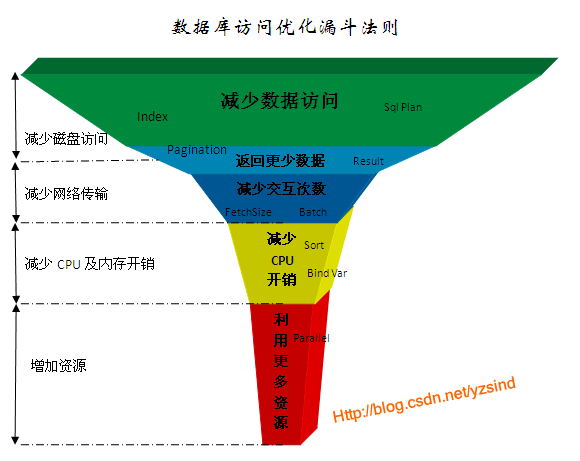
1. **提高数据库性能的原则？（数据库连接，集中，分散等方面）**

参考<http://blog.csdn.net/yzsind/archive/2010/12/06/6059209.aspx>



1. **减少数据访问（减少磁盘访问）**
2. 创建并使用正确的索引

建立索引通常应该考虑：

索引对于查询的效率提高多少

索引对DML(INSERT,UPDATE,DELETE)附加的开销有多少

索引应该建在那些列上

哪些情况能用到索引

1. 只通过索引访问数据

需要检索的数据全都在索引中

1. 优化SQL执行计划

设计适合业务逻辑的良好的SQL（表数量，join数量）

1. 代码中使用函数始终应该首选DBMS自带的函数，代码的执行越接近DBMS核心，则代码执行速度越快
2. 慎用自定义函数。（自定义函数中可能隐藏有查询，阻碍了居于开销的优化器（CBO）对整个查询的优化效果，子查询隐藏在函数中，优化器“鞭长莫及”）
3. **返回更少数据（减少网络传输或磁盘访问）（不需要的数据不要取出来）**
4. 数据分页处理（限制每次取出数据的数量）

客户端(应用程序或浏览器)分页

应用服务器分页

数据库SQL分页

1. 只返回需要的字段

通过去除不必要的返回字段可以提高性能

1、减少数据在网络上传输开销

2、减少服务器数据处理开销

3、减少客户端内存占用

4、字段变更时提前发现问题，减少程序BUG

5、如果访问的所有字段刚好在一个索引里面，则可以使用纯索引访问提高性能。

1. **减少交互次数（减少网络传输）（需要的数据以尽量少的交互次数全部取出）**
2. 批量处理数据，每次数据库连接应该完成尽可能多的工作，充分利用每次数据库访问
3. 使用IN List代替多个where，减少SQL查询的次数
4. 设置每次返回结果的Fetch Size

可以加大fetch\_size，这样可以减少结果数据传输的交互次数及服务器数据准备时间，提高性能

1. 使用存储过程

大型数据库一般都支持存储过程，合理的利用存储过程也可以提高系统性能。（各个DBMS内部编程方式有较大差异，存储过程移植性不好，存储过程中会包含一定的业务逻辑，使得系统维护和管理更加困难，所以普通业务逻辑尽量不要使用存储过程.

1. **减少服务器CPU开销（减少CPU及内存开销）**
2. 使用绑定变量

绑定变量是指SQL中对变化的值采用变量参数的形式提交，而不是在SQL中直接拼写对应的值(java里面的preparedStatement)

