数据库复习

Cyanny

本次考试满分是 120 分，其中 20 分是附加题，并不要求完成。
复习要点：
1、在 PPT 上没有某些考试题目的答案；
2、注意复习数据库中讲授过的基本文件的结构和读取方式；
3、注意复习数据库中讲授过的基本索引的结构和读取方式；
4、注意复习较为复杂的 ORM（对象关系映射）或 ER-表结构的映射规则

1. 为性能而设计

提纲：一定会考，数据库之间的差别很大，差异性大，要知道开发成功数据库应用的要点
提高性能的方法，优缺点，异步方法：批处理，峰值处理，实时性不好；同步方法

1.1. 开发成数据库要点

- 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”，应用是使用数据库的应用，数据最终会被多个应用使用，要充分利用数据库
- 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

1.2. 现状

应用系统的首要目标是满足业务需求，但实际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。
数据库开发为性能而设计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

1.3. 一些概念（看看就行）

1. **关系代数**

代数表达式——**可以变换表达式，求出同样的值，即等价变换**
关系数据库基础是关系代数，关系数据库操作都是关系代数的操作符
任何一个 SQL 都可以转化为不同形式的表达式，但是有相同结果，每一个表达式，都是一个路径，**查询优化器**选择最高效，省时间的路径

2. **数据库基本特性**

表、key、完整性约束（组件约束和引用约束）、锁、视图，事务

3. **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**
  二维表就是关系，关系操作是表和表之间的操作。关系不包含重复的数据，但是 DBMS 中并未这样实现；关系的记录之间没有顺序，如果数据之间有次序要求，则不能被称为一张关系表，而记录的顺序往往会影响查询的效率
- **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：系不包含重复数据；记录之间没有顺序
- **视图**：是一个 SQL 语句，当使用到 SQL 语句进行查询的时候才生成这个视图，在用的时候才会被执行。视图用在：
  - **权限**：当希望基本表不暴露给别人，所以使用视图来暴露给别人，此时视图可能引用到多张表而形成一个可以被其他应用所看到和使用的视图，从而保护基本表不被别人看到。与接口类似
  - **重构**：用在数据库重构中。完成了数据库重构中，而将原来的表做成完全一样的视图，此时内部的所有重构都不会影响到外部的使用（外部应用使用视图）。如果视图引用到大于 4 张表，此时应当考虑到性能的问题，但是应用不会看到视图的引用情况，所以使用视图可能会掩盖一些现象

4. 数据库范式

范式是为了让数据库减少冗余

- 1NF 确保原子性（Atomicity），原子性的粒度、原子性的价值，每一列只有单值，单个字段当做单个值来使用，不做拆分来验证和使用
- 2NF 检查对键的完全依赖，价值在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
- 满足范式是为了降低冗余，逆范式——保持一定的冗余，提高效率

5. SQL 复杂度

SQL 的复杂度取决于 from 后面包含的表数量（笛卡尔积的基数）。没有任何厂商支持完整的 SQL99 标准，而是只支持最基本的部分（query table/ query view/ select/ insert/ update/ delete）。每一个数据库的存储过程和触发器都不一样

6. 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

7. 约束应该明确说明

- 数据库中存在隐含约束是不良设计
- 字段的性质随着环境变化而变化时，会出现错误和不稳定性
- 数据语义属于 DBMS，别放到应用程序中
- 约束可以放在程序中检查，也可以放在数据库中，如果数据库是中心级数据库，则放在数据中检查，便于多个应用共享

8. **过于灵活的危险性**

- 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

9. **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

10. **关于 NoSQL**

在基于 web 的应用中大量地使用非关系型数据库，**放宽对一致性的要求而提高吞吐量**。因为系统有并发的限制，所以单一请求的性能作为系统的性能指标是不可行的，因此引入吞吐量的概念。**致性的要求会产生大量的回滚操作，从而降低吞吐量。NoSQL 认为脏写、不一致性都是可以接受的，只要在一段时间内保证数据库的一致性即可（关系型数据库要求实时保证数据库的一致性）**。Google、新浪微博等都是使用 NoSQL 实现的

1.4. 数据库之间的差异很大

数据库之间的比较，比如 Oracle/MySQL/SQL Server/DB2，数据库的连接、组织形式是完全不同的。要提高数据库的性能最本质的是在某一个数据库上提高性能

1.5. 设计与性能（提高性能的方法）

性能拙劣源于错误的设计，调优就是在目前情况下，优化性能至最佳

1) **数据结构的设计。**

对数据表进行规范化，可以保持适当的冗余量，建立有用的索引，尽量减少表与表之间的关联，运用存贮过程等。

2) 处理流程

异步处理模式：可以允许用户发出操作之后去做其他工作，数据库进行集中的**批处理**，当处理完毕后通知用户，但是实时性不好

同步处理时模式：用户必须等待数据库操作并返回结果，期间不能做其它操作。适合数据库进行实时交易。

要根据系统的需求选择一种合适的手段，这会影响物理结构的设计。

3) 数据集中化

分布式数据存储：具有更灵活的体系结构，减少了单一节点的负担，但是有缺点：远程数据的透明引用访问代价很高；不同的数据源数据结合极为困难。存取结构复杂，保密性不易控制；多个副本的一致性难以保证

数据集中化：提高数据库速度，并且在部署上服务离数据越近，速度会越快。Block 使用充分，命中率高，所需 io 操作少；缺点是：单点故障，会带来很大损失，解决方式是热备份或灾难备份

