



# 数据库开发技术

南京大学软件学院

2013 秋

# Instructor

- 刘嘉
  - Email : [liujia@software.nju.edu.cn](mailto:liujia@software.nju.edu.cn)
  - Office : 911

# Objectives

- 高性能数据库开发原则
- SQL中的优化
  - 优化SQL提高数据库应用效率
  - 优化索引提高数据库应用效率
  - 优化物理结构提高数据库应用效率
  - 优化数据库设计提高数据库应用效率
- SQL中其他的主题
  - SQL的经典模式
  - SQL中的分布式、动态定义；
  - 并发、大数据量等专题
- 课程实践

# Grading

- Two marked assignments 40%
- Final Exam 60%

# Reference Books

- Database Management Systems, Third Edition
  - [Raghu Ramakrishnan ; Johannes Gehrke]
- Database Solutions : A Step-by-Step Guide to Building Databases , Second Edition
  - [Tomas M. Connolly ; Carolyn E. Begg]
- The Art of SQL
  - [Stephane Faroult; Peter Robson]
- Expert Oracle Database Architecture : 9i and 10g Programming Techniques and Solutions
  - [Thomas Kyte]

# Reference Books

- SQL Clearly Explained (Third Edition)
  - [Jan L.Harrington]
- Oracle DBA手记 1.2.3.4
  - [盖国强(Eygle)等]
- Pro Oracle SQL
  - [Karen Morton...]

# 你应该了解的

- 关系代数
  - 选择 (select)、投影 (project)、连接 (join)、联合 (union)、差 (difference)、乘积 (product)
- DB, DBMS, 基于数据库的应用程序
- 数据库的基本特性 (表、KEY、完整性约束、锁、视图、事务...)
- SQL: 基本的DDL, DML, 触发器, 存储过程等的语法和基本用法
- 数据库设计的基本原则

# 开发成功数据库应用的要点

- 需要理解数据库体系结构
- 需要理解锁和并发控制特性
  - 每个数据库都以不同的方式实现
- 不要把数据库当“黑盒”
- 性能、安全性都是适当的被设计出来的
- 用尽可能简单的方法解决问题
  - “创造”永远追不上开发的步伐
- DBA和RD之间的关系



# Chapter I 为性能而设计

- 应用系统的首要目标
  - 满足业务需求 (business requirement)
- 实际的情况是
  - 在技术挑战的刺激下
  - 忽视目标
  - 重视手段
  - 忽视数据的质量
  - 重视按期交付功能
- 强调良好的模型和合理的数据库设计是任何信息系统的基础

# 数据的关系视图

- 数据库只是对现实世界的有限描述
- 对特定的业务活动的描述不止一种
- “关系模型”中的“关系”的含义
- 关系模型的一致性
  - 只要遵守关系理论，可以保证基于数据库的任何查询结果与原始数据具有同样的有效性
  - 关系理论包括
    - 关系不包含重复数据
    - 记录之间没有顺序

# 规范化 (Normalization)

- 规范化→ 枯燥？沉闷？像古典文学？
- 1NF 确保原子性 (Atomicity)
  - 原子性的粒度、原子性的价值
- 2NF 检查对键的完全依赖
  - 价值在于控制数据冗余和查询性能
- 3NF 检查属性的独立性
- 规范化的价值
  - 合理规范化的模型可应对需求变更
  - 规范化数据重复降至最少


# 有值、无值、空值

- 表中的每一条记录都应该是特定“事物”的状态描述，如果大部分特征信息都显示“我们不知道”，无疑大大降低了信息可信性
- 存在空值意味着关系模型存在严重的问题，动摇了查询优化的基础
- 空值对程序逻辑是危险的；必须使用空值的话，一定要清楚它在特定情况下的影响。

# 限用Boolean型字段

- SQL中并不存在Boolean类型
- 实现flag表示标志位的Y/N或T/F
  - 例如：order\_completed
  - 但是...往往增加信息字段能包含更多的信息量
  - 例如：completion\_date completion\_by
  - 或者增加order更多状态标示
- 极端的例子：四个属性取值都是T/F，可以用0-15这16个数值代表四个属性所有组合状态
  - 技巧可能违反了原子性的原则
  - 为数据而数据，是通向灾难之路

