行 96\*1024 个并发事务操作，每个事务占用一个 undo segment slot，注意，如果事务中有临时表事务，还会在临时表空间中的 undo segment slot 再占用一个 undo segment slot，即占用2个undo segment slot。如果错误日志中有：Cannot find a free slot for an undo log。则说明并发的事务太多了，需要考虑下是否要分流业务。
* 回滚段（rollback segment )采用 轮询调度的方式来分配使用，如果设置了独立表空间，那么就不会使用系统表空间回滚段中undo segment，而是使用独立表空间的，同时，如果回顾段正在 Truncate操作，则不分配。

## rollback segment（为了提高并发度)

* 在Innodb中，undo log被划分为多个段，具体某行的undo log就保存在某个段中，称为回滚段。可以认为undo log和回滚段是同一意思。

## row

* 最基本row中包含一些额外的存储信息 DATA*TRX*ID，DATA*ROLL*PTR，DB*ROW*ID，DELETE BIT。  
  - 6字节的DATA*TRX*ID 标记了最新更新这行记录的transaction id，每处理一个事务，其值自动+1  
  - 7字节的DATA*ROLL*PTR 指向当前记录项的rollback segment的undo log记录，找之前版本的数据就是通过这个指针  
  - 6字节的DB*ROW*ID，当由innodb自动产生聚簇索引时，聚簇索引包括这个DB*ROW*ID的值，否则聚簇索引中不包括这个值.，这个用于索引当中  
  - DELETE BIT位用于标识该记录是否被删除，这里的不是真正的删除数据，而是标志出来的删除。真正意义的删除是在commit的时候。
* 更新一行的过程：
* begin->用排他锁锁定该行->记录redo log->记录undo log->修改当前行的值，写事务编号，回滚指针指向undo log中的修改前的行

## read view

* 在innodb中，创建一个新事务的时候，innodb会将当前系统中的活跃事务列表（trx*sys->trx*list)创建一个副本（read view)，副本中保存的是系统当前不应该被本事务看到的其他事务id列表。当用户在这个事务中要读取该行记录的时候，innodb会将该行当前的版本号（trx\_id)与该read view进行比较。
* 简单来说，Read View记录读开始时，所有的活动事务，这些事务所做的修改对于Read View是不可见的。除此之外，所有其他的小于创建Read View的事务号的所有记录均可见
* 新建事务(当前事务)与正在commit 的事务不在活跃事务列表中。
* 函数：read*view*sees*trx*id。
* read\_view中保存了当前全局的事务的范围：
* 【low*limit*id 最迟的事务id， up*limit*id 最早的事务id】
  1. 当行记录的事务ID小于当前系统的最早活动id，就是可见的。
* if (trx*id < view->up*limit\_id) {
* return(TRUE);
* }
  1. 当行记录的事务ID大于等于当前系统的最迟活动id，就是不可见的。
* if (trx*id >= view->low*limit\_id) {
* return(FALSE);
* }
  1. 当行记录的事务ID在活动范围之中时，判断是否在活动列表中，如果在就不可见，如果不在就是可见的。
* for (i = 0; i < n\_ids; i++) {
* trx*id*t view*trx*id
  + = read*view*get*nth*trx*id(view, n*ids - i - 1);
* if (trx*id <= view*trx\_id) {
* return(trx*id != view*trx\_id);
* }
* }

## 不同隔离级别下read\_view的生成规则

### 读已提交

* 在每次语句执行的过程中，都关闭read*view, 重新在row*search*for*MySQL函数中创建当前的一份read\_view。
* 这样就可以根据当前的全局事务链表创建read\_view的事务区间，实现read committed隔离级别。

### 可重复读

* 在repeatable read的隔离级别下，创建事务trx结构的时候，就生成了当前的global read view。
* 使用trx*assign*read\_view函数创建，一直维持到事务结束，这样就实现了repeatable read隔离级别。

## update

- 1)事务1更改该行的各字段的值

* 当事务1更改该行的值时，会进行如下操作：  
  - 用排他锁锁定该行  
  - 记录redo log  
  - 把该行修改前的值Copy到undo log，即上图中下面的行  
  - 修改当前行的值，填写事务编号，使回滚指针指向undo log中的修改前的行
  + 2)事务2修改该行的值
* 与事务1相同，此时undo log，中有有两行记录，并且通过回滚指针连在一起。
* 因此，如果undo log一直不删除，则会通过当前记录的回滚指针回溯到该行创建时的初始内容，所幸的时在Innodb中存在purge线程，它会查询那些比现在最老的活动事务还早的undo log，并删除它们，从而保证undo log文件不至于无限增长。
* 当事务正常提交时InnoDB只需要更改事务状态为COMMIT即可，不需做其他额外的工作，而Rollback则稍微复杂点，需要根据当前回滚指针从undo log中找出事务修改前的版本，并恢复。如果事务影响的行非常多，回滚则可能会变的效率不高，根据经验每事务行数在1000～10000之间，Innodb效率还是非常高的。很显然，Innodb是一个COMMIT效率比Rollback高的存储引擎。