4) 接近 DBMS 核心

代码里离 DBMS 核心越近运行越快,充分利用每次数据库访问。

5) 保持数据库连接的稳定。链接稳定减少交互。

6) 充分利用每次的数据库访问，减少服务与数据库之间的交互。

不要多次访问数据库，提取多段信息，要在合理范围内，利用每次数据库的访问，完成尽量多的工作

7) 优化 SQL 语句

把逻辑放入到查询 SQL 中而不是 SQL 宿主语言中；优化 SQL 语句的结构。

8) **谨慎地使用自定义函数，不当的位置会使自定义函数执行次数过多而造成性能下降；**

Select 中调用一次，where 中可能每一行检查都会调用，如果自定义函数内部还执行了一个 select 语句，则无法被查询优化器优化

9) 构建稳定的应用需要防御式编程，但在合理的情况下可以使用进攻式编程。

- **防御式编程：**在开始处理前检查所有参数的合法性，可移植性高

如通过 left join 返回的各种情况做判断（检查那个字段错误）

限用 boolean 型字段

约束应明确说明：数据中存在隐含约束是一种不良设计

- **进攻式编程**

  不做合法性检查，以合理的可能性为基础，绝大多数情况下不出错，进攻式可以提高吞吐量，要对异常谨慎控制

2. SQL 优化

提纲：where 优化：distinct；买 BMW 例子；摆脱 distinct 方法，用 exist 和 in，告诉查询优化器，先优化哪些路径

2.1. SQL 和优化器概念

- 优化器：借助关系理论（关系代数）提供的语义无误的原始查询进行有效的等价变换，寻找最优路径，产生新能最优的执行方案
- 优化：在数据处理的真正被执行的时候发生
- 影响优化的因素：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
- Sql 语句先执行关系操作，在执行非关系操作（order by）
- **逻辑查询处理阶段简介（网上找的，理解一下）**

FROM：对 FROM 子句中的前两个表执行笛卡尔积（Cartesian product)(交叉联接），生成虚拟表 VT1

ON：对 VT1 应用 ON 筛选器。只有那些使<join_condition>为真的行才被插入 VT2。

OUTER(JOIN)：如 果指定了 OUTER JOIN（相对于 CROSS JOIN 或(INNER JOIN),保留表（preserved table：左外部联接把左表标记为保留表，右外部联接把右表标记为保留表，完全外部联接把两个表都标记为保留表）中未找到匹配的行将作为外部行添加到 VT2,生成 VT3.如果 FROM 子句包含两个以上的表，则对上一个联接生成的结果表和下一个表重复执行步骤 1 到步骤 3，直到处理完所有的表为止。

WHERE:对 VT3 应用 WHERE 筛选器。只有使<where_condition>为 true 的行才被插入 VT4.

GROUP BY：按 GROUP BY 子句中的列列表对 VT4 中的行分组，生成 VT5.

CUBE|ROLLUP：把超组(Suppergroups)插入 VT5,生成 VT6.

HAVING：对 VT6 应用 HAVING 筛选器。只有使<having_condition>为 true 的组才会被插入 VT7.

SELECT：处理 SELECT 列表，产生 VT8.

DISTINCT：将重复的行从 VT8 中移除，产生 VT9.

ORDER BY：将 VT9 中的行按 ORDER BY 子句中的列列表排序，生成游标（VC10).

TOP：从 VC10 的开始处选择指定数量或比例的行，生成表 VT11,并返回调用者。

2.2.优化策略

2.2.1 加倍留意非关系操作

一旦关系操作完成就再也回不去了，优化：

- ✓ 把查询结果传给外部查询的关系操作
- ✓ 无论优化器多么聪明，都不会合并两个查询，而只是顺序执行
- ✓ 只要不是纯关系操作层，查询语句的编写性能的影响重大，因为 sql 引擎将严格执行它规定的执行路径
- ✓ **最稳妥的方式:**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

例如：oracle 里的 rownum

不应该，order by 在 where 之后才做，所以还没有存在，查询失败

```
select empname, salary from employees where status != 'EXECUTIVE' and rownum<= 5
order by salary desc
```

而应该

```
select *  from (select empname, salary from employees  where status != 'EXECUTIVE'
order by salary desc)  where rownum<= 5
```

2.2.2 优化器的有效范围

- ✓ 优化器需要借助数据库中找到的信息
- ✓ 能够进行数学意义上的等价变换
- ✓ 优化器考虑整体响应时间
- ✓ 优化器改善的是独立的查询语句
- ✓ **策略是：**如果是若干个小查询，优化器会个个优化；如果是一个大查询，优化器会将它作为一个整体优化

2.2.3 使用 SQL 语句要考虑的因素

1) **数据总量**

Sql 考虑最重要的因素：必须访问的数据总量；没有确定目标容量之前，很难判断查询执行的效率

2) **定义结果集的查询条件**

好的查询条件：满足此条件的数据很少，可以过滤很多数据

Where 字句：特别在子查询或视图中可能有多个 where 字句

过滤的效率有高有低，受到其他因素的影响

影响因素：过滤条件，主要的 sql 语句，庞大的数据对查询影响

3) **结果集的大小**

- 查询所返回的数据量，重要而被忽略
  - 取决于表的大小和过滤条件的细节
  - 例外是若干个独立使用效率不高的条件结合起来效率非常高
- 从技术角度来看，查询结果集的大小并不重要，重要的是用户的感觉
- 熟练的开发者应该努力使响应时间与返回的记录数成比例

