SYBASE
TECHWAVE
ASIA PACIFIC CONFERENCE 2009

SQL Anywhere 使用及开发最佳实践

王军
Sybase iAnywhere

目 标

- 帮助您开发应用系统
  - 高生产效率
  - 优秀设计
  - 高性能
  - 高扩展性

欢迎加入我们的论坛

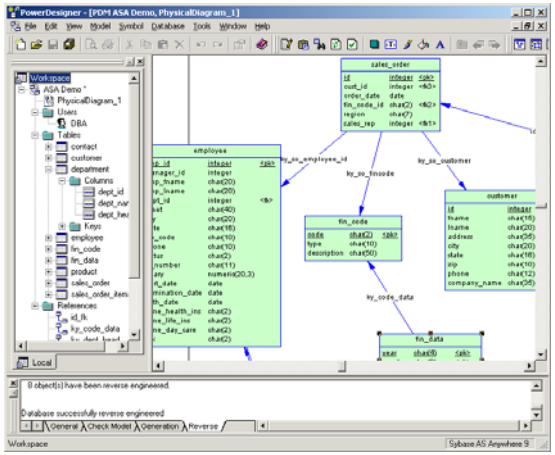
- **如果你有开发的心得, 不管是与数据库服务器有关的, API 有关的, 数据同步有关的, 以及其它任何方面的:**
  - 发表到我们的论坛上
    - sybase.public.sqlanywhere.general

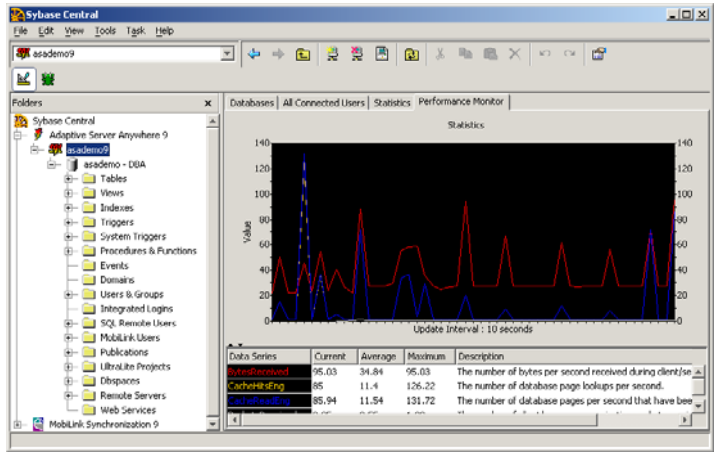
议题

- SQL Anywhere 概述
- 总体设计原则
  - 开发计划设计技巧
  - 应用开发技巧
- 一些技术细节
  - 事物隔离级别
  - SQL Anywhere支持的游标(cursor)
  - 物化视图
- MobiLink性能优化

SQL Anywhere主要组件

- **关系数据库系统**
  - –Adapative Server Anywhere
  - –UltraLite
  - –UltraLiteJ
- **数据同步技术**
  - –MobiLink
  - –QAnywhere
  - –SQL Remote
- **其它**
  - –Web Service 和 XML
  - –管理工具
  - –设计：PowerDesigner
  - –开发：Dbisql，索引顾问，存储过程调试器
  - –报表工具：InfoMaker，Datawindow.NET

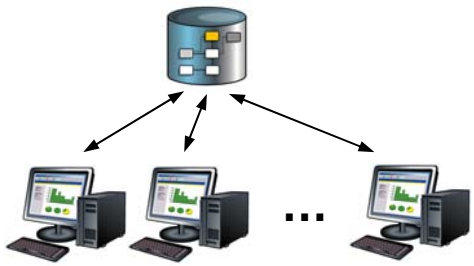


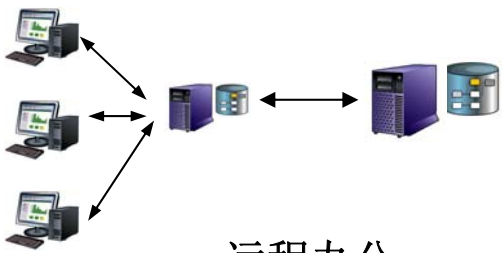


工作场景

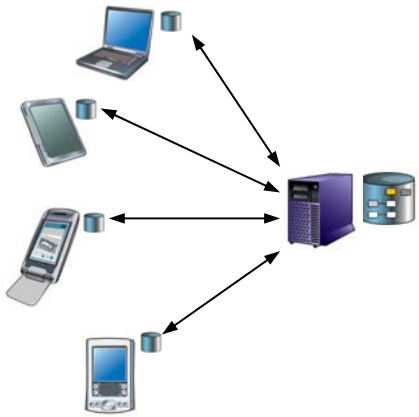
单机版

服务器





远程办公



移动环境

- 中小型企业
  - 财务
  - 管理
- 解决方案
  - 电话系统
  - 销售点
- 远程办公
  - 存储管理
  - 资产控制
- 移动
  - 移动销售
  - 现场服务
  - 检查/ 巡检

议题

- SQL Anywhere 概述
- 总体设计原则
  - 开发计划设计技巧
  - 应用开发技巧
- 一些技术细节
  - 事物隔离级别
  - SQL Anywhere支持的游标(cursor)
  - 物化视图
- MobiLink性能优化

什么时候需要考虑性能与扩展性?

