第一讲

层次型数据库的最大问题：没有科学的抽象

大数据量、高并发情况下提高数据库性能

NoSQL：放宽对一致性的要求提高吞吐量

关系代数对 SQL 有何帮助？通过代数变换进行查询优化！

视图的两种用途：1. 权限（不暴露基本表，用途同接口类似）；2. 重构（使用视图提供给现有应用，保证和重构之前一致）

视图的问题：掩盖了复杂性，潜在的性能问题

范式：尽量降低冗余。冗余：维护数据的一致性需要进行更多操作

反范式/逆范式：一定程度上，打破范式，提高性能——有现有的最佳实践，不要打着提高性能的幌子随意打破范式

1NF: 字段只被当成**一个值**来使用，e.g. 身份证（可拆分，但不拆分进行验证）

第二讲

常用 DBMS 单 CPU 最大并发用户大约是 3000

但，仅限于用户不访问相同数据的情况

——对于同一数据，访问仍是串行的

数据库并发的最小单元——块（物理视图）

质量和设计相关，和测试无关

性能很重要！——必须在设计完成之前进行测试

SQL 的写法会直接影响数据库的性能——直接或间接影响查询优化器的决定

DBA 的职责：备份＋恢复；开发人员的职责：剩下的一切（设计表结构、索引、设置缓存……）

“创造”永远追不上开发的步伐——使用经验而非创造

数据语义属于 DBMS，别放在应用程序中——约束明确说明

灵活的数据库设计往往导致更低的性能

同步/异步、集中/分布——影响性能

第三讲

集中式存储的最大问题：单点故障（复杂的保障手段）

分布式不是“切割”，而是“复制”

- 访问性上升
- 维护一致性难度增大
- 难以进行资源的整体考虑
- 一般用 NoSQL

道理是很难解释的；积累经验自然有更好的方案。

一知半解是非常危险的。（Abstraction != Blockbox）

不同方式访问数据库的速度：核心 SQL>扩展 SQL>C>ODBC

尽可能将逻辑放入查询中，而不是宿主语言中。

进攻式 SQL：

- 假定：正常的操作是大多数（经验阀值 80%）
- 提高吞吐量的有效手段
- 让数据库检查错误（更新行数）
- 多数银行应用不作需要进行交互的有效性检查
- 勇敢和鲁莽的限界很模糊

原则都是有例外的，否则就是公理了。
查询优化器在 3 表以下查询时基本可以遍历所有路径，在 3 表以上时需要对其进行暗示
SQL 语句部分的执行顺序：from/where/select/order by

- 在单一语句中，order by 总是最后做的，因为它不是关系操作

查询优化器：

- 基于规则（对优化水平进行评分）
- 基于成本（规则+系统状况）
- 针对不同大小的表的优化结果可能会完全不同

整体优化好于分步优化——使用大的查询而非多个小查询
优化时的考虑因素：

- 表的大小
- 过滤条件的好坏（强的条件导致更小的中间表，减少和系统的交互）
  - 分类：select、join
  - 通过 join 约束表的连接顺序
- 结果集的大小
- 表的数量
- 并发用户数

数据库并发的最小单位是数据块：

- 不存在行锁
- 块越大，整体的并发性能越差
- 行迁移
  - 记录跨 2 个数据块
  - 一个记录只出现一次，需要第二次迁移时，会将整个记录写入新的数据块中

第四讲
慎用 DISTINCT：去重是容易的，发现错误是很难的
使用*的两个合理场景：COUNT(*)或嵌套查询中 EXISTS (SELECT * ...)
EXISTS 和 IN 就是为了解决 DISTINCT 的问题
EXISTS 嵌套：

- 不需要计算出结果集，也未必会执行完
- 关联嵌套，子查询依赖外部查询、无法单独运行
- 在运行时将外部关联数据逐一带入后运行
- 同外层查询中字段相关的字段一定要有索引，以便提高效率
- 内部单独优化，有自己的 plan，然后将外部的数据带入
- 优点：只进行一次优化；减少了一次表连接（叉乘）
- 缺点：需要代入多次
- **唯一适用情况**：位于和 EXISTS 同一 WHERE 查询中的过滤条件非常强时==产生的中间结果集很小==带入次数少==带入的成本小于表连接的成本

IN 嵌套：

- 不依赖外层查询，只需要执行**一次**
- 非关联嵌套，不需要索引
- 只优化一次，同时也只执行一次，但**一定会执行完**
- 绝大部分情况下，IN 的优化效果好于 EXISTS 和指定顺序的 JOIN