4) **获得结果集所涉及的表的数量**

表的数量会影响性能

连接：太多的表连接（八张）就该质疑设计的正确性了；对于优化器，随着表数量增加，复杂度指数增长；编写太多表的复杂查询时，多种方式连接的选择失误几率很高

视图：会掩盖多表连接的事实

减少复杂查询和复杂视图

5) **并发的用户数（同时修改数据的用户数）**

- ✓ 设计时要注意：数据块访问争用，阻塞，闩定，保证读取的一致性
- ✓ 一般而言，整体的吞吐量>个体响应时间
- ✓ 数据存贮采用固定大小的区块，可以存取多条记录，I/O 交互简单，在内存与缓冲中好处理；但是当修改后的数据太长，则会进行迁移到另一个 block 存储；数据块的太大，会带来数据块的访问争用的问题，影响并发性能

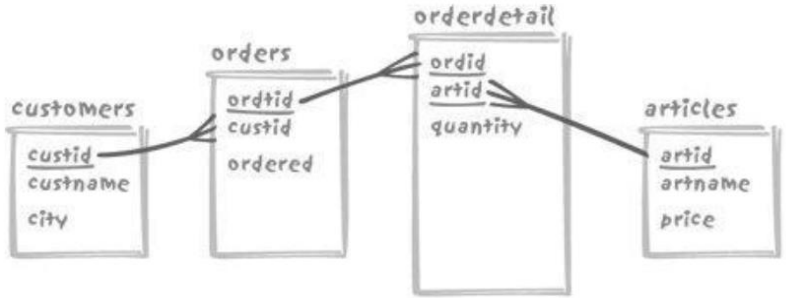
2.2.4 过滤

1) 如何限定结果集时最为关键的因素，是使用 SQL 各种技巧的判定因素
2) 过滤条件的含义：
   Where 字句和 having 字句
   Join 过滤条件
   Select 过滤条件
3) **过滤条件的好坏，取决于**
   最终需要的数据是什么，来自哪些表
   哪些输入值会传递到 DBMS 引擎
   能过滤掉不想要的数据的条件有哪些
   高效过滤条件是查询的主要驱动力

2.2.5 摆脱 distinct，使用 exists 和 in 操作

1) 避免在最高层使用 distinct（会产生错误结果集），因为发现重复的数据容易发现不准确的连接难，发现结果的不正确就更难了
2) **摆脱 distinct 的方法：使用 exists 和 in，exits 需要用到嵌套子查询；**
   1. **exists 查询**
      - ✓ 需要使用**关联嵌套**，子查询中要使用外层查询的表中的字段信息
      - ✓ 内部查询中的 join 关系与外部表**没有直接的叉乘关系**，而是带入外部查询的结果值到内部查询中，减少一次叉乘次数，但是内部查询要带入多次外部表的值
      - ✓ exits **被优化一次后多次执行，查询不需要完整执行产生结果集**，只要满足条件，判定行集非空，就为 true；可能找到第一条满足条件的数据，就执行完毕返回；子查询时间难以估算。经过查询优化器优化后，优化的时间远远超过执行的时间，虽然执行多次，效率反而更高
      - ✓ exits 暗示查询优化器，这是一个**内部查询优化**，优化器采用随机方式，独立优化
      - ✓ 外部查询所用到的字段条件所对应的内层查询字段一定要在表中加上**索引**，提高内部查询的效率（BMW 中 orders 表要在 custid 字段添加索引）
      - ✓ **避免了 distinct 所带来的结果集的错误**
      - ✓ 适用于：**外部条件在结果集中所占比率较小的情况**，可以减少子查询的次数（BMW 中属于最近 6 月买 BMW 较少的情况）
   2. **in 查询**
      - ✓ 可以使用关联子查询和非关联子查询
      - ✓ 非关联子查询中，内层查询不在依赖外层查询，优化一次后只需要执行一次，但必须完整执行，再和外部查询比较取出交集
      - ✓ 非关联子查询不需要在于外层字段有关的字段上添加索引，因为二者没有关系
      - ✓ **适用于：**外部查询条件所占比例比较大的情况，in 的效率会远远高于 exists 效率

   例子：



不对的方式：

找出最近6个月住在nanjing，购买了BMW的所有客户

```
select distinct  c.custname
    from customers c
        join orders o
          on o.custid = c.custid
        join orderdetail od
          on od.ordid = o.ordid
        join articles a
          on a.artid = od.artid
    where c.city = 'Nanjing'
      and a.artname = 'BMW'
      and o.ordered >= somefunc /*函数，返回六个月前的具体日期*/
```

改进：

摆脱distinct的方法

```
select c.custname
   from customers c
   where c.city = 'Nanjing'
     and exists (select null
                   from orders o,
                        orderdetail od,
                        articles a
                  where a.artname = 'BMW'
                    and a.artid = od.artid
                    and od.ordid = o.ordid
                    and o.custid = c.custid
                    and o.ordered >= somefunc )
```

客户在Nanjing市，而且满足Exists存在性测试即在最近六个月买了BMW

Exists嵌套子查询和外层select关系非常密切

非关联子查询

```
select custname
from customers
where city = 'Nanjing'
    and custid in (select o.custid
                   from orders o,
                        orderdetail od,
                        articles a
                  where a.artname = 'BMW'
                    and a.artid = od.artid
                    and od.ordid = o.ordid
                    and o.ordered >= somefunc)
```