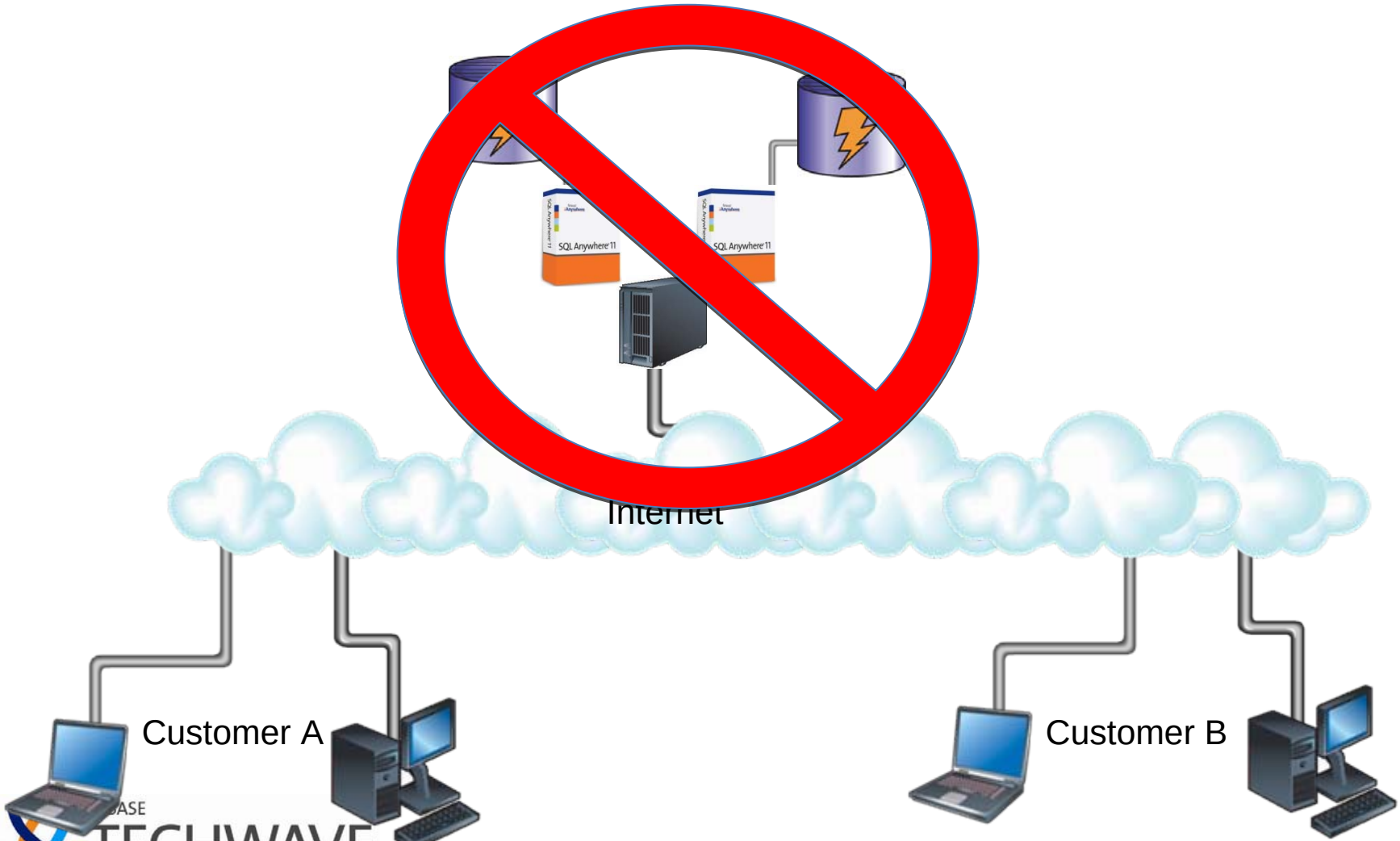
- 在设计与计划阶段
- 应用系统的规模设计阶段
- 提升性能以部署数据库时

越早越好!

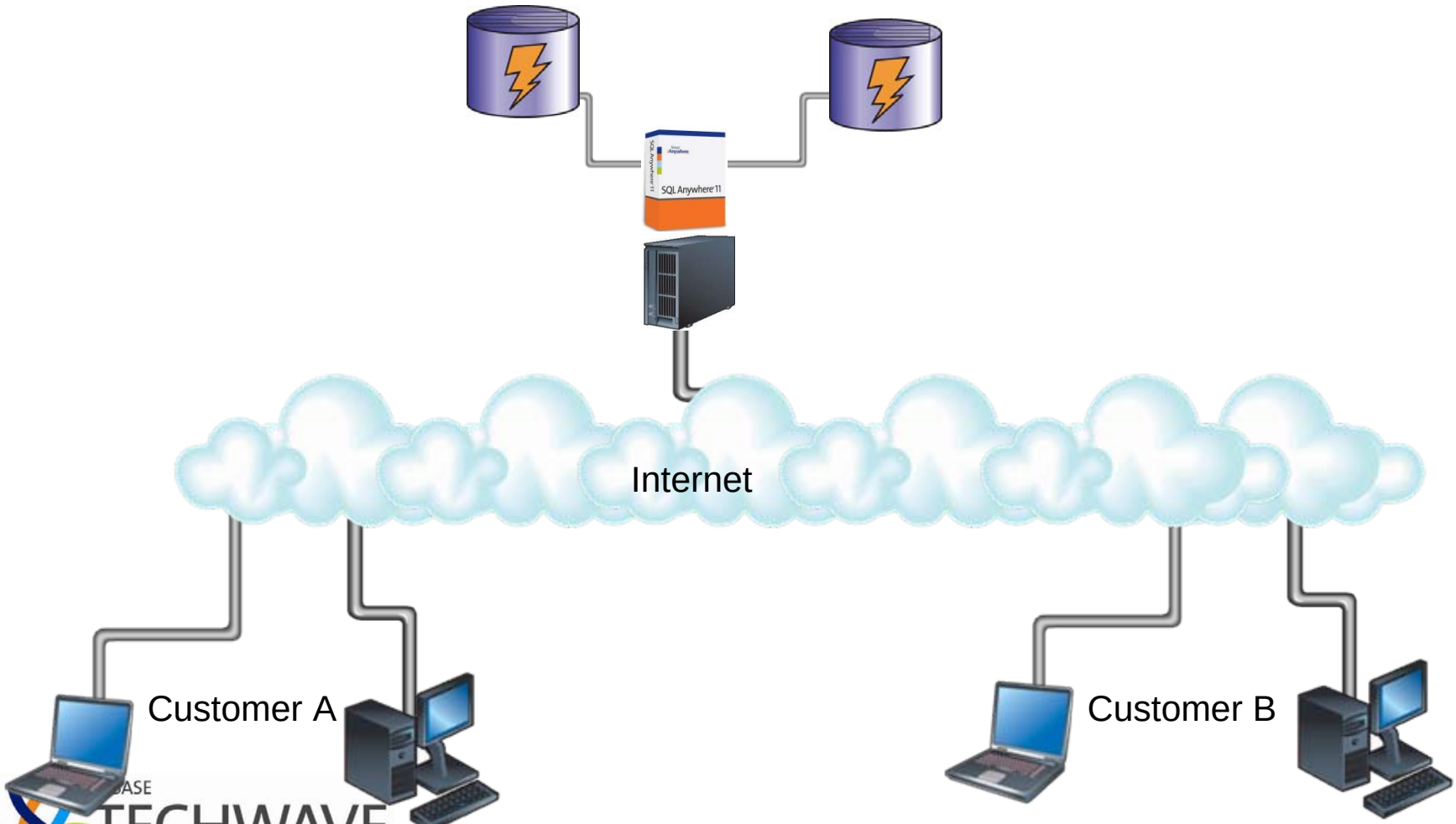
跟性能有关的几个方面

- 数据库的物理组织有关
  - 数据库文件的特性
  - 索引
- 数据库模式设计
- 服务器特性有关
  - CPU, 磁盘的读写
- 网络有关
  - 带宽
  - 网络延迟
- 应用系统设计
  - 服务器－客户端通讯的效率
  - 查询的复杂程度
  - Trigger 设计
  - 锁的考虑
  - 工作站的处理情况 (CPU, 磁盘等)