1. 合理使用排序

你需要注意是否一定要这么做了，大记录集排序不仅增加了CPU开销，而且可能会由于内存不足发生硬盘排序的现象，当发生硬盘排序时性能会急剧下降

1. 减少模糊查找，例如LIKE
2. 不要把太多的复杂计算交给数据库

大量复杂运算在数据库里不利于数据库高并发处理

1. **优化数据库物理存储及其环境**
2. 选择性能优良的数据库服务器
3. 数据库物理存储的考虑

客户里数据越近，访问速度越快

数据分散磁盘存储，减小单个磁盘的访问压力，增加数据库对于并发的处理能力

1. 数据表的分区
2. **SQL 优化**

IN：先进行内层子查询，再进行外层查询（有一个中间结果集，外层查询的时候使用中间结果集进行查询）。内外层查询间无关联，内层查询过程不使用外层的变量。

EXIST：先通过外层查询条件进行过滤，再进行内层子查询，内外层查询之间有关联。一般内层中会使用到外层的变量。

**选择两者的原则**：尽量在查询初期过滤掉更多的数据（IN首先通过子查询条件过滤，EXIST首先通过外层查询条件过滤）。

1. **索引**

**目的**：提高查询效率

**结构**：常见的索引有B-TREE索引、位图索引、全文索引，这里重点是B-TREE

**索引的几个关键点**：

1. 使用索引可能可以提高查询效率，但是只是可能，没有适当设计的索引可能会降低查询速度，有时候使用所以带来的维护代价可能高于使用它因为加快查询获得的好处
2. 索引是一种以原子粒度访问数据的手段，而不是为了检索大量数据的
3. 在某些列上使用索引可能会导致其他列的查询变慢
4. 系统地对表的外键加上索引的做法非常普遍，提高并发
5. 为每个外键建立索引，可能会导致多余索引

反向键索引或叫逆向索引（reverse index）、哈希索引（hash indexing），他们的目的都是数据尽可能分散

**为什么不使用索引？六点（不要标题党）**

|  |
| --- |
| **PPT上老师的例子**：   1. 情况1：我们在使用B+树索引，而且谓词中没有使用索引的最前列 跳跃式索引（仅CBO）（最左前缀） 2. 情况2：使用SELECT COUNT(\*) FROM T，而且T上有索引，但是优化器仍然全表扫描 3. 情况3：对于一个有索引的列作出函数查询   Select \* from t where f(indexed\_col) = value   1. 情况4：隐形函数查询 2. 情况5：此时如果用了索引，实际反而会更慢 3. 情况6：没有正确的统计信息，造成CBO无法做出正确的选择 |
| 总结：归根到底，不使用索引的通常愿意就是“不能使用索引，使用索引会返回不正确的结果”，或者“不该使用索引，如果使用了索引就会变得更慢” |

**网上的资料**：

1. 不等于操作不能使用索引
2. 经过普通运算或函数运算后的索引字段不能使用索引
3. 含前导模糊查询的Like语法不能使用索引（”%XXX”）
4. B-TREE索引里不保存字段为NULL值记录，因此IS NULL不能使用索引
5. Oracle在做数值比较时需要将两边的数据转换成同一种数据类型，如果两边数据类型不同时会对字段值隐式转换，相当于加了一层函数处理，所以不能使用索引。
6. 给索引查询的值应是已知数据，不能是未知字段值。
7. **数据库的物理组织**

IOT 索引组织表

把索引当成数据仓库，所有数据都存放在索引中。

对IOT表插入的效率也许低于堆文件（显然低啊）

I**OT的用途**

全索引表

代码查找表

高频度的一组关联数据查询

**IOT最大的优点**：记录是排序的…（效率惊人）

任何有序数据便于某些处理的同时，必将对其他处理不利

可能会造成有些列的查询效率降低。例如table(a,b,c)是ITO，且b跟a的相关度不高，则可能出现对b的查询效率变低的情况

分区是一种数据分组的方式

提高并发性（concurrency）和并行性（parallelism）

从而增强系统架构的可伸缩性（scalable）

循环分区： 循环分区完全是一种不受数据影响的内部机制，可以把分区定义为各个磁盘的存储区域，通常会和数据存储设备的数量一样多，一个表可能会存储在一个或者多个分区中，插入数据时，数据会按照某方法循环的加载到各个分区，以保持数据访问引起的磁盘I/O平衡。

对分区表进行查询，当数据按分区键均匀分布时，收益最大

**问题：它们分别是如何提高查询效率的（考试可能考其中的一个）**

分区技术用的好的话可以提高性能，是因为一方面分区把一大块数据分成了n小块，这样查询的时候很快定位到某一小块上，在小块中寻址要快很多；另一方面CPU比磁盘IO快很多倍，而硬件上又有多个磁盘，或者是RAID（廉价磁盘冗余阵列），可以让数据库驱动CPU同时去读写不同的磁盘，这样才有可能可以提高效率。