关联子查询中，orders表中custid字段要有索引，而对非关联子查询则不需要，因为要用到的索引是customers的主键索引

内层查询不再依赖外层查询，只需要执行一次

其他方法：from 字句中的内嵌视图

3) **Solution：**

找到分辨率最强的条件，即满足条件的结果集小，选择使用 exits 还是 in

解决方案不止一种，查询和数据隐含的假设密切相关

预先考虑查询优化器的工作，以确定它能找到所需要的数据

2.2.6 大数据量查询原则

原则：越快踢出不要的数据，查询的后续阶段必须处理的数据量就缺少，查询效率越高

应用：

- ✓ 集合操作，如 union 语句，但是不要 cut-and-paste
- ✓ Group by&having 字句
  - ■ 所有影响聚合函数的结果条件都应该放在 hanving 字句中
  - ■ 任何无关聚合条件都应该放在 where 子句中
  - ■ 减少 group by 必须执行排序操作所处理的数据量

2.2.7 非关联子查询变成内嵌视图—降低查询维度

例子：

再回头看订单和客户的例子

- 在订单完成前有不同状态，记录在orderstatus（ordid,status,statusdate）中
- 需求是：列出所有尚未标记为完成状态的订单的下列字段：订单号，客户名，订单的最后状态，以及设置状态的时间

不好的方式，5 张表连接查询，外面三张+内部两张：

```
select c.custname, o.ordid, os.status, os.statusdate
    from customers c,
         orders o,
         orderstatus os
   where o.ordid = os.ordid
     and not exists (select null
                     from orderstatus os2
                     where os2.status = 'COMPLETE'
                        and os2.ordid = o.ordid)
     and os.statusdate = (select max(statusdate)
                          from orderstatus os3
                          where os3.ordid = o.ordid)
     and o.custid = c.custid
```

改进

非关联子查询变成内嵌视图

```
select c.custname, o.ordid, os.status, os.statusdate
  from customers c,
       orders o,
       orderstatus os,
       (select ordid, max(statusdate) laststatusdate
        from orderstatus
        group by ordid) x
 where o.ordid = os.ordid
   and os.statusdate = x.laststatusdate
   and os.ordid = x.ordid
   and os.status != 'COMPLETE'
   and o.custid = c.custid
```

2.3. SQL 语句方面注意

1. 注意在做否定意义的查询要小心进入陷阱：

如，没有选修'B2'课程的学生 ：

```
select students.*
from students, grades
```

where students.sno=grades.sno
AND grades.cno <> 'B2'（一定选课，但是没有选课的人）
上面的查询方式是错误的，正确方式见下方：
select * from students（选出了没有选课的学生+选课的学生且没有选课 B2）
where not exists (select * from grades
where grades.sno=students.sno AND cno='B2')

(2)关于 **JOIN**

JOIN：如果表中有至少一个匹配，则返回行（INNER JOIN）
LEFT JOIN：即使右表中没有匹配，也从左表返回所有的行
RIGHT JOIN：即使左表中没有匹配，也从右表返回所有的行
FULL JOIN：只要其中一个表中存在匹配，就返回行

2.4. SQL 查询优化总结

1. 暗示查询优化器如何优化
   使用 join 来暗示表连接顺序，当有多表连接操作时，考虑使用 exists 和 in 操作来优化；如果不使用 join 则是让查询优化器自己优化，自己确定表连接顺序（先小表，再大表），效率可能较低
2. 将多维度的查询进行降维处理，一次连接的表不要超过 3 张，超过了就将非关联子查询变成内嵌视图，降维处理
3. 考虑取出的数据在表中的比例，当查询返回记录超过数据总量 10%就不使用索引，查询结果集少于 10%是好的查询条件
4. 避免在高层使用 distinct，使用 exists 和 in 来处理
5. 避免在高层使用 select *，这样会产生冗余的结果集，降低性能

3. 索引和数据表的物理实现

提纲：B 树索引结构；为什么没有使用我的索引？为什么使用索引？快速访问数据，
索引会带来存储和处理的负担，没有使用浪费存储空间，插入效率慢
请列举没有使用索引的例子？要举例
注意基本的索引结构和存取方式

3.1 索引概念

1. 索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。是一种以原子粒度访问数据的手段，而不是为了大量数据的访问。是一种数据访问方式；索引是顺序存取

2. 索引分类：

聚簇索引：按照数据存放的物理位置为顺序的，索引的叶节点就是物理上的叶节点，聚簇索引能提高多行检索的速度

非聚簇索引；索引顺序与数据物理排列顺序无关，叶节点仍然是索引节点，保留一个指针指向数据块，非聚簇索引对于单行的检索很快。

一个表最多只能有一个聚簇索引

3. 索引结构：B 树

4. 索引目的：提高查询效率

5. 索引使用时的考虑

- ✓ 检索比率，一般适用于满足条件的数据量少的情况
- ✓ 磁盘访问，内存访问，记录存储

6. 索引与外键

a) 如果没有外键和引用的话，一次修改会导致多次修改

b) 大系统普遍取消外键的关联，取消参照完整性（降低在更新主表时候的过多引用）是提高数据库性能的一个措施。如果有大量的外键关联，则做一次主表查询可能会导致连接多个代码表

c) 如果有外键的话，则需要对外键加上索引，但是不一定，如果该外键不经常使用就不用添加索引，索引建立必须要有理由，无论是外键还是其他字段，并不是外键都要添加索引

d) 如果系统为外键自动添加索引，常常会导致同一字段属于多个索引，为每个外键建立索引，会导致多余索引

7. 系统生成键

1) 系统生成键远好于寻找当前最大值并加 1；好于用一个专用表保存下一个值“且枷锁更新”
2) 系统生成键是串行插入
3) 如果插入并发性过高，在主键索引的创建操作上会发生严重的资源竞争
4) 解决方案：反向键索引（逆向索引）；哈希索引
5) 系统生成键使用数字比使用字符串效率高
6) 不使用系统生成键，可能会导致插入时主键取值不唯一，有利于主键的唯一性