多数据库部署在同一台服务器上



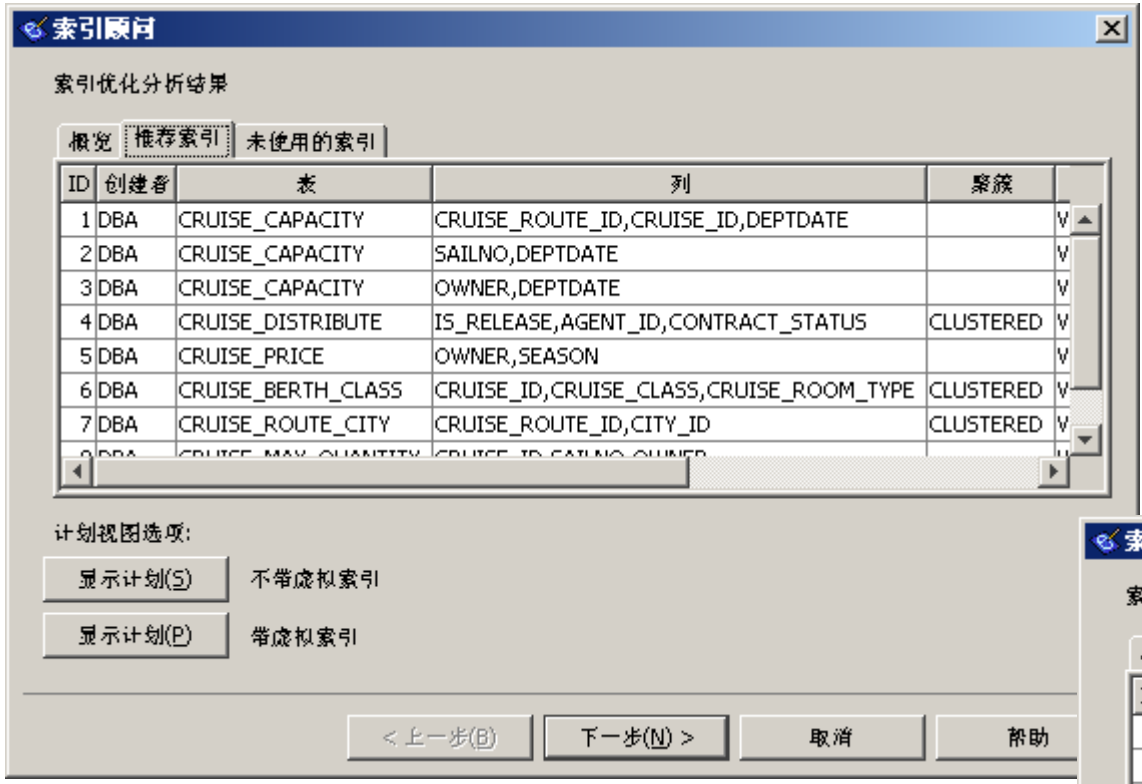
多数据库部署在同一台服务器上

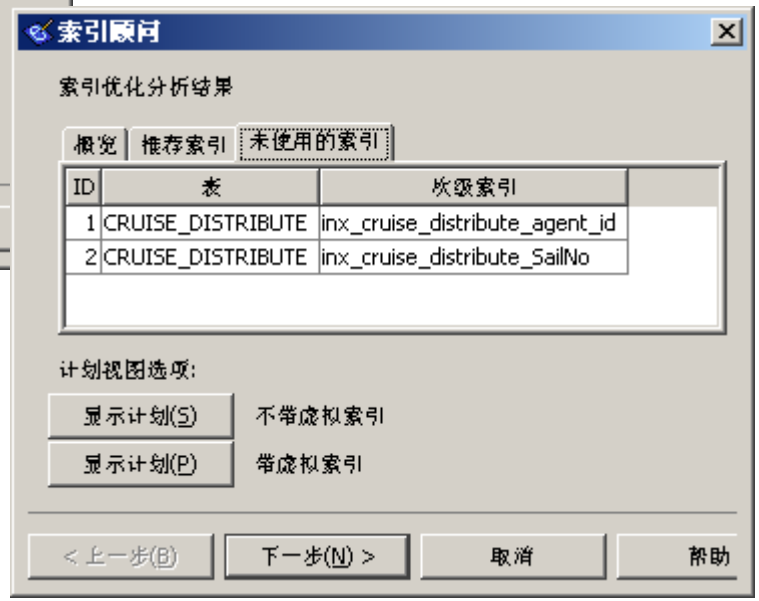


模式（Schema）设计

- 定义表
  - 规范化数据
  - Entity/Relationship (ER) 设计
- 对所有表定义合适的主键（Primary Key）
  - 对于有复制的环境是很有用的 (可以减少传送的事务日志)
- 定义合适的外键关系
  - 外键有利于关联查询的最优化
- 定义合适的索引
  - 对于主键、外键不用建立索引（索引会自动创建）
  - 建立适量的索引，太多未必有好处
  - SQL Anywhere 允许定制外键索引
    - 列的顺序, 排序的顺序(ASC,DESC)等
  - 可以利用索引顾问得到推荐的索引建立策略

SQL Anywhere 索引顾问





模式设计: 主键（primary keys）

- **数据管理的注意事项:**
  - 保证应用系统控制主键的赋值及使用
  - 一般情况不更新主键 (特别是在复制环境中)
    - 一般不要使用电话号码, SSN/SIN 号码, 或其它外部标识作为主键值
  - 不让用户/客户看到主键值是很困难的
  - 一旦部署后，键(主键、外键等)的格式就不容易改变了

模式设计: 如何规划主键

- **一般情况下: 大的、复杂的主键很难有高的查询效率**
  - 数据更新及数据检索均会有较大影响
    - 定位1条记录需要多个标识条件
    - Group By, Order By 查询时需要对多个列进行检索
    - 建立索引时需要多个列，增加了磁盘占用空间
  - 考虑主键的替代; 转换为唯一性约束或普通索引
- **选择数据类型时要慎重**
  - Double 或 float, 因为它们不精确，一般不建议作为主键

模式设计: 如何规划主键

- **Integer 是效率最高的, 不管对于检索还是存储**
  - 比 DECIMAL类型效率要高的多
  - 允许自动增长列作为主键; 不用通过应用系统来维护主键，简化了开发工作量
  - 自动增长列是非常好的; 在很多情况下都是有益的; 特别推荐在数据同步环境中使用
  - 全局自动增长列可在复制环境中产生唯一的主键值 (并且在UltraLite/UltraLiteJ数据库中也是这样！)
- **但是在也有例外….**