## select

* 查询时会将行的事务id与read\_view中的活动事务列表进行匹配。
* 记录可见，且Deleted bit = 0；当前记录是可见的有效记录。
* 记录可见，且Deleted bit = 1；当前记录是可见的删除记录。此记录在本事务开始之前，已经删除。

## InnoDB MVCC 与 理想 MVCC的区别

* 一般我们认为MVCC有下面几个特点：
* 每行数据都存在一个版本，每次数据更新时都更新该版本
* 修改时Copy出当前版本随意修改，各个事务之间无干扰
* 保存时比较版本号，如果成功（commit)，则覆盖原记录；失败则放弃copy（rollback)
* 就是每行都有版本号，保存时根据版本号决定是否成功，有乐观锁的味道
* 而Innodb的实现方式是：
* 事务以排他锁的形式修改原始数据
* 把修改前的数据存放于undo log，通过回滚指针与主数据关联
* 修改成功（commit)啥都不做，失败则恢复undo log中的数据（rollback)
* 二者最本质的区别是，当修改数据时是否要排他锁定，如果锁定了还算不算是MVCC？
* Innodb的实现真算不上MVCC，因为并没有实现核心的多版本共存，undo log中的内容只是串行化的结果，记录了多个事务的过程，不属于多版本共存。但理想的MVCC是难以实现的，当事务仅修改一行记录使用理想的MVCC模式是没有问题的，可以通过比较版本号进行回滚；但当事务影响到多行数据时，理想的MVCC就无能为力了。
* 比如，如果Transaciton1执行理想的MVCC，修改Row1成功，而修改Row2失败，此时需要回滚Row1，但因为Row1没有被锁定，其数据可能又被Transaction2所修改，如果此时回滚Row1的内容，则会破坏Transaction2的修改结果，导致Transaction2违反ACID。
* 理想MVCC难以实现的根本原因在于企图通过乐观锁代替两阶段提交。修改两行数据，但为了保证其一致性，与修改两个分布式系统中的数据并无区别，而两阶段提交是目前这种场景保证一致性的唯一手段。二段提交的本质是锁定，乐观锁的本质是消除锁定，二者矛盾，故理想的MVCC难以真正在实际中被应用，Innodb只是借了MVCC这个名字，提供了读的非阻塞而已。

# 8.105 InnoDB锁分类

## record lock

- 1)InnoDB里的行锁（record lock)是索引记录的锁  
- 2)record lock锁住的永远是索引，而非记录本身，即使该表上没有任何索引，那么innodb会在后台创建一个隐藏的聚集主键索引，那么锁住的就是这个隐藏的聚集主键索引。所以说当一条sql没有走任何索引时，那么将会在每一条聚簇索引后面加X锁，这个类似于表锁，但原理上和表锁应该是完全不同的。  
- 3)若多个物理记录对应同一个索引，若同时访问，也会出现锁冲突  
- 4)当表有多个索引时，不同事务可以使用不同的索引锁住不同的行。

* 如果走的是聚簇索引，那么会锁住聚簇索引；
* 如果走的是二级索引，那么会同时锁住二级索引和聚簇索引
  + 5)即便在条件中使用了索引字段，但是否使用索引来检索数据是由MySQL通过判断不同执行计划的代价来决定的，如果MySQL认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。

## gap lock

* 锁定一个范围的记录,但不包括记录本身。锁加在未使用的空闲空间上,可能是两个索引记录之间，也可能是第一个索引记录之前或最后一个索引之后的空间.
* 示例：
* create table test(id int,v1 int,v2 int,primary key(id),key idx\_v1(v1))Engine=InnoDB DEFAULT CHARSET=UTF8;
* 该表的记录如下：
* +----+------+------+
* | id | v1 | v2 |
* +----+------+------+
* | 1 | 1 | 0 |
* | 2 | 3 | 1 |
* | 3 | 4 | 2 |
* | 5 | 5 | 3 |
* | 7 | 7 | 4 |
* | 10 | 9 | 5 |
* 间隙锁（Gap Lock)一般是针对非唯一索引而言的，test表中的v1（普通索引，非唯一索引)字段值可以划分的区间为：
  + （-∞,1)
  + （1,3)
  + （3,4)
  + （4,5)
  + （5,7)
  + （7,9)
* （9, +∞)
  + 假如要更新v1=7的数据行，那么此时会在索引idx\_v1对应的值，也就是v1的值上加间隙锁，锁定的区间是（5,7)和（7,9)。同时找到v1=7的数据行的主键索引和非唯一索引，对key加上锁。

