chapter1 为性能而设计

1. 应用系统的首要目标：满足业务需求（business requirement）

2. 调优：

* 处理流程
  + 操作模式
    - 异步模式处理（批处理）
    - 同步模式处理（实时交易）
* 数据集中化
  + 分布式数据系统增加复杂性
    - 远程数据的透明引用访问代价很高
    - 不同数据源数据结合极为困难
    - 数据库部署：离数据越近，访问速度越快
* 保持数据库连接稳定，减少交互
* 战略优先战术
* 简单
* 充分利用每次数据库访问，完成尽量多的工作
* 接近DBMS核心
  + 代码离SQL内核越近，运行的越快
* 把逻辑放入查询中
  + 尽量把条件逻辑放到SQL语句中，而不是SQL宿主语言中
* 慎用自定义函数
  + 如果自定义函数中包含select，则不能为CBO Optimizer优化

上届整理：

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

3. 防御式编程（code defensively），在开始处理之前先检查所有参数的合法性。

SQL的进攻式编程（Offensive Coding with SQL），本质特征是：以合理的可能性为基础。对数据库编程而言，尽量同时做几件事情的进攻式编程有切实的优势。

执行效率来说，进攻式编程要简洁、更节约时间，吞吐量更高。

chapter2 SQL的本质

1. 优化器借助关系理论提供的语义无误的原始查询进行有效的等价变换。优化在数据处理真正被执行时发生。

* 优化器会检查索引、数据的物理布局、可用内存大小、可用处理器个数、直接或间接涉及的表和索引的数据量
* 优化器根据数据库的实际实现情况对理论上等价的不同优化方案做出权衡，产生可能的最优查询执行方案

2. 关系操作可以优化，所以最稳妥的是：在关系操作层完成尽量多的工作，对于不完全的关系操作，加倍留意查询的编写。

3. 使用SQL需要考虑：

* 获得结果集所需访问的数据量
* 定义结果集所需的查询条件
* 结果集的大小
* 获得结果集所涉及的表的数量
* 同时修改这些数据用户的多少
  + 数据块访问争用（block-access contention）
  + 阻塞(locking)
  + 闩定(latching)
  + 保证读取一致性(read consistency)

4. 过滤，PPT买BMW的例子

* 避免最高层distinct
  + 关联子查询exists：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，执行多次。
  + 非关联子查询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）。内层查询只需执行一次。
  + 当内层子查询结果集大时，in比exists好。

5. 大量数据查询

* Group by & having子句
  + 所有影响聚合函数结果的条件都应在Having子句
  + 任何无关聚合条件都应该放在where子句
  + 减少group by必须执行排序操作所处理的数据量

chapter3 索引及数据表的物理实现

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

**分类**：

* 聚集索引：表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。（物理地址唯一）
* 非聚集索引：表中行的物理顺序与键值的逻辑顺序不匹配。
* 聚集索引能提高多行检索的速度，非聚集索引对于单行的检索很快。

**关键点**：

* + 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
  + 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
  + 在某些列上使用索引可能会导致其他列的查询变慢
  + 系统地对表的外键加上索引的做法非常普遍，提高并发
  + 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（解决插入并发性过高的资源竞争）

**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）   如T，T(X,Y)上有索引，做SELECT \* FROM T WHERE Y=5   1. 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 2. 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   比如查询SQL中存在隐式类型转换比如no为VARCHAR类型，但是查询时把该字段当做INT类型查询：SELECT \* FROM T WHERE no = 2124   1. 情况5：此时如果用了索引，实际反而会更慢   如经常更新数据，碎片太多，索引占用空间太大   1. 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**：

* 不等于操作不能使用索引
* 经过普通运算或函数运算后的索引字段不能使用索引
* 含前导模糊查询的Like语法不能使用索引（”%XXX”）
* 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
* 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。
* 给索引查询的值应是已知数据，不能是未知字段值。