模式设计: 自动增长列作为主键

- 自动增长列的一些缺点:
  - 经常需要通过不同的主键值区分不同的商业对象
  - 对于一些应用系统可能需要随机生成主键值
  - 字母数字值可以帮助数据输入过程中保持数据的一致性，而自动增长列不好做到这一点(如:Manager001,Employee001)
  - 自动增长列不支持自检标识符
- 要综合考虑上面这些因素

自我检查约束标识

- **增加额外的字母(或数字）作为标识**
  - 例如: 加拿大社会保险号码
    - 8位数字 + 检验数字
    - 联邦政府发布检验规则，因此金融服务机构可以验证 SIN(social insurance number) 号码
    - 算法可以防止一个有效的数字的任何两位数字换位
  - 可以将字母+数字作为标识组合
- **GUIDs 是唯一性的, 但是它们冗长且不支持自检**

其它情况

- 美国信用卡号码
  - 账号与卡号是分离的
    - 个人可以拥有多张卡,附属卡
  - 前10位是卡号
    - 账号后面的5位数字为账号后缀
    - 当卡丢失时5位后缀可以更改; 主卡号码不变
- 加拿大邮政编码
  - 情况不同:不是自检查约束, 因为它们的格式复杂而难于分类 (如. N2L 6R2)
    - 避免了换拉错误
    - 特别是为加拿大邮局减少了不正确的邮寄地址情况的发生

非整型键的益处

- 通过简单的键格式即可区分关键商业对象
  - 当与外部团体联系时，特别是有关电话号码时，可以利用格式的好处
  - 如可以将 AAA-999 作为一种对象类型, 999-AAA 作为另外一个, 999-AAA-999 作为另外一个, 等等.
  - 如果利用字母, 不使用用元音从而避免生成误拼的单词
- 可以区分非法的键值与未知的键值
  - 对于客户服务机构很有用

模式设计: 主键生成 – 手工

- 如何生成这些标识?
- 手工分配键值范围
  - 麻烦, 但在某些商业环境中是可行的
  - 经常产生数据录入错误
    - 考虑自检查约束或 字母数字组合从而避免数据录入问题
      - 例如: 加拿大邮政编码
        - » 如： N2L 6R2
- 我们真的需要客户编码从000000001开始吗?

模式设计:主键生成 – 键表

- 建立 1 个独立的 “键值” 表, 每个商业对象对应 1 条记录
  - 增加新的键值:
    - 建立一个数据库的连接
    - 根据其对应的记录计算其下一个键值
    - 更新记录后，马上提交
  - 几个弊端: 需要额外的连接, 日志, 锁, 可能产生争用
- 注意: 避免连续事务的设计

这样设计是不好衡量的(特别是针对于争用等)

模式设计:主键生成– 键池

- 对每一个潜在商业对象建立一个单独的永久表 (池)
- 每一个事务从该表中删除1个键值，将该键值插入到实际表中
  - 回退时，键值又回到池中
  - 当池中的键值快用完时，可以通过触发器、事件等增加池中的键值

物理模式设计的问题

- 为了得到高的性能, 列的顺序可能是很重要的
  - 原则: 将经常访问的列放于1行的前面
  - 对于字符类型列的定义，通过 INLINE及PREFIX 参数控制大对象的存储
- 复合外键索引的列顺序不一定非要跟主键一致
- 使用 PCTFREE 减少内部碎片(缺省为：4 KB（及以上）的页面大小应用 200 字节)

复合外键索引的列顺序不一定非要跟主键一致

- 主表

```
CREATE TABLE tbCustomer (
  Id                INTEGER ,
  name              varchar(20),
  tel               varchar(20),
  address           varchar(80),
  Constraint Pkey_tbCustomer Primary key(name,tel)
);
```

- 从表

```
CREATE TABLE tbOrder (
  Id                INTEGER PRIMARY KEY,
  name              varchar(20),
  tel               varchar(20),
  mount             int,
  sum               numeric(10),
  FOREIGN KEY (tel,name) REFERENCES tbCustomer ( tel,name )
);
```

物理模式设计: 外键

- 外键对于复杂查询的优化是至关重要的
  - 当有外键时，连接查询的优化器选择及基数估算会更准确
  - 也能引起多种查询优化的重写
- 但是要权衡使用参考完整性
  - 在查询过程中不会使用的索引，照样需要做索引的维护
    - 减少某些物理索引，而是使用共享索引，可以减少索引的维护
  - 在特定的情况下，当应用系统经过充分测试后，可以考虑删除参考完整性约束及检查性约束

物理模式设计: 实体类型层次结构

- **ETH(Entity-type hierarchies): 具有多个子类型的对象, 例如:**
  - 保险客户: 所有人, 付款人, 被保人, 受益人
  - 投资: 股票, 基金, 定期存款, 现金, 分红
- **可以是一个非常有用的数据模型**
- **ETH 的实施或许是模式设计选择中最难的**
  - 没有正确答案，只有根据实际情况的权衡
  - 规范化的解决方案是很麻烦的, 并且有可能每次访问时都会关联查询

物理模式设计: 实体类型层次

- **设计选择:**
  - 1个物理表,对每个子类在每行中有不同的可用属性
    - 子类标识存储在每1行中，通常在每次查询时都需要通过条件指定
      - 可以通过已经指定该条件的视图访问
    - 定义完整性约束将更难
      - 2个或更多对象具有相同的外键
      - 可能需要权衡2个或多个子对象具有相同的外键, 这有可能会影响到Update操作

物理模式设计: 实体类型层次

- **设计选择:**
  - 多个物理表, 每个子对象1个表
    - 只涉及到1个子对象的查询是准确的
      - 不需要额外的条件、映射
    - 键值的产生更困难，因为需要在所有子对象中保持唯一
    - 需要多个子对象的查询需要 UNION 查询操作
    - 需要2个或多个子对象的查询需要 OUTER JOINs 操作

Client-server 应用系统的性能

- 都是关于减少响应时间
- 没有真正正确的答案，只有权衡

影响响应时间的因素

- 服务器端的响应时间
- 网络的响应时间
  - 通过网络的来回时间，可以通过 DBPing 命令估算
- 低效率的 client-server 交互
  - 应用系统发出太多的请求
    - 不是所有的请求是应用必须的; 有些只是占用了网络
  - 每次请求时太多 (或太少) 的数据通过网络传送
  - 多次请求相同的数据
  - 重复发出类似的请求，而不是重复使用类似的请求

服务器的响应时间

- 无论何时请求受阻, 都会增加响应时间
  - 例如:
    - 查询执行计划的响应时间
      - 例如, 使用用户自定义函数（ user-defined functions (UDFs)）或 使用子查询
    - 由于锁的争用而产生阻塞
      - 通过应用系统设计或事务隔离级别来控制
    - 由于内部并发控制机制或共享服务器对象的争用