3.2 索引的优点，为什么使用索引？

1. 什么时候使用 B 树索引：

仅当要通过索引访问表中很少一部分行

如果要处理表中多行，而且可以使用索引而不用表

2. 索引的 5 种优点

- ✓ 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
- ✓ 可以大大加快数据的检索速度，这也是创建索引的最主要的原因。
- ✓ 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
- ✓ 在使用分组和排序子句进行数据检索时，同样可以显着减少查询中分组和排序的时间。
- ✓ 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

3. 应该建立索引的条件

1) 在**经常需要搜索的列**上，可以加快搜索的速度；
2) 在作为**主键**的列上，强制该列的唯一性和组织表中数据的排列结构；
3) 在**经常用在连接的列**上，这些列主要是一些外键，可以加快连接的速度；

4) 在**经常需要根据范围进行搜索的列上创建索引**，因为索引已经排序，其指定的范围是连续的；外键建索引由于连接加快还会减少死锁几率。
5) 在**经常需要排序的列**上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
6) 在**经常使用在 WHERE 子句中的列**上面创建索引，加快条件的判断速度。

3.3 索引的局限性

1. 为什么不为每一列建立索引

1) 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
2) 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
3) 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

2. 索引会带来的问题

1) 索引有可能降低查询性能，带来磁盘空间的开销和处理开销等
2) 太多的索引，让设计不稳定
3) 对于大量数据检索，索引效率反而更低
4) 创建索引回来带系统的维护和空间的开销
5) 数据修改需求大于检索需求时，索引会降低性能

3. 这些列不应该建立索引

1) 对于那些在查询中**很少使用或者参考的列不应该创建索引**。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
2) 对于那些**只有很少数据值的列也不应该增加索引**。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，**结果集的数据行占了表中数据行的很大比例**，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
3) 对于那些定义为 **text, image 和 bit 数据类型的列不应该增加索引**。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。
4) **当修改性能远远大于检索性能时，不应该创建索引。**这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

4. 为什么没有使用我的索引？(ppt)（不使用索引的情况）

主要是因为：使用索引反而得不到正确结果；或使查询效率变得更慢

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

表 T，T(X,Y)上有索引，做 SELECT * FROM T WHERE Y=5

跳跃式索引（仅 CBO）

2) **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，不带任何条件的 count 会引起全表扫描。

3) **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

4) 情况 4：隐形函数查询（主要是时间和类型变化这种隐形函数查询）

不等于符"<>"会限制索引，引起全表扫描，如果改成 or 就可以使用索引了。

is null 查询条件也会屏蔽索引。

5) 情况 5：此时如果用了索引，实际反而会更慢

数据量本来不够大，oracle 自己计算后认为不用索引更合算，则 CBO 不会选择用索引

6) **情况 6**：没有正确的统计信息，造成 CBO 无法做出正确的选择；

如果查询优化器认为所有会使查询变慢，则不会使用索引

表分析就是收集表和索引的信息，生成的统计信息会存在 user_tables 这个视图。CBO 根据这些信息决定 SQL 最佳的执行路径。

其他：

1. 对于两个公有字段的表，如果在做外表的表上对该字段建立索引，则该索引不会被使用因为外表的数据访问方式是全表扫描。

2. 查询使用了两个条件用 or 连接，**如果条件 1 中的字段有索引而条件 2 中字段没有，则仍会全表扫描。**

4. 物理组织形式的优化？

提纲： IOT，堆文件，IOT 和堆文件的差别；分区的特性

4.1 堆文件 vs 索引组织表（IOT）

资料来自 http://hi.baidu.com/bystander1983/blog/item/adbebd11fabd7ef5c3ce7994.html

4.1.1 堆文件

1. **文件结构**：堆文件就是一般的数据表，使用"heap"的结构，数据没有特定的顺序；表是无组织的，只要有空间，数据可以被放在任何地方。堆文件的表和表主键上的索引要分别留出空间
2. **读取/访问方式**：获取表中的数据是按命中率来得到的。没有明确的先后之分，在进行全表扫描时，并不是先插入的数据就先获取。数据的存放是随机的，也可以根据可用空闲的空间来决定。

4.1.2 索引组织表 IOT

1. **数据读写的冲突问题（理解就好，来自 PPT）**

并发用户数很大的系统：紧凑的方式存储数据 vs 分散的方式存储

没有并发的修改密集型：数据查询要快 vs 数据更新要快

DBMS 所处的基本单元（页、块）通常不能和谐

2. **IOT**（(index organized table）索引组织表，oracle 提供数据存储存储方式

当索引中增加额外的字段（一个或多个，它们本身与实际搜索条件无关，但包含查询所需的数据），能提高某个频繁运行的查询的速度。

IOT 存储在索引结构中的表，所有字段纳入索引，不存在主键的空间开销，允许在主键索引中存储所有数据，这个表本身就是索引

3. **存取方式/访问方式**

数据的存放是严格规定的，**记录的存放是排序的**，查询效率非常高。数据插入以前其实就已经确定了其位置，所以不管插入的先后顺序，它在那个物理上的哪个位置与插入的先后顺序无关。这样在进行查询的时候就可以少访问很多 blocks，但是插入的时候，速度就比普通的表要慢一些。