## next-key lock

* 行锁与间隙锁组合起来用就叫做Next-Key Lock。锁定一个范围，并且锁定记录本身。对于行的查询，都是采用该方法，主要目的是解决幻读的问题。
* InnoDB工作在可重复读隔离级别下，并且会以Next-Key Lock的方式对数据行进行加锁，这样可以有效防止幻读的发生。Next-Key Lock是行锁和间隙锁的组合，当InnoDB扫描索引记录的时候，会首先对索引记录加上行锁（Record Lock)，再对索引记录两边的间隙加上间隙锁（Gap Lock)。加上间隙锁之后，其他事务就不能在这个间隙修改或者插入记录。
* 例如一个索引有10,11,13,和20这四个值，那么该索引表可能被next-key locking的区间：
* （负无穷，10)
  + 【10,11)
  + 【11,13)
* 【13,20)
* 【20,正无穷)
* 如果包含唯一索引，那么会对其进行优化，降级为Record Lock。
* 但是如果唯一索引是复合索引，而查询仅是最左前缀，则仍会使用next-key lock。  
  示例：

## 意向锁

* innodb的意向锁主要用户多粒度的锁并存的情况。比如事务A要在一个表上加S锁，如果表中的一行已被事务B加了X锁，那么该锁的申请也应被阻塞。如果表中的数据很多，逐行检查锁标志的开销将很大，系统的性能将会受到影响。为了解决这个问题，可以在表级上引入新的锁类型来表示其所属行的加锁情况，这就引出了“意向锁”的概念。举个例子，如果表中记录1亿，事务A把其中有几条记录上了行锁了，这时事务B需要给这个表加表级锁，如果没有意向锁的话，那就要去表中查找这一亿条记录是否上锁了。如果存在意向锁，那么假如事务Ａ在更新一条记录之前，先加意向锁，再加Ｘ锁，事务B先检查该表上是否存在意向锁，存在的意向锁是否与自己准备加的锁冲突，如果有冲突，则等待直到事务Ａ释放，而无须逐条记录去检测。事务Ｂ更新表时，其实无须知道到底哪一行被锁了，它只要知道反正有一行被锁了就行了。
* 说白了意向锁的主要作用是处理行锁和表锁之间的矛盾，能够显示“某个事务正在某一行上持有了锁，或者准备去持有锁”

## gap lock的危害

* 【什么是间隙锁】
* 当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，
* 【危害】
* 因为Query执行过程中通过过范围查找的话，他会锁定整个范围内所有的索引键值，即使这个键值并不存在。
* 间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害  
  Session*1 Session*2  
  阻塞产生，暂时不能插入

commit; 阻塞解除，完成插入

# 8.106 InnoDB加锁分析

## 一致性非锁定读（快照读，无锁，读不会阻塞，也不会阻塞其他事务读写)

* 一致性非锁定读基于MVCC，实现了非阻塞读，读不加锁。
  + 1)在read committed隔离级别下：
* 一致性非锁定读总是读取被锁定行的最新一份快照数据. 产生了不可重复读的问题.
  + 2)在repeatable read 事务隔离级别下：
* 一致性非锁定读总是读取事务开始时的行数据版本. 解决不可重复读的问题

## 一致性锁定读（有锁，读可能阻塞，会阻塞其他事务写)

* 一致性锁定读就是select ... for update 和 select ... in shard mode，读可能会被阻塞，因为是加了锁的。
* SELECT ... LOCK IN SHARE MODE（其他事务可读，可加共享锁，不可加排他锁，不可写)
* 在扫描到的任何索引记录上加共享的next-key lock，还有聚簇索引加排它锁
* 保证读到的是最新的数据（参见幻读)，并且保证其他事务无法修改正在读的数据，事务完毕后解锁。但是自己不一定能够修改数据，因为有可能其他的事务也对这些数据加了共享锁。
* SELECT ... FOR UPDATE（其他事务可读，不可加共享锁&排他锁，不可写)
* 在扫描到的任何索引记录上加排它的next-key lock，还有聚簇索引加排它锁
* 保证读到的是最新的数据（参见幻读)，并且保证同一个事务的读-改-写是一个原子性的操作，事务完毕后解锁。

