

数据库复习

目录

1. 数据库设计的原则,比较集中式分布式的区别(结构上的差异,工作方式的差异,分别适合什么应用)	2
1.1 集中式 DBS:	2
1.2 分布式 DBS:	2
2. B 树索引,结构,怎么工作,什么时候用 B 树索引, B 树索引的限制,用了之后是什么状况。其他索引(位图索引,哈希索引,倒排索引,函数索引大概是什么样子,适合什么场景)	3
2.1 索引概念.....	3
2.2 索引的优点,为什么使用索引?	4
2.3 索引的局限性(索引的限制)	4
2.4 其他索引.....	6
3.sql 的优化(两个题之一) 执行步骤,软解释,硬解释,哪些步骤消耗代价,消耗代价的情况,中间解析过程的情况.....	7
3.0 执行步骤.....	7
3.1SQL 和优化器概念	8
3.2 优化策略.....	9
3.3 加倍留意非关系操作.....	9
3.4 优化器的有效范围.....	9
3.5 使用 SQL 语句要考虑的因素.....	9
3.6 过滤.....	10
3.7SQL 语句方面注意	10
3.8SQL 查询优化总结	11
3.9 大数据量查询原则.....	11
3.10 非关联子查询变成内嵌视图—降低查询维度	11
4.嵌套查询判断消耗(两个题之二)	12
4.1 摆脱 distinct,使用 exists 和 in 操作.....	12
5.降维优化的过程.....	14
6.物理组织形式,读数据,存数据,怎么去判断这样的情况,怎么插入、更新,行迁移,堆文件为什么是最常用的文件组织形式,其他文件哪些方面比他强,哪些比他弱	15
6.1 堆文件.....	15
6.2 索引组织表 IOT	15
6.3 数据分区.....	16
7.树状结构,多种方式,自顶向上,自顶向下,使用的不同范围	18
7.1 一些概念.....	18
7.2 三种树状结构模型*	18
7.3 查询方式.....	19
8.范式、反范式,什么时候用	22

1. 数据库设计的原则，比较集中式分布式的区别（结构上的差异，工作方式的差异，分别适合什么应用）

1.1 集中式 DBS:

定义：是指数据库中的数据集中存储在一台计算机上，数据的处理集中在一台计算机上完成。

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

集中式数据库系统的优缺点：

集中式数据库系统的优点：

在集中式数据库里，大多数功能（如修改、备份、查询、控制访问等）都很容易实现。数据库大小和它所在的计算机不需要担心数据库是否在中心位置。例如，小企业可以在个人计算机（PC）上设立一个集中式数据库，而大型企业可以由大型机来控制整个数据库。

集中式数据库系统的缺点：

当中心站点计算机或数据库系统不能运行时，在系统恢复之前所有用户都不能使用系统。从终端到中心站点的通信开销是很昂贵的。

1.2 分布式 DBS:

定义：是指数据存放在计算机网络的不同场地的计算机中，每一场地都有自治处理能力并完成局部应用；而每一场地也参与(至少一种)全局应用程序的执行，全局应用程序可通过网络通信访问系统中的多个场地的数据。

结构：一个分布式数据库在逻辑上是一个统一的整体，在物理上则是分别存储在不同的物理节点上。一个应用程序通过网络的连接可以访问分布在不同地理位置的数据库。它的分布性表现在数据库中的数据不是存储在同一场地。更确切地讲，不存储在同一计算机的存储设备上。这就是与集中式数据库的区别。从用户的角度看，一个分布式数据库系统在逻辑上和集中式数据库系统一样，用户可以在任何一个场地执行全局应用。就好像那些数据是存储在同一台计算机上，有单个数据库管理系统（DBMS）管理一样，用户并没有什么感觉不一样。

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

分布式数据库系统的优缺点：

分布式数据库系统的优点：

- (1)具有灵活的体系结构
- (2)适应分布式的管理和控制机构
- (3)经济性能优越
- (4)系统的可靠性高、可用性好
- (5)局部应用的响应速度快

(6)可扩展性好，易于集成现有系统。

分布式数据库系统的缺点：

(1)系统开销大，主要花在通信部分

(2)复杂的存取结构，原来在集中式系统中有效存取数据的技术，在分成式系统中都不再适用

(3)数据的安全性和保密性较难处理。

2. B 树索引，结构，怎么工作，什么时候用 B 树索引，B 树索引的限制，用了之后是什么状况。其他索引（位图索引，哈希索引，倒排索引，函数索引大概是什么样子，适合什么场景）

2.1 索引概念

1. 索引是对数据库表中的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

2.2 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，同样可以显著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。

2.3 索引的局限性（索引的限制）

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。**

2.4 其他索引

2.4.1 位图索引

主要针对**大量相同值的列**而创建(例如: 类别, 操作员, 部门 ID, 库房 ID 等), 索引块的一个索引行中存储键值和起止 Rowid, 以及这些键值的位置编码, 位置编码中的每一位表示键值对应的数据行的有无. 一个索引块可能指向的是几十甚至成百上千行数据的位置.