查询执行计划的响应时间

- **任何查询操作的中止都会增加服务器的计算/估算时间; 例如:**
  - 嵌套查询时经常中断从表中取数据的操作
  - 全表扫描的子查询的估算将有非常大的开销
    - SQL Anywhere服务器将用很长的时间通过大量内存及查询重写来减少子查询的估算时间
  - 从扩展页面中取数据将会中断从基本页面中取数据

查询执行计划的响应时间

- 如何写SQL语句将是至关重要的
  - 仔细检查关联条件,包括用户自定义函数,表达式,以及类型转换
  - 用户自定义函数中的SQL查询会影响到查询最优化,可能会影响SQL执行效率
  - 考虑使用 Window Function,而不是使用嵌套子查询, 从而避免慢的子查询估算时间
  - 只从必须的表中取必须的字段

查询执行计划的响应时间

- 尽可能简化查询的语法
  - 通过查询列表的别名来标识表达式(包括子查询)是很有用的
    - 如, Select (X+10)/2 as quotient
  - 排除不必要的条件, DISTINCT 处理, 关联查询, 等等.
  - 不要用子查询代替 LEFT OUTER JOIN 查询
    - 子查询不能被优化器重写
    - LEFT OUTER JOIN 查询能有多种方法执行查询
  - 在查询中使用用户自定义函数(user-defined functions (UDFs)) 会降低查询性能
    - 只在必须时使用; 但要权衡利弊

设置查询优化参数

- optimization_level（缺省为9）
  - 控制 SQL Anywhere 查询优化程序为查找 SQL 语句的访问计划而消耗的资源量
  - 我们建议您不要更改 optimization_level 的缺省 PUBLIC 设置。
  - 当查询SQL非常复杂时，优化程序将更积极地搜索备用策略，因而可能在优化阶段花费更多的时间。
  - set option public.optimization_level='0'

Demo

Window functions

- 提供了另外一个在相同查询条件下执行 GROUP BY 的机会
  - 允许各种各样的需要多个查询以保留中间结果的复杂查询
- 请见白皮书
  http://ianywhere.com/developer

使用 WINDOW 函数

- **原 SQL 查询语句(查询库存量小于最大订单量的产品及订单):**

```
Select o.id, o.orderdate, p.*
From salesorders o, salesorderitems s, products p
Where o.id = s.id and productid = p.id
and p.quantity < (Select max(s2.quantity)
                  From salesorderitems s2
                  Where s2.productid = p.id)
Order by p.id, o.id
```

结果:743条记录

使用 WINDOW 函数

- **使用 WINDOW 函数重写查询的ＳＱＬ语句(查询库存量小于最大订单量的产品及订单):**

```
Select  order_qty.id, order_qty.max_q,o.orderdate, p.*
From (Select s.id, s.productid,
      Max(s.quantity) Over (Partition by s.productid
            Order by s.productid) as max_q
      From salesorderitems s) as order_qty,
      products p, salesorders o
Where p.id = productid and o.id = order_qty.id
      and p.quantity < max_q
Order by p.id, o.id
```

结果:743条记录

议题

- SQL Anywhere 概述
- 总体设计原则
  - 开发计划设计技巧
  - 应用开发技巧
- 一些技术细节
  - 事物隔离级别
  - SQL Anywhere支持的游标(cursor)
  - 物化视图
- MobiLink性能优化

隔离级别

- **隔离级别只影响来自其它连接/事务的读请求; 写总是会产生锁**
- **读请求的隔离级别:**
  - 0– 没有阻塞;
  - 1 – 别的用户正在修改您想查询的记录时,要等待
  - 2 – 其它事务不能更改符合您的条件的行
  - 3 – 只能有1个用户修改表记录，不能串行读
  - 快照隔离级别 – 写不阻塞读, 可以得到已提交数据的拷贝
- **增加/删除1个外键所在的行需要在主键表上加1个只读锁**

快照隔离级别的支持

- 提供当其它事务中并发写数据库时，数据读取的连续性 (例如， 写不阻塞读)
- 通过设置全局数据库选项, allow_snapshot_isolation
- 3种新的事务隔离级别:
  - “快照” – 能看到其它事务开始时的数据 (第1行被访问/修改开始前)
  - “语句快照” – 需要较少的数据库开销, 但每个语句快照只能看到不同时间点的数据（另外1个事务提交后，才可以看到其修改的数据，写不阻塞读）
  - “只读语句快照” – 在查询上类似于“语句快照”; 当前事务中不能读取到其它事务中已经提交的数据（必须开始新的事务才可以读取到其它事务提交的数据）

快照隔离级别的支持

- **使用是用代价的**
  - 通过“行版本存储”（“row version store”）来维护旧版本数据的拷贝 (数据库临时表空间的一部分) 从而维护数据库事务的一致性
  - 旧版本数据的拷贝会被数据库的“ cleaner” 进程清除
  - 索引将会是新值与旧值的组合
    - 会影响按顺序扫描及索引扫描的性能
- **设置快照隔离级别:**
  - set option isolation_level = 'snapshot'

    或 SET OPTION public.allow_snapshot_isolation = 'ON';
    - - SET TRANSACTION ISOLATION LEVEL STATEMENT SNAPSHOT;
    - - SET TRANSACTION ISOLATION LEVEL READONLY STATEMENT SNAPSHOT;
- **或在应用中通过ODBC设置**
  - SA_SQL_TXN_SNAPSHOT
  - SA_SQL_TXN_STATEMENT_SNAPSHOT
  - SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT

快照隔离级别的支持

- 更新数据的冲突还是存在的; 更新数据的锁使用与其它隔离级别是一样的
  - 隔离级别是可以混用的 (但不推荐)
  - 数据库属性 VersionStorePages 包含在临时文件中用于保存旧数据的页面数量
    - SELECT DB_PROPERTY ('VersionStorePages');
    - BLOB 不存在于临时文件中, 而是存在于主数据库文件中
  - 当存在快照事务时，一些DDL语句会有限制 (如，ALTER TABLE)

隔离级别: 建议

- **设置隔离级别让应用系统在数据访问连续性与并发性中达到最佳平衡**
  - 在隔离级别 0是没有任何保证的 (“脏读”)
- **对于隔离级别 3, 确保服务器能访问索引从而减少锁的数量**
  - 对于隔离级别3，服务器将尽可能地避免顺序扫描数据
- **当必须在事务中使用多个快照隔离级别时，**
  - 指定游标级别的隔离级别，而不是通过数据库的option设置

Demo

执行计划响应时间: 游标

- **ESQL 游标类型:**
  - no scroll, dynamic scroll (default), scroll, sensitive, insensitive
- **ODBC游标类型**
  - static, dynamic, keyset, mixed, forward-only (default)

