1. 提高数据库性能的原则？（数据库连接，集中，分散等方面）

关系理论：关系不包含重复数据，记录间无顺序

限用boolean型字段：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

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

处理流程：异步批处理，同步实时交易

数据集中化：离数据越近，访问速度越快。代码里DBMS核心越近运行越快,充分利用每次数据库访问。Block使用充分，命中率高，所需io操作少（分散：独占资源的时间对其他事务的影响，有利于高并发）

保持数据库稳定：连接稳定，减少交互

把逻辑放入查询中：把条件逻辑放到SQL语句中，而不是宿主语言中

慎用自定义函数：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

一、在程序方面，  
1-》保证在实现功能的基础上，尽量减少对数据库的访问次数；  
2-》通过搜索参数，尽量减少对表的访问行数,最小化结果集，从而减轻网络负担；  
3-》能够分开的操作尽量分开处理，提高每次的响应速度；  
4-> 在数据窗口使用SQL时，尽量把使用的索引放在选择的首列；  
5-> 算法的结构尽量简单；  
6-> 在查询时，不要过多地使用通配符如SELECT \* FROM T1语句，要用到几列就选择几列如：SELECT COL1,COL2 FROM T1；  
7-> 在可能的情况下尽量限制尽量结果集行数如：SELECT TOP 300 COL1,COL2,COL3 FROM T1,因为某些情况下 用户是不需要那么多的数据的。  
8-> 不要在应用中使用数据库游标，游标是非常有用的工具,但比使用常规的、面向集的SQL语句需要更大的开销；  
按照特定顺序提取数据的查找。

二  数据库方面。

1  避免使用不兼容的数据类型。例如float和int、char和varchar、binary和varbinary是不兼容的。  
数据类型的不兼容可能使优化器无法执行一些本来可以进行的优化操作。例如:   
SELECT name FROM employee WHERE salary ＞ 60000   
在这条语句中,如salary字段是money型的,则优化器很难对其进行优化,因为60000是个整型数。  
我们应当在编程时将整型转化成为钱币型,而不要等到运行时转化。

2  尽量避免在WHERE子句中对字段进行函数或表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
SELECT \* FROM T1 WHERE F1/2=100  
应改为:   
SELECT \* FROM T1 WHERE F1=100\*2

SELECT \* FROM RECORD WHERE SUBSTRING(CARD\_NO,1,4)=’5378’  
应改为:  
SELECT \* FROM RECORD WHERE CARD\_NO LIKE ‘5378%’

SELECT member\_number, first\_name, last\_name   FROM members  
WHERE DATEDIFF(yy,datofbirth,GETDATE()) > 21  
应改为:  
SELECT member\_number, first\_name, last\_name   FROM members  
WHERE dateofbirth < DATEADD(yy,-21,GETDATE())  
即：任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。

3   避免使用!=或＜＞、IS NULL或IS NOT NULL、IN ，NOT IN等这样的操作符,  
因为这会使系统无法使用索引,而只能直接搜索表中的数据。例如:   
SELECT id FROM employee WHERE id != 'B%'   
优化器将无法通过索引来确定将要命中的行数,因此需要搜索该表的所有行。

4 合理的建立索引 提高查询效率。

5、尽量使用数字型字段，一部分开发人员和数据库管理人员喜欢把包含数值信息的字段设计为字符型，  
这会降低查询和连接的性能，并会增加存储开销。  
这是因为引擎在处理查询和连接回逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

6、合理使用EXISTS,NOT EXISTS子句。如下所示：  
1.SELECT SUM(T1.C1)FROM T1 WHERE(  
(SELECT COUNT(\*)FROM T2 WHERE T2.C2=T1.C2>0)  
2.SELECT SUM(T1.C1) FROM T1WHERE EXISTS(  
   SELECT \* FROM T2 WHERE T2.C2=T1.C2)  
两者产生相同的结果，但是后者的效率显然要高于前者。因为后者不会产生大量锁定的表扫描或是索引扫描。  
如果你想校验表里是否存在某条纪录，不要用count(\*)那样效率很低，而且浪费服务器资源。可以用EXISTS代替。如：  
IF (SELECT COUNT(\*) FROM table\_name WHERE column\_name = 'xxx')  
可以写成：  
IF EXISTS (SELECT \* FROM table\_name WHERE column\_name = 'xxx')

经常需要写一个T\_SQL语句比较一个父结果集和子结果集，从而找到是否存在在父结果集中有而在子结果集中没有的记录，如：  
1.SELECT a.hdr\_key   FROM hdr\_tbl a---- tbl a 表示tbl用别名a代替  
WHERE NOT EXISTS (SELECT \* FROM dtl\_tbl b WHERE a.hdr\_key = b.hdr\_key)

2.SELECT a.hdr\_key   FROM hdr\_tbl a  
LEFT JOIN dtl\_tbl b ON a.hdr\_key = b.hdr\_key   WHERE b.hdr\_key IS NULL

3.SELECT hdr\_key   FROM hdr\_tbl  
WHERE hdr\_key NOT IN (SELECT hdr\_key FROM dtl\_tbl)   
三种写法都可以得到同样正确的结果，但是效率依次降低。