查询优化器只知道优化的好坏，不知道条件的强弱。
越快地剔除不需要的数据，后续的查询效率越高。——尽量减少中间结果集的大小。
查询优化器不可能减少表连接操作，表连接的减少必须由人完成。
降低表的使用次数：在 FROM 中使用嵌套表。
精确的经验随条件而转移，不要去轻信。

第五讲
默认存储方式：堆文件，随机存储（提高并发性，多进程并发时随机挑选空闲块写入）
当一个数据块约有 30%空闲时，则认为该块是可以插入数据的（“空闲”）——不“空闲”时，实际空闲的空间用于防止潜在的“行迁移”
找不到？新增一个块，提高“高水平位”。
因为插入是个随机的过程，所以实际读取的顺序不一定和逻辑顺序相等。
RowID——直接读取数据块的方式，快；物理实现，无法在 SQL 中使用
“胖表”——单一记录很大（单一字段很大或字段很多），一个块中能容纳的记录多；相对应的，“瘦表”

将直接从硬盘中读取一个块定义为一次 I/O 操作，I/O 操作次数影响性能。
B 树的特殊之处：叶节点之间是“连续”的——块之间有指针连接——水平遍历！
使用索引可能降低查询性能
查询优化器何时使用索引？1. 检索比例（10%）；2. 表的胖瘦；3. 是否需要访问基本表
索引本身有开销：1. 空间——可能大于基本数据大小；2. 处理开销——索引是顺序结构（使用索引可能需要多花 150%-200%的时间）
索引更适合读密集型事务，而不适合写密集型事务。

复合键索引的操作方式：index(x,y)实际仍以 x 为索引，y 的顺序是散的
索引只是提供了另一种访问数据的方式，并不保证提高查询效率
任何 DBMS 中，主键索引是自动加的
目录是区域访问方法，索引是点状访问——索引不适合大段访问
数据库中区域访问的方式：分区（partition）

既然只查询索引字段只需要使用索引，为何不在整个表上建索引？
1. 不利于并发，基于堆的基本表利用随机插入有利于并发
2. 索引是层次式的，只有第一个字段是有序的，其他字段是无序的

数据库冲突：数据密集存储和数据分散存储之间的矛盾
- 想读得快，希望数据集中，减少读取次数
- 想写得快，希望数据分散，方便插入

死锁的本质：占有资源时间过长
既然无法避免，解决方案是：降低事务粒度，提高事务的执行效率——死锁的机率降低

给外键加上索引是常见做法。例外：不容易改变的表（比如代码表）的外键可以不加
字段顺序可能会影响索引个数（外键索引容易导致这个问题），多索引插入会——慢

第六讲

在实际应用中，为了提升性能，可能不作外键约束

- 牺牲参考完整性
- 违反第三范式
- 冗余，多处修改
- 应用代码控制一致性
- 插入时需要进行检查

反向索引：解决数据存储的“热点”问题（将 12345 存成 54321），增强并发

- 争议：毫无意义的系统生成字段是否有意义？
- 合理性：
  - 理论上的唯一性不代表事实上的唯一性（身份证）
  - 大部分人不喜欢非数值型字段作为主键
  - 数值型快——插入时判定唯一性快

物理结构和 SQL 无关，使用 SQL 的好坏却受物理结构影响。

冲突：

- 数据紧凑 vs 数据分散
  - 数据的紧凑度和查询的效率正相关
  - 数据分散有利于并发
- 查询 vs 更新
  - 任何提高查询效率的手段，都会导致其他操作效率降低
  - 注重整体的效率，提高频繁事务的效率

中庸不是找中间点，而是找平衡点。

索引组织表（IOT）

在索引中增加额外字段，提高某个频繁运行的查询的速度。

“索引组织表(IOT)不仅可以存储数据，还可以存储为表建立的索引。索引组织表的数据是根据主键排序后的顺序进行排列的，这样就提高了访问的速度。但是这是由牺牲插入和更新性能为代价的(每次写入和更新后都要重新进行重新排序)。”

http://blog.csdn.net/dnnyyq/article/details/5195472

- 存储方式是有顺序的，而不再是随机的
- 仅针对主键有序，对其他操作不利
- 可以按范围查询
- 查询！查询！
- 仅 Oracle 提供，其他数据库中的不同机制：聚簇索引

数据分区（Partition）

本质：提供了一种数据管理的方法

效率提高取决于如何使用

典型例子：银行

- Motivation：主要的查询是按月份（当月）和年份（当年）为日期范围的
- 分 13 个区，前 12 个存储过去 12 个月的数据，最后一个存储所有以往数据
- 进入新的月份，增加新的分区，并将原来第 12 区和 13 区合并——滑动窗口

