开发成功数据库应用的要点-黑盒的问题

- 上一讲，留下一个思考题
  - 对大多数码农而言，数据库锁机制好像都是自动和透明实现的，那么深入了解每个数据库的锁机制实现细节，对码农编码有什么影响嘛？

问题是：这对我们码农有什么影响吗？

- Oracle的无阻塞设计有一个副作用，就是如果确实想保证一次最多只有一个用户访问一行数据，就得开发人员自己做一些工作

例如：一个资源调度程序主要有两张表：

Resources（Resource_name，other_data）

Schedules(resource_name, start_time, end_time)

往Schedules中插入一个房间预订之后，提交之前，应用将查询

```
Select count(*)
From schedules
Where resource_name = :resource_name   FOR UPDATE
        and (start_time < :new_end_time) and (end_time > :new_start_time)
```

Test:

A：(701，10:00，12:00)

B：(701, 11:00，12:00)

不能把数据库当成黑盒使用

必须深入了解你所使用的数据库的体系结构和特征

什么时候，码农需要自己考虑并发的问题？

不知道，看情况

黑盒和数据库独立性的问题

- 数据库有脱离实现级别的使用方法
- 我的观点是
  - 要构建一个完全数据库独立的应用，而且是高度可扩展的应用是极其困难的
  - 实际上，这几乎是不可能的
- 要构建一个完全数据库独立的应用
  - 你必须真正了解每个数据库具体如何工作
  - 如果你清楚每个数据库工作的具体细节，你就会知道，数据库独立性可能并不是你真正想要的

例如：Null值造成的数据库迁移障碍

例子：在表T中，如果不满足某个条件，则找出X为NULL的所有行，如果满足就找出X等于某个特定值的所有行。

```
Declare
  L_some_variable varchar2(25)
Begin
  If(some_condition)
  Then
    L_some_variable := f(…);
End if;
For x in (select * from t where x=l_some_variable)
Loop
…
```

*Select * from t where(x = l_some_variable OR(x is null and l_some_variable is NULL)*

select * from t where nvl(x,-1) = nvl(l_some_variable,-1)

创建一个基于函数的索引：create index t_idx on t(nvl(x,-1))

关于黑盒的问题总结几点

- 数据库是不同的。在一个数据库上取得的经验也许可以部分应用于另一个数据库，但是必须有心理准备，二者之间可能存在一些基本差别，可能还有一些细微的差别。
- 细微的差别（比如对NULL的处理）与基本差别（如并发控制机制）可能有同样显著的影响。
- 应当了解数据库，知道它是如何工作的，他的特性如何实现，这是解决这些问题的唯一途径。

我能不能找一个大牛帮我调优

- 先把程序写出来，之后再让专家在生产环境中帮我调优
  - 这个想法是错误的.....
- 性能调优（目前情况下性能优化至最优）
  - 根据当前CPU能力、可用内存、I/O子系统等资源情况来设置相应参数
  - 通过索引、物理结构、SQL的优化，具体提高某一个查询的性能

如果有个专家能通过一些参数、技巧提高了你的系统一个数量级的性能，
不能说这个专家牛逼，大概只能说明你的程序太烂了。

性能拙劣的罪魁祸首是错误的设计

- 提高整体性能
  - 技巧决定系统性能的下限
  - 设计决定系统性能的上限
- 比如，新闻的门户网站
  - 动态页面vs静态页面
  - 静态页面+内容管理系统

性能优化要考虑整体

- 性能指标都是有成本的、安全和优化中寻找平衡
- 性能指标以吞吐量为核心（每秒处理多少事务）
  - 而尽量不用一个事务几秒能处理完成
- 性能指标要考虑整体性
  - 优化手段本身就有很大的风险，只不过你没意识到罢了
  - 任何一个技术可以解决一个问题，但必然存在另一个问题的风险
  - 对于带来的风险，控制在可接受的范围才是有成果
  - 性能优化技术，使得性能变好，维持和变差是等概率的事件

使用优化工具

MySQL常用的工具
MySQLadmin
MySQLshow
SHOW [SESSION | GLOBAL] variables
SHOW [SESSION | GLOBAL] STATUS
Information_schema
SHOW ENGIN INNNODB STATUS
SHOW PROCESSLIST
Explain
Show index
Show log

MySQL不常用，但是好用的工具	
zabbbix	监控主机、系统、数据库
Pt-query-digest	分析慢日志
MySQLslap	分析慢日志
sysbench	压力测试工具
MySQL profiling	统计数据库整体状态工具
Performance Schema	性能状态统计的数据
Workbench	管理、备份、监控、分析、优化工具

整体层面的性能优化考虑

- 问题一：cpu负载高，io负载低
  - 内存不够
  - 磁盘性能差（磁盘问题、raid设计不好、raid降级）
  - **SQL的问题**
  - **并发锁机制的问题**
  - **事务设计问题，大量小数据IO**
  - **大量的全表扫描**