4. **优点**

- 记录排序，查询效率惊人（最大的优点）
- 提高缓冲区缓存效率，因为给定查询在缓存中需要的 block 更少。
- 减少缓冲区缓存访问，提高可扩展性（每个缓冲区缓存获取都需要缓冲区缓存的多个闩 shuan，而闩是串行化设备，会限制应用的扩展能力）。
- 获取数据的工作总量更少，因为获取数据更快。
- 每个查询完成的物理 I/O 更少。
- 节约磁盘空间的占用，主键没有空间开销，索引就是数据

5. **缺点**

插入效率也许低于堆文件；

对于经常更新的表不适合用 IOT，因为维护索引的开销较大，何况是多字段索引

6. **适用情况**

- **全索引表：**完全由主键组成的表。这样的表如果采用堆组织表，则表本身完全是多余的开销，因为所有的数据全部同样也保存在索引里，此时，堆表是没用的。
- 代码查找表。如果你只会通过一个主键来访问一个表，这个表就非常适合实现为 IOT.
- 如果你想**保证数据存储在某个位置上**，或者希望数据以某种特定的顺序物理存储，IOT 就是一种合适的结构。
- 高频度的一组关联数据查询：经常在一个主键或唯一键上使用 between 查询

4.2 数据分区

1. **分区（一种数据分组方式）**

- **特性：**

  分区能够提高并发性和并行性

  从而增强系统架构的可伸缩性
- **循环分区**：不受数据影响的内部机制，分区定义为各个磁盘的存储区域，可以看做是随意散布数据的机制，报纸带来的磁盘 i/o 操作的平衡

2. **数据驱动分区**

数据驱动分区：根据一个或多个字段中的值来定义分区，是一种手工分区，一般叫分区视图，即 MYSQL 中的 merge table

3. 分区的实现方式

分区是为了方便管理

哈希分区：把不同的列**随机**平均的分布到不同的物理环境，达到备份和恢复（写 undo 和 redo 文件）效率高，降低错误回滚压力（为了管理）

范围分区：把字段的值分布到一个物理范围，这个范围是你在创建分区时指定的分区键决定的。这种分区方式是最为常用的

列表分区：把不同的列存到不同的物理环境，某列的值只有几个，容易按值进行分区

*来自 Note：数据越聚集，检索效率越高，最多的是按**时间**来进行滑动窗口的分区。*

小数据量分区会降低并发的访问性能，大数据量——分区对并发的影响远小于查询带来的效率。

*b) **滑动窗口：按照时间分区***

i. 可以使得最常使用到的数据被聚集在一起，这样可以提高检索效率。绝大部分银行系统都是针对最近的数据加以访问

ii. 可以把最需要的数据（当月、当前的数据）放到最快的物理设备当中，通过物理的部署状况来提高查询效率

4. 分区是双刃剑？优点和缺点

- **问题：它们分别是如何提高查询效率的（考试可能考其中的一个）**

表分区：当表中的数据量不断增大，查询数据的速度就会变慢，应用程序的性能就会下降，这时就应该考虑对表进行分区。表进行分区后，逻辑上表仍然是一张完整的表，只是将表中的数据在物理上存放到多个表空间(物理文件上)，利于高速检索，查询数据时，不至于每次都扫描整张表。*（注：表空间：是一个或多个数据文件的集合，所有的数据对象都存放在指定的表空间中，但主要存放的是表， 所以称作表空间。）*

优点：

- **改善查询性能**：对分区对象的查询可以仅搜索自己关心的分区，高速检索。
- **增强可用性**：如果表的某个分区出现故障，表在其他分区的数据仍然可用;
- **维护方便**：如果表的某个分区出现故障，需要修复数据，只修复该分区即可;
- **均衡 I/O**：可以把不同的分区映射到磁盘以平衡 I/O，改善整个系统性能。

- **分区缺点**
  - 分区表相关：已经存在的表没有方法可以直接转化为分区表。
  - 除了堆文件之外的任何存储方法，都会带来复杂性
  - 选错存储方式会带来大幅度的性能降低
  - 大数据量的并发写入更新效率较低
  - 从本质上来说降低了并发的个数，但是在数据量非常庞大的情况下，降低并发所带来的缺陷远远小于分区所提高的性能
  - 由于强制的数据聚合可能会导致其他数据的分散，所以不同的查询请求也可能会形成性能上的矛盾

5. 分区的最佳方法

- **按什么字段进行分区要整体考虑，因为：更新分区键会引起移动数据，应该避免这么做。**

例如：实现服务队列，类型(T1..Tn),每一个类型三种状态（W|P|D）等待，处理，结束状态

分区：按照请求类型分区，分成（T1.。Tn）个分区：如果有 p1--Pn 个进程来请求，可以有 N 的并发，并发压力均匀的分散到不同的分区，并发进程数可控制的。

按照状态来分区：数据分成三个区域 W，P，D，允许分区键的移动，记录可以跨分区移动，提高处理效率。 将所有 W 状态的放到 W 分区，**降低了轮询的开销**，没有并发问题，不需要锁住某条记录，读 W 一条记录，就删掉，写入 P 区分。没有资源并发冲突。读的进程和处理进程可以单独处理。W 分区读取在等待状态的数据非常快，**不用检索。**