这种方式存储数据, 相对于 B*Tree 索引, 占用的空间非常小, 创建和使用非常快.

2.4.2 哈希索引

所谓 Hash 索引, 实际上就是通过**一定的 Hash 算法**, 将需要索引的键值进行 Hash 运算, 然后将得到的 Hash 值存入一个 Hash 表中. 每次需要检索的时候, 都会将检索条件进行相同算法的 Hash 运算, 再和 Hash 表中的 Hash 值进行比较, 并得出相应的信息.

HASH 索引在有限制条件(需要指定一个确定的值而不是一个值范围)的情况下非常有用.

HASH 的缺点

- (1) Hash 索引仅仅能满足 "=", "IN" 和 "<=>" 查询, 不能使用范围查询。

由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值, 所以它只能用于等值的过滤, 不能用于基于范围的过滤, 因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系, 并不能保证和 Hash 运算前完全一样。

- (2) Hash 索引无法被用来避免数据的排序操作。

由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值, 而且 Hash 值的大小关系并不一定和 Hash 运算前的键值完全一样, 所以数据库无法利用索引的数据来避免任何排序运算;

- (3) Hash 索引不能利用部分索引键查询。

对于组合索引, Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值, 而不是单独计算 Hash 值, 所以通过组合索引的前面一个或几个索引键进行查询的时候,

Hash 索引也无法被利用。

(4) Hash 索引在任何时候都不能避免表扫描。

前面已经知道, Hash 索引是将索引键通过 Hash 运算之后, 将 Hash 运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中, 由于不同索引键存在相同 Hash 值, 所以即使取满足某个 Hash 键值的数据的记录条数, 也无法从 Hash 索引中直接完成查询, 还是要通过访问表中的实际数据进行相应的比较, 并得到相应的结果。

(5) Hash 索引遇到大量 Hash 值相等的情况后性能并不一定会比 B-Tree 索引高。

对于选择性比较低的索引键, 如果创建 Hash 索引, 那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦, 会浪费多次表数据的访问, 而造成整体性能低下。

2.4.3 函数索引

基于函数的索引, 类似于普通的索引, 只是普通的索引是建立在列上, 而它是建立在函数上。当然这回对插入数据有一定影响, 因为需要通过函数计算一下, 然后生成索引。但是插入数据一般都是少量插入, 而查询数据一般数据量比较大。函数索引还有一个功能, 只对部分行建立索引。

2.4.4 倒排索引

也常被称为反向索引、置入档案或反向档案, 是一种索引方法, 被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。通过倒排索引, 可以根据单词快速获取包含这个单词的文档列表。倒排索引主要由两个部分组成: “单词词典”和“倒排文件”。

“倒排索引”是实现单词到文档映射关系的最佳实现方式和最有效的索引结构。

3.sql 的优化（两个题之一）执行步骤，软解释，硬解释，哪些步骤消耗代价，消耗代价的情况，中间解析过程的情况

3.0 执行步骤

语法检查：检查 SQL 拼写是否符合语法规范

语义检查: 检查访问对象是否符合存在及用户是否具有相应权限

解析: 在共享池中检查是否有完全相同的之前完全解析好的, 如果存在, 跳过选择执行计划和产生计划, 直接运行

硬解析: 就是对提交的 SQL 完全重新从头进行解析, 创建解析树, 生成执行计划对 SQL 的执行来说是开销昂贵的动作, 在很多项目中对功能相同的代码要保持一致性, 用绑定变量

软解析: 在共享池 (shared pool) 中找到了与之完全相同的 SQL 解析好的结果会跳过硬解析后面的两个步骤

执行计划: 以缩排列表的方式显示 SQL 语句的执行步骤

3.1 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(Supergroups)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。

3.2 优化策略

3.3 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
```

3.4 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

3.5 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，重要的是用户的感受
- 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，锁定，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

3.6 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

3.7 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

3.8SQL 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

3.9 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 having 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

3.10 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
```

4. 嵌套查询判断消耗（两个题之二）

4.1 摆脱 distinct，使用 exists 和 in 操作

- 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
- 摆脱 distinct 的方法：使用 exists 和 in，exists 需要用到嵌套子查询：
 - 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
 - ✓ exists 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