# 理解子类型 (SubType)

- 表过“宽” (有太多属性) 的另一个原因，是对数据项之间的关系了解不够深入
- 一般情况下，给子类型表指定完全独立于父表主键的主键，是极其错误的




# 约束应明确说明

- 数据中存在隐含约束是一种不良设计
- 字段的性质随着环境变化而变化时设计的错误和不稳定性
- 数据语义属于DBMS，别放到应用程序中



# 过于灵活的危险性

- “真理向前跨一步就是谬误”
- 不可思议的四通用表设计 
  - Objects(oid, name), Attributes(attrid, attrname, type)
  - Object\_Attributes(oid, attrid, value)
  - Link(oid1, oid2)
- 随意增加属性，避免NULL
- 成本急剧上升，性能令人失望



# 如何处理历史数据

- 历史数据：例如：商品在某一时刻的价格
- Price\_history
  - (article\_id , effective\_from\_date , price)
- 缺点在于查询当前价格比较笨拙
- 其他方案
  - 定义终止时间
  - 同时保持价格生效和失效日期，或生效日期和有效天数等等
  - 当前价格表+历史价格表


# 设计与性能

- 调优 (tuning)
  - 在目前情况下优化性能至最佳
- 性能拙劣的罪魁祸首是错误的设计

# 处理流程

- 操作模式 (operating mode)
  - 异步模式处理 (批处理)
  - 同步模式处理 (实时交易)
- 处理数据的方式会影响我们物理结构的设计

# 数据集中化 (Centralizing)

- 分布式数据系统复杂性大大增加 
  - 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
    - Copy的数据传输开销
    - 无法从数据规划中获益（物理结构，索引）
- 数据库该如何部署呢？
  - 中庸、分析、决策
- 离数据越近，访问速度越快



# 系统复杂性

- 数据库的错误很多
  - 硬件故障
  - 错误操作...
- 数据恢复往往是RD和DBA争论焦点
  - DBA，即便确保数据库本身工作正常，依然无法了解数据是否正确
  - RD，在数据库恢复后进行所有的功能性的检查

# 小结

- 错误的设计是导致灾难性后果的源泉
- 解决设计问题会浪费惊人的精力和智慧
- 性能问题非常普遍
- 打着“改善性能”的旗号进行非规范化处理，常常使性能问题变得更糟
- 成果的数据建模和数据操作应严格遵循基本的设计原则。

# 课堂作业

- 请根据实际工作经验描述数据库开发中涉及到的高性能数据库设计的内容
- 可以包括但不仅包括
  - 数据库设计
  - 分布式或集中式的选择
  - 批处理或者实时系统的选择
  - 数据库恢复和备份方案
  - .....

# Chapter 2 高效访问数据库

- 开发环境切换到生产环境是一场战役
- 常常忽略从大局上把握整个架构和设计
- 本章讨论编写高效访问数据库的程序需要实现哪些关键目标



# 查询的识别

- 定位被执行的SQL比较容易
- 确定哪些应用提交了这些SQL比较困难
- 养成为程序和关键模块加注释的习惯
- 易识别的语句有助于定位性能问题

# 保持数据库连接稳定

- 连接稳定
- 减少交互
- 所有的事情，试试吧~~~~~（课后练习）
  - Open the file
    - Until the end of file is reached
    - Read a row
    - Connect to the server specified by the row
    - Insert the data
    - Disconnect
  - Close the file
  - 如果依次对每一行作连接/中断 7.4行/秒
  - 连接一次，所有行逐个插入 1681行/秒
  - 连接一次，以10行为一数组插入 5914行/秒
  - 连接一次，以100行为一个数组插入 9190行/秒