## 当前读

* select \* from table where ? lock in share mode;
* select \* from table where ? for update;
* insert into table values (…);
* update table set ? where ?;
* delete from table where ?;
* 所有以上的语句，都属于当前读，读取记录的最新版本。并且读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加S锁 (共享锁)外，其他的操作，都加的是X锁 (排它锁)。
* InnoDB 中的加锁，不仅要对where中走的索引加锁，还会对主键聚簇索引加锁。
* 具体加锁情况要根据事务隔离级别和where中走的索引情况具体分析。
* 为什么将 插入/更新/删除 操作，都归为当前读？
* 更新步骤：
* 一个Update操作的具体流程。当Update SQL被发给MySQL后，MySQL Server会根据where条件，读取第一条满足条件的记录，然后InnoDB引擎会将第一条记录返回，并加锁 (current read)。待MySQL Server收到这条加锁的记录之后，会再发起一个Update请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update操作内部，就包含了一个当前读。同理，Delete操作也一样。Insert操作会稍微有些不同，简单来说，就是Insert操作可能会触发Unique Key的冲突检查，也会进行一个当前读。
* 注：根据上图的交互，针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后在读取下一条加锁，直至读取完毕。

## 两段锁协议

* 在事务执行过程中，随时都可以锁定，锁只有在执行commit或rollback时才会释放，并且所有的锁都是在同一时刻被释放。
* 除了DML时隐式加排他锁外，读的时候也可以显式加锁，比如select ... in shard mode 和 select ... for update。
* MySQL也支持lock tables和unlock tables语句，这是在服务器层实现的，和存储引擎无关。它们有自己的用途，但并不能代替事务处理，如果应用需要用到事务，还是应该选择事务型存储引擎。
* 并且显式使用lock tables会与事务中使用的锁产生冲突，因此建议不要显式地使用lock tables。

## 锁与事务隔离级别中的当前读

* 前面讲事务隔离级别的实现注重于快照读，这里主要讲的是当前读，也就是一致性非锁定读+DML的锁实现。
* 读已提交：当前读时，对读到的记录加行锁
* 可重复读：当前读时，对读到的记录加行锁，同时对读取的范围加间隙锁。
* 可序列化：不区分当前读与快照读，所有的读都是当前读，读加读锁，写加写锁。

## 案例 DML+select...for update

### RC+where走聚簇索引

* 聚簇索引加行级排他锁

### RC+where走二级索引（包括唯一索引和非唯一索引)

* 满足条件的记录的二级索引加排他锁，聚簇索引加排他锁

### RC+where无索引

* 在聚簇索引上全表加排他锁。为了效率考量，MySQL做了优化，对于不满足条件的记录，会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了2PL的约束。

### RR+where走聚簇索引

* 聚簇索引加排他锁

### RR+where走唯一索引

* 满足条件的记录的唯一索引加排他锁，聚簇索引加排他锁

### RR+where走非唯一索引

* GAP锁锁住的位置，也不是记录本身，而是两条记录之间的GAP。
* 三个GAP锁：[6,c]与[10,b]间，[10,b]与[10,d]间，[10,d]与[11,f]
* Insert操作，如insert [10,aa]，首先会定位到[6,c]与[10,b]间，然后在插入前，会检查这个GAP是否已经被锁上，如果被锁上，则Insert不能插入记录。因此，通过第一遍的当前读，不仅将满足条件的记录锁上 (X锁)，同时还是增加3把GAP锁，将可能插入满足条件记录的3个GAP给锁上，保证后续的Insert不能插入新的id=10的记录，也就杜绝了同一事务的第二次当前读，出现幻读的情况。
* 对于每条满足条件的记录，会先加非唯一索引上的排他锁，加GAP上的GAP锁，然后加聚簇索引上的排他锁。

### RR+where无索引

* 在聚簇索引上全表加排他锁，同时会锁上聚簇索引上的所有GAP。
* MySQL也做了一些优化，就是所谓的semi-consistent read。semi-consistent read开启的情况下，对于不满足查询条件的记录，MySQL会提前放锁。针对上面的这个用例，就是除了记录[d,10]，[g,10]之外，所有的记录锁都会被释放，同时不加GAP锁。semi-consistent read如何触发：要么是read committed隔离级别；要么是Repeatable Read隔离级别，同时设置了 innodb*locks*unsafe*for*binlog 参数。