网络的响应时间及性能

- 响应时间: 数据包发出，被另外一个计算机接收到所用的时间
- 吞吐量: 在给定的一段时间内传送的数据量（ bits (bytes)）
- 局域网（LAN）: 典型的为1ms (或许更少) 的响应时间, 最少 1MB/sec 的吞吐量
- 广域网（WAN）: 5-500 ms 的响应时间, 4-200KB/sec 的吞吐量
  - 这是一个大体估计

减少网络响应时间

- **增加数据库服务器网络数据包的大小**
  - 在 V11上缺省值已经从 1460 提高到了 7300; 更大的网络包大小对于大的结果集是有益的
  - 能提高大的 FETCHes 和多行/列取数据的性能，, 或BLOB类型数据的性能 (对于检索与插入均有提升)
- **使用数据库连接参数 CommBufferSize**
  - 只对于需要设置该参数的连接设置
  - 如： "D:\Program Files\SQL Anywhere 11\bin32\dbisql" -c "eng=remotedb;uid=dba;pwd=sql;CommBufferSize=8000"

减少网络响应时间

- **考虑修改TCP/IP参数 ReceiveBufferSize 和 SendBufferSize**
  - 提前为TCP/IP网络协议分配内存用于接收/发送数据包
  - 缺省值是跟操作系统有关的 (操作系统, 驱动, 制造商等)
  - 设置为 64K ~ 256K 一般是比较合适的
  - "D:\Program Files\SQL Anywhere 11\bin32\dbsrv11" -x "tcpip(ReceiveBufferSize=128k; SendBufferSize=128k)" remotedb.db
  - "D:\Program Files\SQL Anywhere 11\bin32\dbisql" -c "eng=remotedb;uid=dba;pwd=sql;CommBufferSize=8000;CommLinks='tcpip(ReceiveBufferSize=128k; SendBufferSize=128k)'"

提高网络吞吐量

- **客户端与服务器端通过modem或广域网联接进，传输压缩或许能提高吞吐量**
  - 通过设置客户端字符串中的连接参数 Compress=YES, 或数据库服务器命令行选项 –pc （压缩除同机连接之外的所有连接 ）
  - 数据包在加密前进行压缩
  - 压缩后的数据可以是原数据大小的10%，甚至更小, 这完全依赖于具体数据及具体应用系统
  - 考虑增大数据包大小，以得到更大的压缩比及更少的数据包
  - 压缩对每一个连接需要额外的 ~46K 内存
  - 必须分析应用系统的性能并验证压缩的效果
  - **压缩需要额外的 CPU; 在局域网中, 压缩的开销可能会超过因此而减小的带宽的开销**

减少网络响应时间

- 减小请求的数量从而使用客户端与服务器之间的通讯更高效
  - 在应用中每次取较多的数据；每次提交更多的数据
  - 对于较大的结果集使用预取（ PREFETCHing）
  - 通过应用系统缓存数据，而不是每次重新到服务器上去取数据
  - 将多个SQL组织起来，或使用存储过程，一块提交给服务器，而不是分多次向服务器发起请求
  - 尽可能的将数据与列绑定（将结果集列绑定到应用程序数据缓冲区 ）
    - 使用 SQLBindCol() ，而不是使用 SQLGetData()
  - 对于查询操作不要使用 COMMITing
    - 对于 JDBC 和 ODBC，这是缺省值
    - 如果没有事务，COMMIT会触发CHECKPOINT

减少网络响应时间: 大的 fetches/inserts

- **对于相对较大的结果集，使用大的 fetches**
- **每个请求获得几条记录; 通过应用系统明确指定**
  - Prefetching 可能会发生，也可能不会发生
- **获取大数据的数量可以通过接口指定，包括： ODBC 和 JDBC**
- **大的 (多行) inserts**
  - ESQL, ODBC, JDBC均支持
  - 考虑使用 LOAD TABLE 为表加载数据
  - 一定时间间隔执行 COMMIT ，从而减少锁争用, 限制回退事务日志的大小

提高应用系统的效率

- 记住当应用系统退出时通过SQLFreeStmt()删除语句定义释放占用的空间
- 当游标需要只读时，声明为只读
  - 对于可更新游标，一些语义上的最优化会禁用,例如消除关联, 可以大大简化原始请求
  - 对于某些接口可以设置结果集的 adaptive PREFETCHing ， 即使已经声明为 FORWARD ONLY (例如 ODBC)

提高应用系统的效率

- 仔细检查应用系统中的嵌套查询
- 可供选择: 将嵌套查询修改为 LEFT OUTER JOIN查询
  - 准确的用法依赖于应用系统的实际情况

提高应用系统的效率

- **通过 OPEN … WITH HOLD 只有具有合适的WHERE条件时**
  - 所有的锁及时通过commit释放 (当前行除外); 不保证其它行的状态
  - 但当ROLLBACK时， 语义上不明确
    - 当ROLLBACK 时，游标的内容是不明确的
    - 考虑设置选项 ANSI_CLOSE_CURSORS_ON_ROLLBACK 在ROLLBACK时强制关闭所有游标
      - set option ANSI_CLOSE_CURSORS_ON_ROLLBACK='on'
      - 或 set option close_on_endtrans = 'on‘
        - (close_on_endtrans 选项会替代 ansi_close_cursors_on_rollback 选项。)

通过物化视图提高查询效率

- **物化视图的功能**
  - 在数据库中象表一样存储计算结果数据
  - 只读,不允许进行数据修改操作
  - 可以对其中的列建立索引
  - 可以设置查询计划最优化
    - **ALTER MATERIALIZED VIEW 物化视图名称 ENABLE USE IN OPTIMIZATION;**