但是分区键的移动，可以降低每个分区对同一个资源的竞争。

分区取决于：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

- **当数据分区键均匀分布时，分区表查询收益最大**

5. 如何处理树状结构

提纲：不同的数据库设计会导致不同的性能，数据库设计，哪些数据库存储方式；邻接模型物化路径；嵌套集合；差别

5.1 一些概念

- 关系型数据库无法直观地解决层次式问题，所以需要一种变换。关系型数据表中的字段之间是平级且等价的，没有层次关系
- **层次式结构（树状结构）不能直接放在关系型数据库中，需要变换一种形式**。树状结构中，节点之间有父子关系，存在兄弟节点，有根节点也有子节点
- **简单树状结构要求：一个节点只有一个父节点；所有的节点类型都是一样的**。如果存在多个父节点，则为物料单 BOM
- 简单树状结构的例子：档案位置（楼->层->房间->橱->柜），找到档案是一个自顶向下的遍历过程；风险分析（解决对冲基金的风险问题，一个基金可能包含多种基金、股票，甚至有可能包含平级的基金。计算一个基金的风险，要计算这个基金的组成部分的加权风险）

5.2 三种树状结构模型

5.2.1 邻接模型

1. **邻接模型**：id，parent_id(指向上级)
   自顶向下查询，假设兄弟节点无序，主要用于单父节点。Connect by 相当容易实现
2. **特性**：
   a) 插入、移动、删除节点快捷
   b) 只支持单父节点，不支持多父节点
   c) Connect by 容易实现
   d) 递归实现，用 oracle 的 with，表示出树的层次

e) 删除子树较难
f) 三种模型中性能最高，每秒返回的查询记录数最多，遍历一次，不是基于关系的处理，性能最好

5.2.2物化路径

1. **物化模型**：PathID（1，1.1，1.2，1.1.1，1.2.1，…），使用层次式的路径明确地标识出来，一般用字符串存路径。每一个节点都存储在树中的位置信息，它允许节点之间有顺序（因为路径的标识有顺序），比如家族族谱
2. **特性**：
   a) 查询编写不困难，找出适当的记录并缩排显示算容易
   b) 计算由路径导出的层次不方便。
   c) 查询复杂度主要在路径字符串的处理
   d) 树的的深度要自己写函数计算，可以计算“.”的数目或者去掉“.”后字符串的长度
   e) 子节点有顺序，但不应该暗示任何兄弟节点的排序
   f) 会产生重复记录的问题
   g) 物化路径 path 不应该该是 KEY，即使他们有唯一性
   h) 所选择的编码方式不需要完全中立
   i) 三种模型中性能中等

5.2.3嵌套集合模型

1. **嵌套集合模型**：每一个节点都有一个左编号，都有一个右编号，left_num,right_num，某节点后代的 left_num 和 right_num 都会在该节点的 left_num 和 right_num 范围内
2. **特性**
   a) 易理解，查找某一个节点的子节点很容易，但是对结果集排序不好操作，缩排无法处理
   b) 适合深度优先遍历
   c) 动态计算深度困难，不要显示人造根节点，为了缩排显示要硬编码最大深度，缩排处理会降低查询性能
   d) 数据元素之间不再是点和线的关系，而是以容纳和被容纳的方式
   e) 计算量大，对存储程序要求高。它是基于指针的解决方案。
   f) 数据更新，删除，插入开销很大，较少使用
   g) 三种模型中，查询的性能最低

5.2.4 自底向上查询（看看就好）

自底向上查询，查询某一个子节点的所有祖先节点

i. 关系操作可以合并相同的父节点，很容易地找到所有的祖先结果集，但是不能做到很好的层次排序

ii. MySQL 有一个机制，可以解决字串索引的问题，substring index

iii. 因为自底向上需要遍历多个节点，而自顶向下只需要遍历一个节点，所以物化路径的方式，自底向上的查询效率远远低于自顶向下的查询效率
iv. 对于嵌套集合模型，自顶向下和自底向上的查询效率相同，它们的差异仅仅体现在自底向上查询的排序过程，这个排序比较耗时间

6. ER 模型到数据库模式

提纲：一对一，部分参与；多对多，部分参与；一对多，部分参与；如何映射
自反映射；继承如何映射

6.1 ER 模型

1. 自顶向下的数据库设计，识别 entity 和 relationship
2. **强实体**：有自己的主键来标识自己的实体，学生和课程
   弱实体： 没有自己的主键来标识自己的实体 ，例如 选课（studentid,courseid）作为自己的主键
但是如果 选课添加了一个（id,studentid,courseid）就是强实体
3. **关联关系**：关联类：人 公司 合同类（雇佣的时间，工资，薪水），
   关联类：约束是人和公司被实例化出来后，只可能有一个关联关系，只会有一个合同，则合同应该是弱实体
如果合同是强实体：人 1 ----- *合同 * （强实体）-------- 1 公司（人和公司可以有多份 id，有多个关联关系）