2. 数据库的物理组织

* 堆文件：

最简单最基本的数据库数据组织方式，新插入的记录直接被插到堆文件末尾（插入效率较高），因此堆文件是无序的。

查询时线性搜索整个文件，查询效率较差。但对于查询所有记录且访问顺序不重要情况下的查询效率比较高。

* IOT：（索引组织表）

插入效率可能低于堆文件，最大优点是记录是排序的。在主键索引中存储表中所有数据。表即索引，在查询一组高频度关联数据以及应用于全索引表、代码查找表时比较高效。

用途：全索引表、代码查找表、高频度的一组关联数据查询

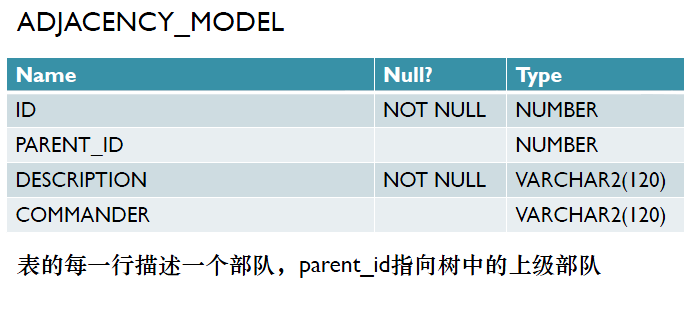
* 分区：

分区是将数据分组，提高了并发性和并行性，从而增强系统架构的可伸缩性。对于多用户并发访问情况，查询效率可能较高。

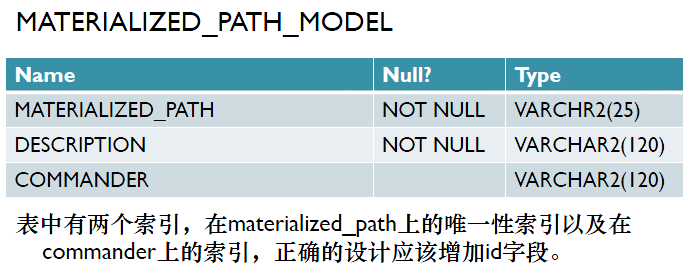
* + 实现方式：
    - 哈希分区：按散列决定记录分在哪个分区
    - 范围分区：范围分区将数据基于范围映射到每一个分区，这个范围是你在创建分区时指定的分区键决定的，最为常用的
    - 列表分区：允许用户将不相关的数据组织在一起，列中的值是有限的若干个，基于这样的特点我们可以采用列表分区。每个分区的该列都是某一个值
  + 更新分区键会引起移动数据，似乎应该避免这么做
    - 例如实现服务队列，类型（T1…Tn）状态（{W|P|D}）
    - 按请求类型分区：进程的等待降低
    - 按状态分区：轮询的开销降低
    - 取决于：服务器进程的数量、轮询频率、数据的相对流量、各类型请求的处理时间、已完成请求的移除频率
    - 要整体考虑

3. 处理层次结构

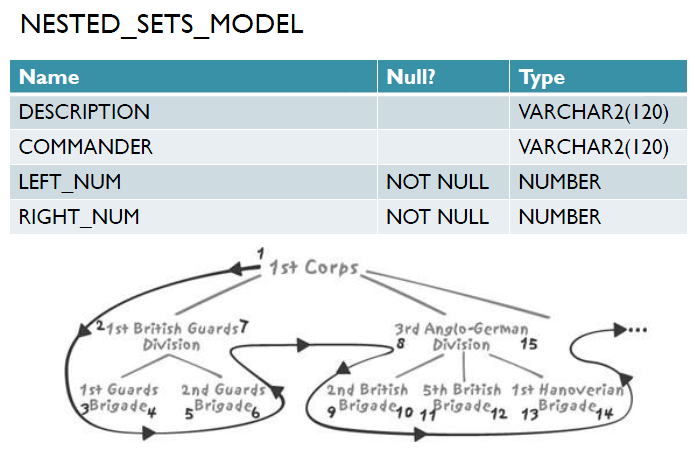
* 树状结构、父子结构、主从结构
* 树状结构保存只需要一张表
* 树的三种模型
  + 邻接模型，表的每一行描述一个部队，parent\_id指向树中的上级部队



* 物化路径模型



* 嵌套集合模型



4. PPT上的问题

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）**

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

chapter4 基于数据的数据库方法学

1. 1NF：同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

2NF：满足第一范式且要求数据库表中的每个实例或行必须可以被组成主键的所有列唯一地区分。

3NF：满足第二范式且要求一个数据库表的非主关键字信息只能被主关键字区分。

2. 建立ER映射

Step 1 Tasks

* Step 1.1 Identify entities
* Step 1.2 Identify relationships
  + 定义关系约束
* Step 1.3 Identify and associate attributes with entities or relationships
* Step 1.4 Determine attribute domains
* Step 1.5 Determine candidate, primary, and alternate key attributes（候选键、主键、替换键）
* Step 1.6 Specialize/Generalize entities (optional step)
* Step 1.7 Check model for redundancy（冗余）
* Step 1.8 Check model supports user transactions（事务）

Step 2 Map ER model to tables

* Step 2.1 Create tables
* Step 2.2 Check table structures using normalization（规范化）
* Step 2.3 Check tables support user transactions
* Step 2.4 Check business rules
* Step 2.5 Review logical database design with users