整体层面的性能优化考虑

- 问题二：IO负载高，CPU负载低
  - **大量小的IO执行写操作**
  - **Autocommit，产生大量小IO**
  - **大量大的IO执行写操作**
  - **SQL的问题**
  - IO/PS磁盘限定一个每秒最大IO次数

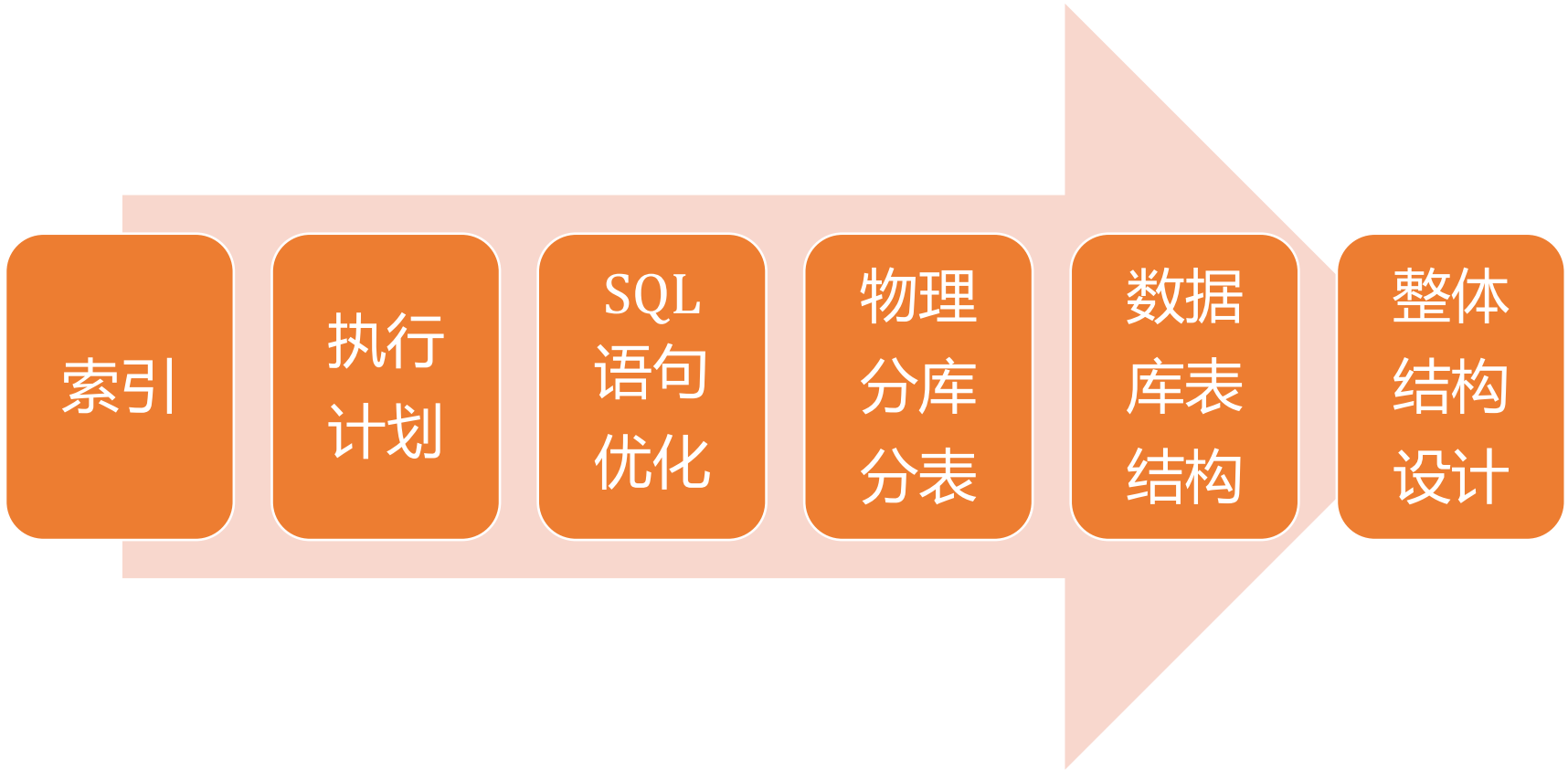
整体层面的性能优化考虑

- 问题三：IO和CPU负载都高
  - 硬件不够用了
  - **SQL存在问题**

性能问题，90%的问题来源都是程序员的问题

开发环境到生产环境是一场灾难

SQL优化的方向



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

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

Practice in class 2-2

- 你对你常用的关系数据库关系系统中，去寻找一些针对优化的工具，去尝试使用一些性能的分析和监控工具（查看数据库官方Reference，首先使用官方的命令和工具）

Practice in class 2-3

- 关于把数据库当成黑盒使用的错误，其实也会在你学习软件开发中遇到类似的问题，比如，对操作系统的黑盒化，比如对某些开发框架的黑盒化等等，请你思考一下，你的学习过程中，还能找到类似的例子嘛？

End

下一节课再见