物化视图

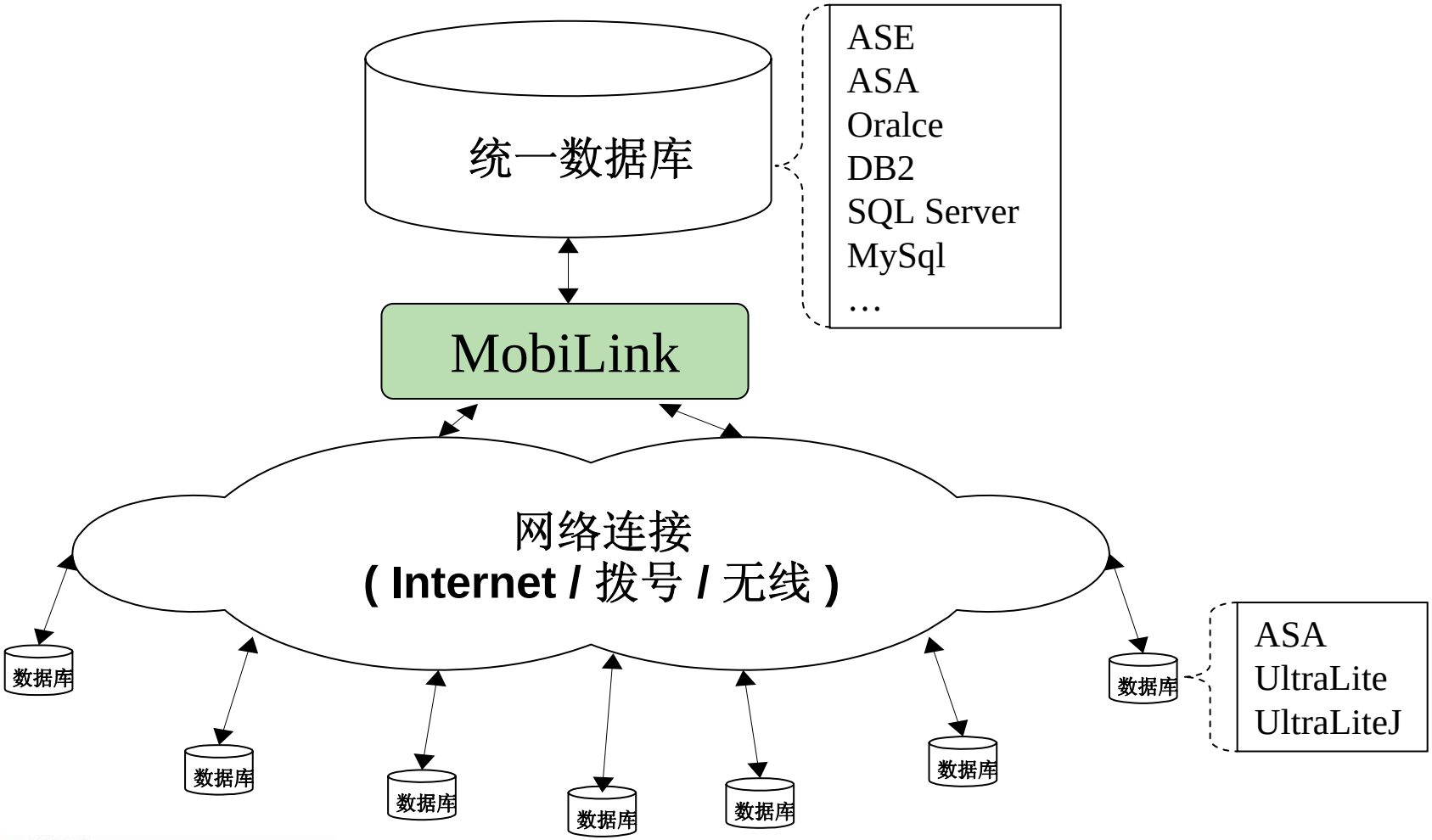
- 使用物化视图的好处
  - 提高查询操作的性能
  - 提高了并发性

Demo

议题

- SQL Anywhere 概述
- 总体设计原则
  - 开发计划设计技巧
  - 应用开发技巧
- 一些技术细节
  - 事物隔离级别
  - SQL Anywhere支持的游标(cursor)
  - 物化视图
- MobiLink性能优化

Mobilink架构


统一数据库
ASE
ASA
Oralce
DB2
SQL Server
MySql
…
MobiLink
网络连接
(Internet / 拨号 / 无线)
数据库
数据库
数据库
数据库
数据库
数据库
数据库
ASA
UltraLite
UltraLiteJ
终端(PC/手机/PDA)

1.为高性能而设计

- 性能是客户可直接感受到的
- MobiLink环境的吞吐量越大, 越好
  - 检测阻碍性能的瓶颈
- 只同步需要的数据
  - 较少的同步
  - 同步较少的数据
    - 通过减少传输的数据加快同步速度

2. 理解现实的数据特征

- **数据和数据库事务中的性能测试需要与生产环境一样**
  - 了解生产数据的特点和规模
- **创建客户端的典型操作**
  - E.g. 质量保证自动化,测试1000用户与30个实际用户的情况
- **如果不是真实的数据,那么数据库缓存、锁争用、系统争用情况 就不会被真正的测试出来**
  - 意外的情况就会出现在生产环境中

3. 模拟生产环境

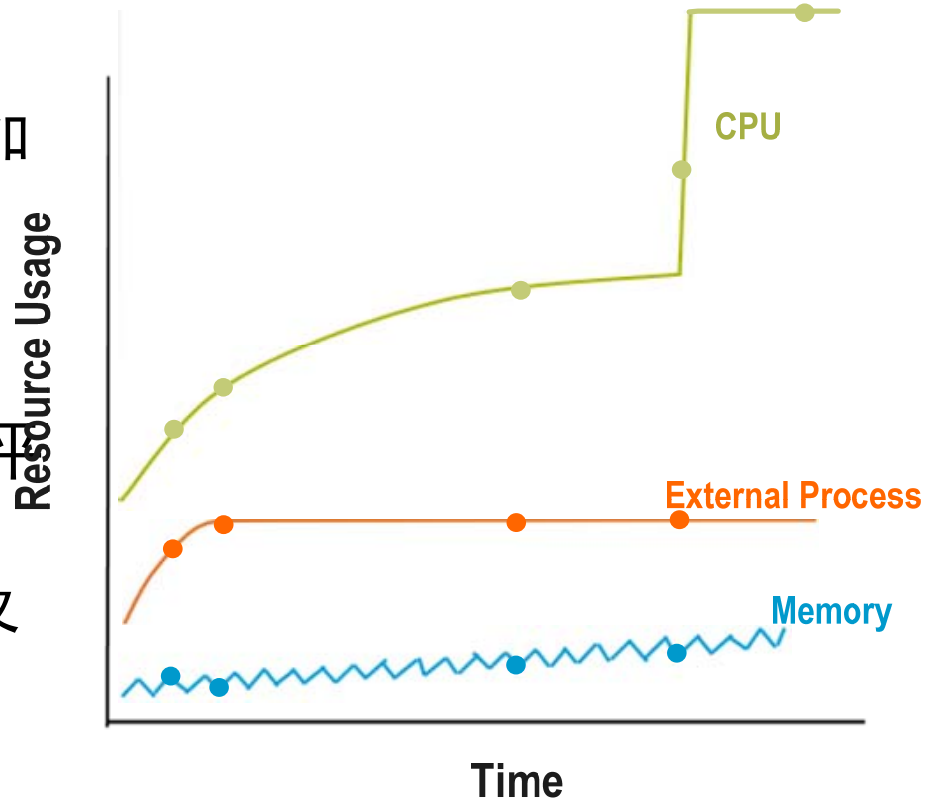
- 正确的模拟要测试的环境
  - 服务器端的工作负荷
  - 活动的客户端数
  - 客户端的工作负荷
  - 硬件
  - 网络
  - 拓扑
  - 安全
  - 时序