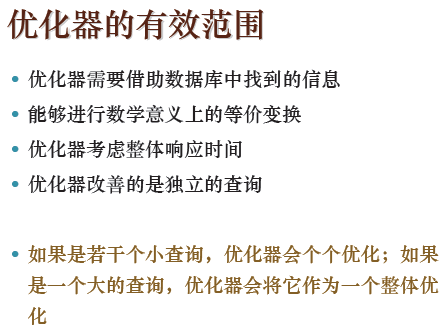
7、尽量避免在索引过的字符数据中，使用非打头字母搜索。这也使得引擎无法利用索引。    
见如下例子：  
SELECT \* FROM T1 WHERE NAME LIKE ‘%L%’  
SELECT \* FROM T1 WHERE SUBSTING(NAME,2,1)=’L’  
SELECT \* FROM T1 WHERE NAME LIKE ‘L%’  
即使NAME字段建有索引，前两个查询依然无法利用索引完成加快操作，引擎不得不对全表所有数据逐条操作来完成任务。  
而第三个查询能够使用索引来加快操作。

8、充分利用连接条件，在某种情况下，两个表之间可能不只一个的连接条件，  
这时在   WHERE 子句中将连接条件完整的写上，有可能大大提高查询速度。例：  
SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD\_NO = B.CARD\_NO   
SELECT SUM(A.AMOUNT) FROM ACCOUNT A,CARD B WHERE A.CARD\_NO = B.CARD\_NO   AND A.ACCOUNT\_NO=B.ACCOUNT\_NO  
第二句将比第一句执行快得多。

9、消除对大型表行数据的顺序存取，尽管在所有的检查列上都有索引，但某些形式的WHERE子句强迫优化器使用顺序存取。如：  
SELECT \* FROM orders WHERE (customer\_num=104   AND order\_num>1001) OR order\_num=1008  
解决办法可以使用并集来避免顺序存取：  
SELECT ＊ FROM orders WHERE customer\_num=104 AND order\_num>1001   
UNION   
SELECT ＊ FROM orders WHERE order\_num=1008   
这样就能利用索引路径处理查询。

10、避免困难的正规表达式。使用 LIKE关键字支持通配符匹配，技术

1. SQL 优化（针对查询优化器的隐性 SQL 优化）



获得结果集所需访问的数据量：

定义结果集所需的查询条件：where子句过滤条件 where与having join与select

结果集的大小：

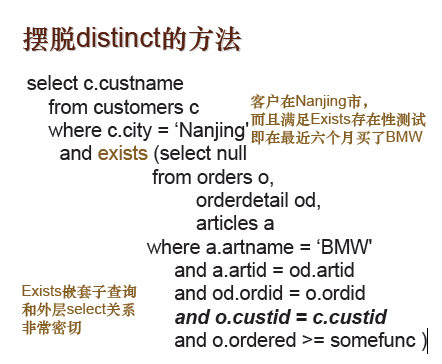
获得结果集所涉及的表的数量：表的数量增加时复杂度增加，多种方式连接的选择失误几率变高

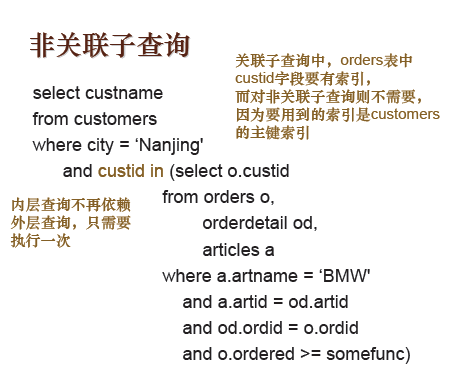
同时修改这些数据用户的多少：并发用户数（数据块访问，阻塞，闩定，保证读取一致性）

）

例子：BMW

避免在最高层distinct应该是一条基本规则





◦发现重复数据容易，发现不准确的连接难

◦发现结果不正确就更难了

大数据量查询

* 越快剔除不需要的数据，查询的后续阶段必须处理的数据量就越少，查询效率就越高
* Set operator是对该原则很好的应用

◦Union语句，但是不要cut-and-paste

* Group by & having子句也存在类似问题

◦所有影响聚合函数结果的条件都应在Having子句

◦任何无关聚合条件都应该放在where子句

◦减少group by必须执行排序操作所处理的数据量

关于DISTINCT，嵌套 SQL，EXISTS 和 IN 的差异

NOT EXITS比NOT IN快，EXITS比IN慢

exits适合内小外大的查询，in适合内大外小的查询

EXITS嵌套子查询和外层select关系非常密切

IN 内层查询不再依赖外层查询，只需要执行一次

优化器借助关系理论提供的语义无误的原始查询进行有效的等价变换

MySQL查询优化器有几个目标，但是其中最主要的目标是尽可能地使用索引，并且使用最严格的索引来消除尽可能多的数据行

SELECT col3 FROM mytable  
WHERE col1 = ’some value’ AND col2 = ’some other value’;   
　　假设col1上的测试匹配了900个数据行，col2上的测试匹配了300个数据行，而同时进行的测试只得到了30个数据行。先测试Col1会有900个数据行，需要检查它们找到其中的30个与col2中的值匹配记录，其中就有870次是失败了。先测试col2会有300个数据行，需要检查它们找到其中的30个与col1中的值匹配的记录，只有270次是失败的，因此需要的计算和磁盘I/O更少。其结果是，优化器会先测试col2，因为这样做开销更小。