# 战略优先战术

- 不要被眼前的细微所迷惑
- 着眼于最终结果
- 考虑解决方案的细节之前，先站的远一些，把握大局

# 先定义问题，再解决问题

- 一知半解是危险的
- 所有技术方案都是我们达到目标的手段
- 新技术本身并不是目标

# 保持数据库Schema稳定

- 在应用程序中使用DDL建立、修改或删除数据库对象，是很差的方式。
  - 除了分区和DBMS已知的临时表之外...
- 数据字典是所有数据库操作的中心
- 任何对数据字典的操作都会引起全局加锁，对系统性能影响巨大。
- 不应在应用程序中进行数据库对象的建立、修改及删除等操作，虽然设计应用的时候需要考虑这些。


# 直接操作实际数据

- 临时工作表 (temporary work table)
- 永久表 (permanent table)
- 查询临时表的语句效率比永久表低
  - 永久表可以设置非常复杂的存储选项
  - 临时表的索引一定不是最优的
  - 查询之前需要为临时表填入数据
- 暂时工作表意味着以不太合理的方式存储更多信息

# 用SQL处理集合

- SQL完全基于集合来处理数据
- 将一次“大批量的数据处理”分割成多次“小块处理”是一个坏主意
  - 占用过多的空间保留原始数据，以备事物回滚之需
  - 万一修改失败，回滚消耗过长的时间

# 动作丰富的SQL语句

- SQL不是过程性语句 
- 不要混淆声明性处理和过程逻辑
  - 最常见的例子出现在从数据库中提取数据，然后循环处理并与数据库再进行交互
- 避免在SQL中引入“过程逻辑”的主要原因
  - 数据库访问，总会跨多个软件层，甚至包括网络访问
  - 在SQL中引入过程逻辑，意味着性能和维护问题应由你的程序承担



# 充分利用每次数据库访问

- 从一个表中提取多段信息，采用多次数据库访问的方法非常糟糕
- 在合理的范围内，利用每次数据库访问完成尽量多的工作

# 接近DBMS核心

- 代码喜欢SQL内核→ 离开核心越近，它就运行的越快



# 只做必须做的

- 没有必要编程实现那些数据库隐含实现的功能

- 例如

- Count(\*) 用以测试 “是否存在”

Select count(\*)

Into counter

From table\_name

Where ...

If (counter > 0) then.....

# 慎用自定义函数

- 自定义函数
  - Select中，被调用一次
  - Where中，可能每行调用一次
  - 如果自定义函数内部还执行了一个select.....则无法被基于成本的查询优化器优化



# SQL的进攻式编程

- 防御式编程 (code defensively)
  - 在开始处理前先检查所有参数的合法性
  - 例子：进行一连串检查，出现不符合就产生异常。以信用卡检查为例（身份、卡号、是否过期、支付额度）
  - ```
Select count (*)  
From customers  
Where customer_id = provided_id
```
  - ```
Select card_num, expriy_date, credit_limit  
from accounts  
where customer_id = provided_id
```

# SQL的进攻式编程

- 进攻式（假设today()返回当前日期）
- Update accouts  
Set balance = balance – purchased\_amount  
Where balance >= purchased\_amount  
And credit\_limit >= purchased\_amount  
And expiry\_date > today()  
And customer\_id = provided\_id  
And card\_num = provided\_cardnum
- 接着检查被更新的行数，如果结果是0，则失败

# SQL的进攻式编程

- Select c.customer\_id, a.card\_num, a.expiry\_date, a.credit\_limit, a.balance

From customers c

Left outer join accounts a

On a.customer\_id = c.customer\_id

And a.card\_num = provided\_cardnum

Where c.customer\_id = provided\_id

- 如果此查询没有返回数据，customer\_id是错的
- 如果card\_num是Null则卡号错误

# SQL的进攻式编程

- 进攻式编程的本质特征
  - 以合理的可能性（reasonable probabilities）为基础
  - 类似DBMS采用的乐观并发控制
  - 乐观的方法比悲观的方法的吞吐量高的多





# SQL的进攻式编程

- 勇敢和鲁莽的界限很模糊
- 进攻式编程要对异常进行谨慎控制
- 防御式编程才是可以构建一个真正可移植的数据库应用