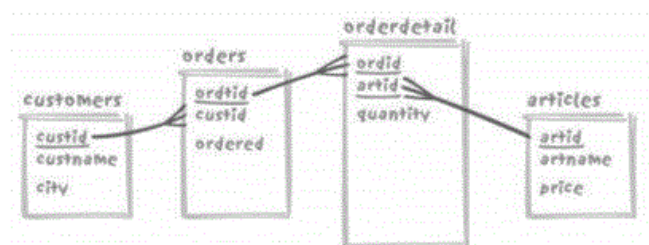
返回；子查询时间难以估算。经过查询优化器优化后，优化的时间远远超过执行的时间，虽然执行多次，效率反而更高

- ✓ exists 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
```

非关联子查询

```
select custname
```

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。

<pre> 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) </pre> <p>Exists嵌套子查询 和外层select关系 非常密切</p>	<p>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</p>	<pre> 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) </pre> <p>因为要用到的索引是customers 的主键索引</p> <p>内层查询不再依赖 外层查询，只需要 执行一次</p>
---	--	--

其他方法：from 子句中的内嵌视图

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 exists 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

5.降维优化的过程

最主要的是非关联子查询变成内嵌视图，应该会考写语句

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

```

2013/11/10 by iKarily 37

6.物理组织形式，读数据，存数据，怎么去判断这样的情况，怎么插入、更新，行迁移，堆文件为什么是最常用的文件组织形式，其他文件哪些方面比他强，哪些比他弱

- **行迁移：** 当一个行上的更新操作(原来的数据存在且没有减少)导致当前的数据不能在容纳在当前块，我们需要进行行迁移。一个行迁移意味着整行数据将会移动，仅仅保留的是一个转移地址。因此整行数据都被移动，原始的数据块上仅仅保留的是指向新块的

一个地址信息。

产生: update

6.1 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。

6.2 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 I/O, 而 IOT 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

6.3 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**

7.树状结构，多种方式，自顶向上，自顶向下，使用的不同范围

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

7.1 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）

7.2 三种树状结构模型*

7.2.1 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

7.2.2 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

7.2.3 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

7.3 查询方式

7.3.1 自顶向下 (Vandamme 查询)

邻接模式:递归实现, 效率较低, 使用 **CONNECT BY** 实现

```
select lpad(description, length(description) + level) description,
       commander
from adjacency_model
connect by parent_id = prior id
start with commander = 'Général de Division Dominique Vandamme'
```

使用递归实现

```

with recursive_query(level, id, description, commander)
as (select 1 level,
        id,
        description,
        commander
    from adjacency_model
    where commander = 'Général de Division Dominique Vandamme'
    union all
    select parent.level + 1,
        child.id,
        child.description,
        child.commander
    from recursive_query parent,
        adjacency_model child
    where parent.id = child.parent_id)
select char(concat(repeat(' ', level), description), 60) description,
        commander
from recursive_query

```

物化路径模型：path 中前半段相同的即为结果，假设 mp_depth()函数返回当前节点深度

```

select lpad(a.description, length(a.description)
        + mp_depth(...)) description,
        a.commander
from materialized_path_model a,
        materialized_path_model b
where a.materialized_path like b.materialized_path || '%'
    and b.commander = 'Général de Division Dominique Vandamme')
order by a.materialized_path

```

嵌套集合模型：直接找到在 left_num 和 right_num 中间的所有节点，缩进比较难处理。


```

select a.description,
       a.commander
from nested_sets_model a,
     nested_sets_model b
where a.left_num between b.left_num and b.right_num
     and b.commander = 'Général de Division Dominique
Vandamme'

```

自顶向下效率：嵌套集合 < 物化路径 < 邻间模型

7.3.2 自底向上（Highland 查询）

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，可以解决字符串索引的问题，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

邻间模型：

```

select lpad(description, length(description) + level) description,
       commander
from adjacency_model
connect by id = prior parent_id
start with description like '%Highland%'

```

物化路径：

```

select description, commander
from (select distinct lpad(a.description, length(a.description)
      + mp_depth(b.materialized_path)
      - mp_depth(a.materialized_path)) description,
      a.commander,
      a.materialized_path
      from materialized_path_model a,
      materialized_path_model b
      where b.materialized_path like a.materialized_path || '%'
      and b.description like '%Highland%')
order by materialized_path desc

```

嵌套集合模型:

```

select lpad(description, length(description) + 6 - depth) description,
      commander
from (select distinct b.description,
      b.commander,
      b.left_num,
      (select count(c.left_num)
       from nested_sets_model c
       where b.left_num between c.left_num
       and c.right_num) depth
      from nested_sets_model a,
      nested_sets_model b
      where a.description like '%Highland%'
      and a.left_num between b.left_num and b.right_num
      and b.left_num > 1)
order by left_num desc

```

自底向上：邻间>物化路径>嵌套集合

8. 范式、反范式，什么时候用

1. 一、二、三范式

1NF: 同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的值（确保原子性）

2NF: 满足 1NF 且要求 DB 表中的每个实例或行必须可以被组成的主键的所有列唯一地区分（检查对键的完全依赖）

3NF: 满足 2NF 且要求一个数据库表的非主关键字只能被主关键字区分(检查属性的独立性)

2. 反范式

逆范式/反范式(打破三范式的规则来生成冗余数据)

七大反范式 Pattern.