尽量比较数据类型相同的数据列。

尽可能地让索引列在比较表达式中独立。

在LIKE模式的开头不要使用通配符。

From 子句中写在最后的表（基础表）将最先被处理，选择记录少的表作为基础表

那些可以过滤掉最大数量记录的条件必须写在where子句的末尾

1. 索引。目的：提高查询效率。

建立索引的目的是加快对表中记录的查找或排序。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。创建索引可以大大提高系统的性能。第一，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。第二，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。第三，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。第四，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。第五，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。

　　也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。第一，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。第二，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。第三，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。

索引是一种以原子粒度访问数据库的手段，而不是为了检索大量数据（网上）

* 函数和类型转换对索引的影响where f(index\_col)=’some value’这种检索条件会使索引无法发挥作用
* 为每一个外键建立索引可能会导致多余索引，多个外键构成主键时考虑顺序，会影响效率（ppt）

结构：B 树

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，而且可以使用使用索引而不用表

因为B树的特点就是适合在磁盘等直接存储设备上组织动态查找表。

为什么不使用索引？六点（不要标题党）

返回不正确结果或降低了效率

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

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

1. 使用SELECT COUNT(\*) FROM T，而且T上有索引，但是优化器仍然全表扫描
2. 对于一个有索引的列作出函数查询

Select \* from t where f(indexed\_col) = value

1. 隐形函数查询
2. 此时使用了索引，世纪反而会更慢
3. 没有正确的统计信息，造成CBO无法做出正确选择

1如果每次都需要取到所有表记录，无论如何都必须进行全表扫描了，那么是否加索引也没有意义了。2、对非唯一的字段，例如“性别”这种大量重复值的字段，增加索引也没有什么意义。3、对于记录比较少的表，增加索引不会带来速度的优化反而浪费了存储空间，因为索引是需要存储空间的，而且有个致命缺点是对于update/insert/delete的每次执行，字段的索引都必须重新计算更新。

1. 数据库的物理组织：  
   堆文件

IOT 索引组织表：全索引表，代码查找表，高频度的一组关联数据查询。最大优点：记录是排序的（有序的实现多采用索引），多表关联的时候（外键连接查询）会有麻烦

操作比较固定，没有大量的更新操作的表。

对于频繁的插入操作的表。iot性能不好，在插入数据时，要寻找插入到那个块，若块的大小不足，需要放在溢出块中，这些操作都需要消耗资源。

heap table 就是一般的表，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，在进行全表扫描的时候，并不是先插入的数据就先获取。数据的存放也是随机的，当然根据可用空闲的空间来决定。  
而iot 就是类似一个全是索引的表，表中的所有字段都放在索引上，所以就等于是约定了数据存放的时候是按照严格规定的，在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，它在那个物理上的那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。  
适用于信息检索、空间和OLAP程序。索引组织表的适用情况：  
1、 代码查找表。  
2、 经常通过主码访问的表。  
3、 构建自己的索引结构。  
4、 加强数据的共同定位，要数据按特定顺序物理存储。  
5、 经常用between…and…对主码或唯一码进行查询。数据物理上分类查询。如一张订单表，按日期装载数据，想查单个客户不同时期的订货和统计情况。  
经常更新的表当然不适合iot，因为oracle需要不断维护索引，而且由于字段多索引成本就大。

分区

数据分组的方式。提高并发性和并行性，从而增强系统架构的可伸缩性。对分区表进行查询，当数据按分区键均匀分布时，收益最大

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，可以看作是随意散布数据的机制，保持更改带来的磁盘I/O操作的平衡

数据驱动分区：根据一个或多个字段中的值来定义分区，手工分区，一般叫分区视图（partitioned view），而MYSQL称为（merge table）

分区的实现方式

◦哈希分区（Hash-partitioning）

◦范围分区（Range-partitioning）

◦列表分区（List-partitioning）

分区是把双刃剑

A. 通过分区键将数据聚集，利于高速检索；

B. 对并发执行的更改操作，分散的数据可以避免访问过于集中的问题

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）

1. 表结构设计。  
   例子：如何把一棵树存到二维表中？  
   一，<id, pid>方式；二，物化路径；三，嵌套集合  
   问题：这三种方式的性能区别

邻接模型：id，parent\_id(指向上级) 自顶向下查询，递归实现。Connect by相当容易实现

物化模型：物化路径 计算由路径导出的层次不方便。仅找出适当的记录并缩排显示算容易，重复记录问题，顺序问题

嵌套集合模型：left\_path,right\_path 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

1. 附加题，与上课内容无关的 SQL 编写。