- 数据更密集，查询效率更高；潜在的并发性能降低
- 可以区分式存储，例如将最新数据存在高速硬盘中

第七讲

分区的核心是管理。

数据量小（几百万、几千万）时使用分区的意义不大，甚至带来并发性能降低；数据量大时，分区带来的查询性能提升超过并发性能的降低。

分区健的修改会引起数据的移动——避免

实例：有 T1、T2、……Tn 种类型的事务，每种事务有 W、P、D 三种状态

- 按事务类型分区：
  - 保证 n 个并发
  - 进程的等待降低
  - 每个分区中寻找特定状态的事务需要轮询
- 按状态分区
  - 状态转变有跨分区的移动
  - 移动分区时不需要锁住状态（读取即删除，写入即新增）——没有资源的并发冲突
  - 轮询的开销降低
  - 移动虽然有性能降低，但带来了额外好处

使用堆文件之外的任何存储方法，都会带来复杂性；错误的存储方式会带来大幅度的性能降低。

计算机的本质是对现实的简化，而这种简化就是模型。

树状结构

数据库中保存树状结构的模型：

- 邻接模型——保存 id 和 pid
- 物化路径模型——将层次路径作为 ID，如 1.1.2.4
- 嵌套集合模型——任意节点的位置通过左编号和右编号共同确定

第八讲

第二次作业：

如何在数据库中实现区间购买票？如 A、B、C、D 四个站点，乘客可以任意购买两站之间的票，如何查询特定时间 C、D 间有多少剩票？

如何查询车次的实例？如 G7000 次列车，有一辆车每隔一天开，还有一辆车每逢 10 的天数开，如何储存这些规则？如何查询？

一次发车配一个车组，包括驾驶员，乘务长，乘务员，乘警等，但一个车组不一定配套一辆车，如何查询某一天某一车次的车组？

晚点了如何记录？

树状结构（cont）

物化路径模型可以记录兄弟节点间的顺序。

	自顶向下查询	自底向上查询
邻接模型	使用数据库提供的递归查询	仍用递归查询[2]

[2] Oralce 中，由于使用了非关系的操作方式，即使使用 DISTINCT 也无法去除重复的祖先节点。

	功能（Oracle: connect by/start with;DB2: with+as）[1] 否则，很麻烦	
物化路径	只需查找所有起始部分相同的 ID（ID 是字符串） 查询简单，但没有邻接模型效率高	只需逐渐去除 ID 的尾部层次
嵌套集合	只需查找在目标节点范围之内的节点 单纯靠范围，节点的深度难以计算	层次显示依然是个问题

性能上来看：

- 邻接模型+connect by，自顶向下和自底向上的效率几乎相同==>过程式而非关系式
- 物化路径自底向上的性能降低很多，因为自顶向下只从一个节点出发，而自底向上需要从多个节点出发
- 嵌套集合两种查询的效率差不多，

结论：SQL 查询的效率还同数据库的设计相关。

LIKE 无法使用索引。但一些数据库提供额外的特性解决这个问题，比如 MySQL 的 substring index。

第九讲

数据库方法学

方法学只是对过去成功经验的总结，**不保证**开发的成功。

数据库 over 文件系统：

- 处理并发修改的最好方法

关联类：

空值多的列作为特殊属性分开存储

正常是数据库设计完成之后，可以开始逆范式和反范式的设计。

1. 合并 1 对 1 关系
2. 1 对多关系冗余一些非主键字段，减少查询时的 JOIN 操作（最常使用）
   a) UI 级别的一致性控制，而非数据库级别控制
3. 1 对多关系冗余外键，减少 JOIN
4. 多对多关系冗余字段，减少查询时 JOIN
   a) 多表连接变为更少的表连接
   b)
5. 针对多值属性，引入重复组，减少表连接
   a) 比如，一个用户有多个号码，正常方法是建立一个号码表，外键到用户表
   b) 解决方法：在用户表中存常用号码，其他号码还是存号码表
6. 使用提取表，将表展开成多维（复制多张表），提高效率
   a) 更新只放在更新表中，检索只放在检索表中

[1] Oracle 中 connect by 的实现不是基于关系，而是提取所有相关记录再处理（基于过程）；with 是利用关系的递归实现，但非常复杂，为了达到和 connect by 相同的效果，需要做很多附加处理。

b) 数据仓库的来源
c) 约束：查询对实时性的要求不高

7. 分区