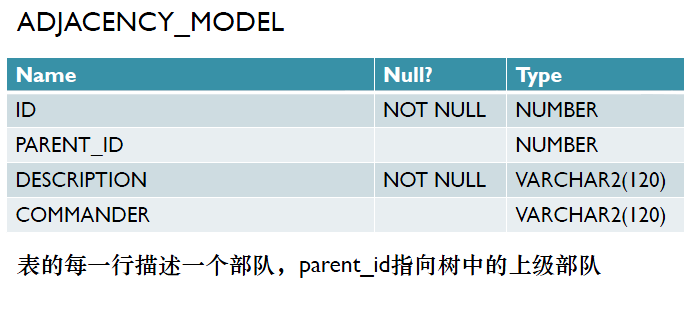
分区在有些时候并不能提高读写效率，比如说我们经常看到的按照日期字段去分区MSDN例子，这个实例中是按照记录的生成时间来分区的，把一年的数据分割成12个分区，每月一个。这样的分区导致分区并不能实现CPU同步写并提高写入性能，因为在同一个时段CPU总是要写入到最新的那一个分区对应的磁盘中。另一个问题是：这样分区是否可以提高读取性能呢？答案是不一定，要看根据什么字段来查询，如果是根据时间来查询，根据时间生成报表那么这种分区肯定会提高查询的效率，但是如果是按照某个客户查询客户最近1年内的账单数据，这样数据分布到不同的分区上，这样的话效率就不一定能提高了，这要看数据在同一个分区上连续分布的读性能高，还是CPU从几个磁盘上同步读取，然后在合并数据的性能更高一些，这和读取数据的记录数也有关系。

1. **表结构设计**

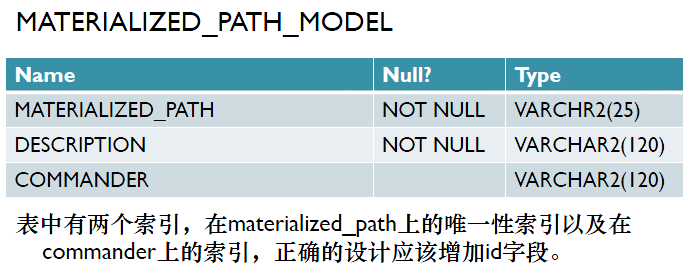
**例子：如何把一棵树存到二维表中？**

1. **邻接模型**

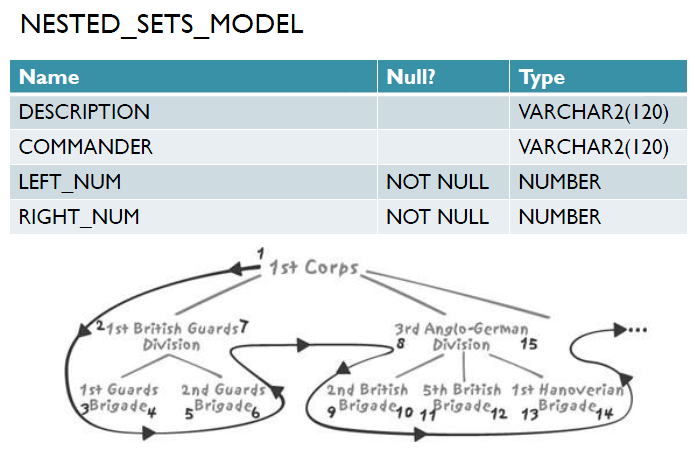
表的每一行描述一个部队，parent\_id指向树中的上级部队



1. **物化路径模型**



1. **嵌套集合模型**



**一，<id, pid>方式；二，物化路径；三，嵌套集合  
问题：这三种方式的性能区别**

1. **自顶向下（Vandamme查询）**

邻接模式:递归实现，效率较低，使用CONNECT BY实现

|  |
| --- |
|  |

使用递归实现

|  |
| --- |
|  |

物化路径模型：path中前半段相同的即为结果，假设mp\_depth()函数返回当前节点深度

|  |
| --- |
|  |

嵌套集合模型：直接找到在left\_num和right\_num中间的所有节点，缩进比较难处理。

|  |
| --- |
|  |

**自顶向下效率：嵌套集合<物化路径<邻间模型**

**二、 自底向上（Highland查询）**

1. **邻间模型:**

|  |
| --- |
|  |

物化路径：

|  |
| --- |
|  |

嵌套集合模型：

|  |
| --- |
|  |

**自底向上：邻间>物化路径>嵌套集合**