3. 模拟生产环境目标

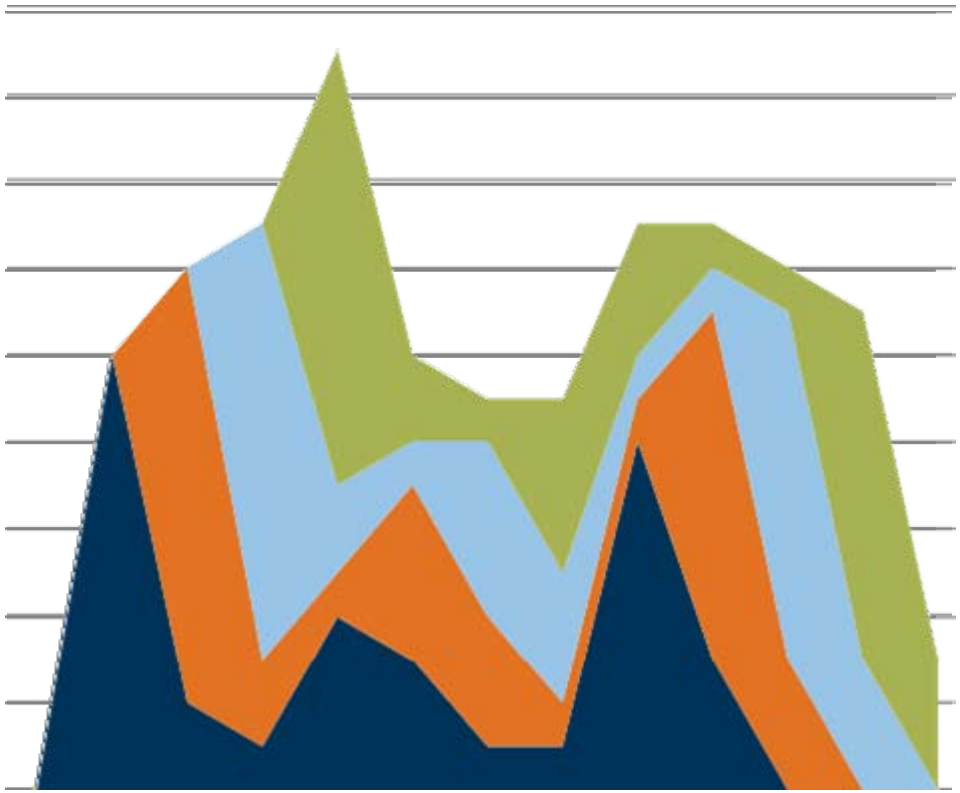
- **最大负荷测试**
  - 测试以证明你的系统将处理峰值负荷
- **持久性测试**
  - 持续测试数天以发现发现其它的问题
  - 应当测试所有有关应用系统的操作
    - 初如化同步
    - 增量同步
    - 晚上的批处理操作
    - 报表
    - 所有有关应用系统的其它操作
      - 应用系统的综合访问
      - 中介服务器
      - 企业认证和验证
      - 执行 JAVA 和 .NET 脚本

3. 模拟生产环境目标

- 监控整个系统
  - –内存随着时间的推移缓慢增加
    - •建议查看是否存在内存泄漏
  - –外部进程线先是上升后变为水平
    - •建议机器上运行适当的进程
  - –CPU的增幅达100％然后呈水平
    - •应该为螺旋形上升
    - •应当为：CPU使用达到峰值后又恢复到正常



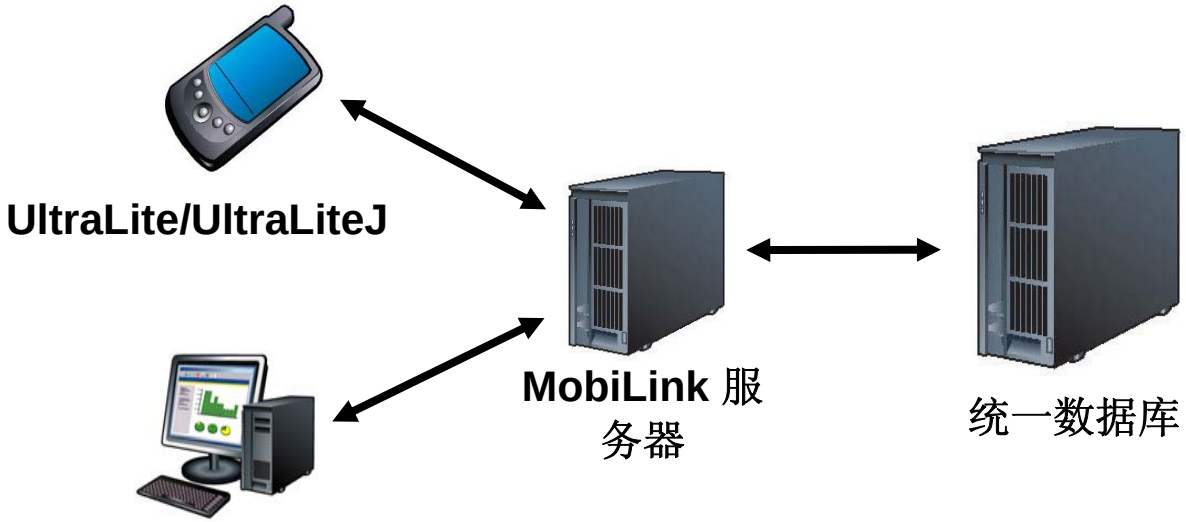
3. 模拟生产环境目标



- **模拟各种典型条件**
  - 同时启动模拟
  - 模拟交错启动

4.尽早在开发周期开始测试

- 尽早进行性能测试以验证架构的可扩展性
  - 避免浪费时间在项目的后期重新设计架构
  - 系统中的各个组成部分都可能是一个潜在的瓶颈



5. 发现瓶颈

- 需要知识，经验和创造力
- 通过测试可以发现瓶颈
  - 网络
  - 系统的相互作用
  - 数据库争用
- 发现工作需要根据经验
  - 测试, 改进, 再修改
    - 每次只更改一个参数或变量
- 数据库管理员必须能够识别和修复影响性能的参数或条件

6. 使用扩展性测试工具

- 好的工具将帮助管理测试过程
  - 分发文件
  - 启动客户端进程
  - 启动所有必需的组件和测试流程
  - 捕获数据(服务器端 vs. 客户端的时序)
  - 提供结果的摘要
    - 帮助在系统测试中发现瓶颈

7. 随着发展变化做测试

- 改变环境变量或参数会改变测试的结果
- 许多条件可以改变环境
  - 客户端的使用
  - 用户数
  - 数据量的变化
  - 软件变更
  - 硬件更改
  - 业务逻辑和要求
- 条件可以改变，直至产品的寿命结束


iAnywhere®
A SYBASE COMPANY