Pattern 1 合并 1 对 1 关系

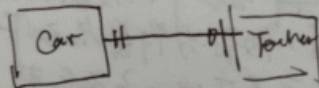
问题是会产生大量冗余值, 若两边都部分参与时不能合并
部分参与为绝大部分都参与时再运用 Pattern 1.

Example:

学院给老师配车. 车少人多, 车完全参与, 人部分参与

Carid	Carname
1	BMW
2	BMW
3	QQ

Teacherid	Teachername
1	Huang
2	Zhang
3	Chen
4	Liu



合并

Car And Teacher

Carid	Carname	Teacherid	Teachername	Carid	Carname
1	BMW	1	Huang	1	BMW
2	BMW	2	Zhang	2	BMW
3	QQ	3	Chen	3	QQ
		4	Liu	Null	Null

Pattern 2 1对N~关系用非键值进行连接.

通用条件: 两表连接时主键与主键部与某非键值相关

如: 查询学生以及该生所在学院

Faculty

Fid	Fname
125	软件

Student

Sid	Sname	Fid	Fname
111250204	xxx	125	软件

引键时

① 如果在UI上, 只允许用户进行选择, 不能自行输入
保证输入一一致性

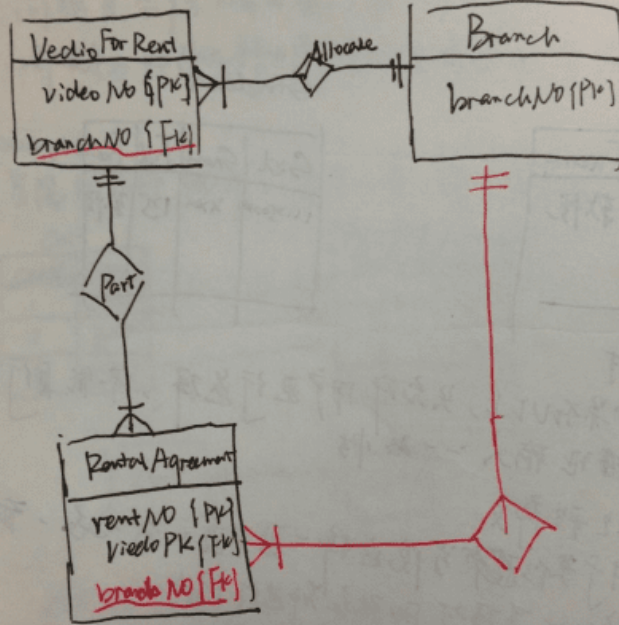
如果上程序员:

对于类似学院这样一般不变=代码表, 在
修改时直接对两张表都进行修改

若表经常变, 则加个 trigger 在表中

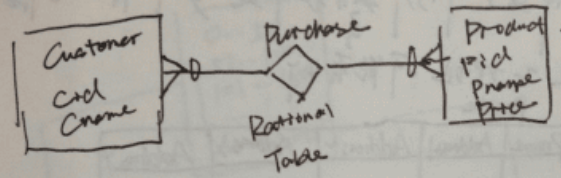
Pattern 3

在1对N关系中复制外键，减少Join表数量，将多表
二多键复制成外键

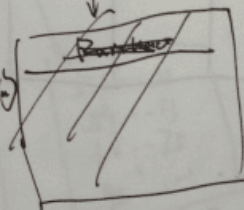


Pattern 4

多对多关系中复制属性，把两张表中经常需要一起
 查询的列中间一关系表中。



减少三表连接查询



Purchase				
cid	Cname	pid	Pname	Price

Pattern 5. 引入重复组.

通常对于一个多值属性, 值不太多(≤ 10), 且不会经常变, 可以在表中建多个有关此属性的列.
 如一个地址, 可能有4个

People

Prod	Prname	Address1	Address2	Address3	Address4

Pattern 6 建立抽取表.

为避免查询与更新这两个不同操作冲突.
 可以将更新与查询放在两张表中, 从2张表抽取查询名, 专门用于查询, 只适用于查询, 系统忙不上一样先

Pattern 7 分区.

列分区: 数据冗余, 关键每次都要.

表大, 有些列不常用, 采用分区方式, 常用~在一个分区, 不常用~另一个分区

id	name	sex	age	phone