6.2 关系映射

a) 1:1，一对一全部参与可以放在一张表中，并且不会有空值，避免使用外键
b) 1:1 部分参与必须分在两张表中，如果放在一张表中则会出现很多空值。
   - ✓ 如果是一个全部参与，另一个部分参与的话，则将全部参与的实体的主键放在部分参与的表中作为外键（以此来描述他们的对应关系）；
   - ✓ 如果两个都是部分参与的话，则参与更多的一方的主键放入参与更少的一方的表中作为外键，这样能产生较少的空值

c) 1：N，在“多方”中添加“1”方的外键，如果“1”是强制参与，则外键不能为空，如果是部分参与，则外键可以为空
d) N:N 会产生一个关系表，将相关的两个实体的主键作为复合外键，作为关系实体的主键。这样关系产生的实体是一个弱实体；也可以必要的话将关系变为强实体
e) 递归关系：添加 parent_id
f) 多值属性：添加一个表，将多值属性分离出去，将该表的键，作为主表的外键
g) 强制性地表达表中的完整性约束，no action，cascade，set null，set default，no check。这些关键字说明了主键被删除后，外键应该怎么办

6.3 继承关系---泛化/特化的问题

1) 泛化 generalization：通过识别 entity 之间的共性，最小化差异的过程（父类）
   特化 specialization:通过识别 entity 的独特特性，最大化差异的过程（子类）
2) **数据库泛化的问题。**

比如设置一个 Staff 实体为父实体，它有 manager、salesPersonnel、secretary 三个子实体，它们共享父实体中的属性。这种结构会产生以下的约束

A. **参与约束**，participation constraint，是不是所有的实例都存在于某一个子实体中，分为强制参与（mandatory），部分参与(optional)
   父实体只是一个抽象的概念。即是不是所有的员工都属于某一个职位，如果有一个员工不是子实体中的一员，则他为父实体的实例，这属于部分参与
B. **disjoint constraint，互斥约束。**一个实例是不是只属于一个子实体。
   分为互斥(disjoint)和不互斥(nondisjoint)
   如果一个人,他既是 manager 又是 sales,则 manager 和 sales 这两个子实体是 overlap 的，不是 disjoint 的

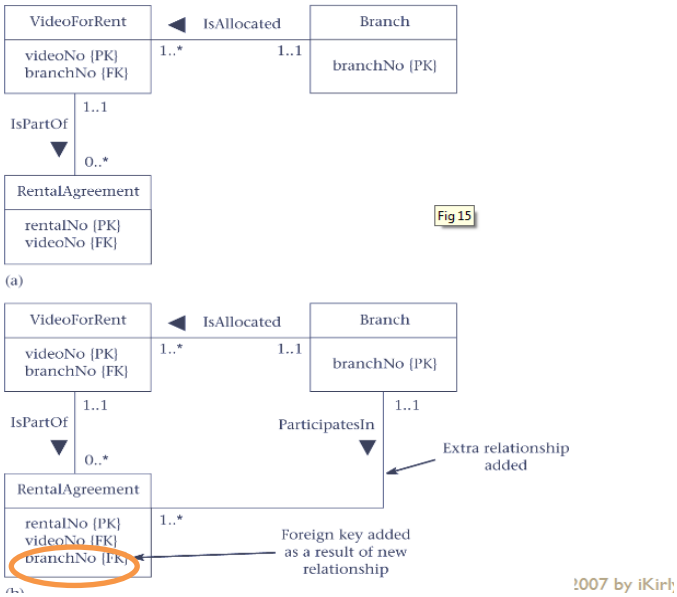
3) 四种特化和泛化的种类

- **a) 强制参与不互斥（mandatory and disjoint）:父和子实体合并为一张表**
  则所有的父实体+子实体都使用一张表，虽然可能会因为某个实例不属于某一个子实体而产生空值，但是产生的空值比较少
- **b) 部分参与不互斥(optional and disjoint)**
  则将父实体做成单独的一张表，所有的子实体做成一张表
- **c) 强制参与互斥(mandatory and nondisjoint)**
  则父实体产生一个主表，子实体分别分成一个表
- **d) 部分参与互斥(optional and nondisjoint)**
  则父实体产生一个主表，子实体分别分成一个表。**只要是 disjoint 的情况，都要将子实体拆分成独立的子表**

6.4 反范式（五种）

逆范式/反范式（打破三范式的规则来生成冗余数据）

1) **合并 1：1 关系**
2) **部分合并 1：*关系，不复制外关键字而复制非 key 的常用的字段**
   - 比如 staff 和 department 之间有**一对多**的关系,将 department 中常被连接查询的字段复制并迁移到 staff 表中（比如 department 的名字是一般需要查询到的，则将名字放入 staff 表中,这样查询 staff 所属部门的名字时就不需要连接两个表了），**降低连接查询的数量**,原来的 department 表还是留下
   - 使用**“一致性控制”模式**来保证两个表中冗余字段的一致性，这种**控制放在 UI 界面**中进行，使用 combinebox 或者 check box 进行选择，而不是获取用户的输入
3) **在 1：*关系中复制 FK（外关键字），减少 join 的表数量，将另一个表的主键复制变成外键，可以看一下，示例**



4) **在*：*关系中复制属性**

- 比如多对多关系的 orders、articles 的关系，有**一个关系表**中保存了 oid 和 aid。如果要知道某个订单中某个产品的名字和价格的话，则需要**三表连接**
- 在现今的电子商务系统中，article 的**某些属性会直接复制到关系表中**，即关系表中除了 oid 和 aid 之后，还包括 price(price 现在已经是缺省放入关系表的字段)、article name、数量

5) 引入重复组，将多值属性写在主表里：例如 user 表中多个电话号码的列，常用地址和常用电话

6) 为了避免查询和更新这两个不可调和的矛盾，**可以将更新和查询放在两张表中**，从工作表提取出查询表，专门用于查询。这个方法演化成了数据仓库。**这个方法只适用于查询的实时性要求不高的情况**