### Serializable

* MVCC并发控制降级为Lock-Based CC，读也加读锁。

# 8.107 死锁

* 死锁的发生与否，并不在于事务中有多少条SQL语句，死锁的关键在于：两个(或以上)的Session加锁的顺序不一致。而使用本文上面提到的，分析MySQL每条SQL语句的加锁规则，分析出每条语句的加锁顺序，然后检查多个并发SQL间是否存在以相反的顺序加锁的情况，就可以分析出各种潜在的死锁情况，也可以分析出线上死锁发生的原因。

# 8.108 只读事务

* Innodb将所有的事务对象维护在链表上，通过trx\_sys来管理，在5.6中，最明显的变化就是事务链表被拆分成了两个链表：
* 一个是只读事务链表：ro*trx*list，其他非标记为只读的事务对象放在链表rw*trx*list上；
* 这种分离，使得读写事务链表足够小，创建readview 的MVCC快照的速度更快；

# binlog

* binlog是MySQL Server层记录的日志， redo log是InnoDB存储引擎层的日志。 两者都是记录了某些操作的日志(不是所有)自然有些重复（但两者记录的格式不同)。
* 选择binlog日志作为replication主要原因是MySQL的特点就是支持多存储引擎，为了兼容绝大部分引擎来支持复制这个特性。

# 8.109 格式

* binlog有三种格式：Statement、Row以及Mixed。从安全性来看，ROW（最安全)、MIXED（不推荐)、STATEMENT（不推荐)。
* –基于SQL语句的复制(statement-based replication,SBR)，
* –基于行的复制(row-based replication,RBR)，
* –混合模式复制(mixed-based replication,MBR)。

## Statement

* 每一条会修改数据的sql都会记录在binlog中。在5.6.24中默认格式。
* 优点：不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。
* 缺点：由于记录的只是执行语句，为了这些语句能在slave上正确运行，因此还必须记录每条语句在执行的时候的一些相关信息，以保证所有语句能在slave得到和在master端执行时候相同 的结果。另外MySQL 的复制,像一些特定函数功能，slave可与master上要保持一致会有很多相关问题。
* ps：相比row能节约多少性能与日志量，这个取决于应用的SQL情况，正常同一条记录修改或者插入row格式所产生的日志量还小于Statement产生的日志量，但是考虑到如果带条件的update操作，以及整表删除，alter表等操作，ROW格式会产生大量日志，因此在考虑是否使用ROW格式日志时应该跟据应用的实际情况，其所产生的日志量会增加多少，以及带来的IO性能问题。

## Row

* 5.1.5版本的MySQL才开始支持row level的复制,它不记录sql语句上下文相关信息，仅保存哪条记录被修改。
* 优点： binlog中可以不记录执行的sql语句的上下文相关的信息，仅需要记录那一条记录被修改成什么了。所以rowlevel的日志内容会非常清楚的记录下每一行数据修改的细节。而且不会出现某些特定情况下的存储过程，或function，以及trigger的调用和触发无法被正确复制的问题。
* 缺点:所有的执行的语句当记录到日志中的时候，都将以每行记录的修改来记录，这样可能会产生大量的日志内容。
* ps:新版本的MySQL中对row level模式也被做了优化，并不是所有的修改都会以row level来记录，像遇到表结构变更的时候就会以statement模式来记录，如果sql语句确实就是update或者delete等修改数据的语句，那么还是会记录所有行的变更。

## Mixed

* 从5.1.8版本开始，MySQL提供了Mixed格式，实际上就是Statement与Row的结合。
* 在Mixed模式下，一般的语句修改使用statment格式保存binlog，如一些函数，statement无法完成主从复制的操作，则采用row格式保存binlog，MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种。

# 8.110 binlog与redo log的区别

- 1)首先，binlog会记录所有与MySQL数据库有关的日志记录，包括InnoDB、MyISAM、Heap等其他存储引擎的日志。而InnoDB存储引擎的redo log日志只记录有关该引擎本身的事务日志。  
  
- 2)其次，记录的内容不同。无论用户将二进制日志文件记录的格式设为STATEMENT还是ROW，又或是MIXED，其记录的都是关于一个事务的具体操作内容，即该日志是逻辑日志。而InnoDB存储引擎的重做日志是关于每个页（Page)的更改的物理情况。  
  
- 3)此外，写入的时间也不同。二进制日志文件仅在事务提交后进行写入，即只写磁盘一次，不论这时该事务多大。而在事务进行的过程中，却不断有redo 条目（redo entry)被写入到重做日志文件中。