# 数据库(database)

## 数据库事务

### 四大特性(ACID)

#### 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

#### 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

#### 隔离性（Isolation）

事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

不考虑事务的隔离性，会发生的几种问题:

##### 脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

##### 不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

##### 幻读

幻读，因为事务1读取的数据状态并不能支持他的下一步的业务，见鬼了一样。

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

　　幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

#### 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

　　例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

### 隔离级别

在MySQL数据库中，支持下面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

#### Serializable（串行化）

可避免脏读、不可重复读、幻读的发生。

#### Repeatable read（可重复读）

可避免脏读、不可重复读的发生。

#### Read committed（读已提交）

可避免脏读的发生。

#### Read uncommitted（读未提交）

最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。

## 数据库优化

### 百万级数据库优化方案

#### 避免全表扫描

对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

#### 避免null值判断

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL也包含在内），都是占用 100个字符的空间的，如果是varchar这样的变长字段， null 不占用空间。

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num = 0

#### 避免使用!=或<>操作符

应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

#### 避免使用or连接条件

应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or Name = 'admin'

可以这样查询：

select id from t where num = 10

union all

select id from t where Name = 'admin'

#### 慎用in和not in条件

in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

#### 慎用like条件

select id from t where name like ‘%abc%’

若要提高效率，可以考虑全文检索。

#### 避免在where子句中使用参数

如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num = @num

可以改为[强制使用索引](#_强制使用索引)：

select id from t with(index(索引名)) where num = @num

select /\*+ index(table IndexName)\*/\* from t where num = @num

#### 避免在 where 子句中对字段进行表达式操作

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2 = 100

应改为:

select id from t where num = 100\*2

#### 避免在where子句中对字段进行函数操作

应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3) = ’abc’ -–name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′) = 0 -–‘2005-11-30’ --生成的id

应改为:

select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'

#### 避免在where 子句中的“=”左边进行函数、算术运算或其他表达式运算

不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

#### 复合索引必须使用该索引的第一个字段

在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

#### 先分页再join

对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

#### 索引并不是越多越好

索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

#### 应尽可能的避免更新[聚合](http://www.baidu.com/link?url=f3MEUUvzoVZ8wfzYsWu0b5p1u-9aoTrl4VsYr8Nfk_hZvTOTsaPCYmLgzGXSjskWfDGBBc_M11UZOSV7mkQN-K)(clustered)索引数据列

因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

#### 尽量使用数字型字段

若只含数值信息的字段尽量不要设计为字符型，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 接时会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。

#### 尽可能的使用 varchar/nvarchar 代替 char/nchar

因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

#### 避免使用select \*

任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

#### 尽量使用表变量来代替临时表

如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

#### 避免频繁创建和删除临时表

避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

#### 新建临时表使用 select into 代替 create table

在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替create table，避免造成大量log，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

#### 临时表先 truncate table，然后 drop table

如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

#### 尽量避免使用游标

因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

#### 基于集的解决方案

使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

#### 游标并不是不可使用

与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

#### SET NOCOUNT ON和SET NOCOUNT OFF的使用

在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE\_IN\_PROC 消息。

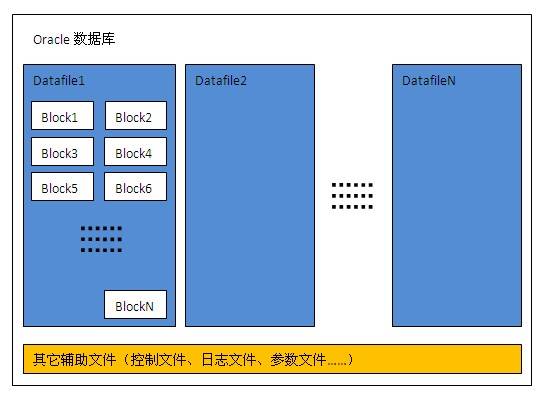
#### 尽量避免大事务操作

尽量避免大事务操作，提高系统并发能力。

## Oracle

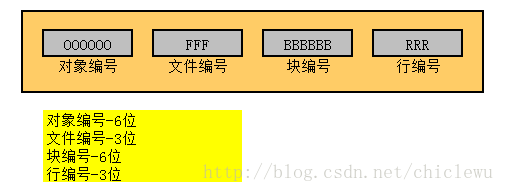
### 数据块(Block)

数据块是数据库中数据在磁盘中存储的最小单位，也是一次IO访问的最小单位，一个数据块通常可以存储多条记录，数据块大小是DBA在创建数据库或表空间时指定，可指定为2K、4K、8K、16K或32K字节。下图是一个[Oracle](http://lib.csdn.net/base/oracle)数据库典型的物理结构，一个数据库可以包括多个数据文件，一个数据文件内又包含多个数据块:



### ROWID

ROWID是每条记录在数据库中的唯一标识，通过ROWID可以直接定位记录到对应的文件号及数据块位置。ROWID内容包括对像号、文件号、数据块号、记录槽号，如下图所示：



### 索引

#### 强制使用索引

强制索引语法：SELECT /\*+index(table IndexName)\*/\* from table

如果查询语句中，表用到别名时，上边语句中的table一定要是别名，否则不走执行强制索引。

#### 索引使用条件

INDEX\_COLUMN = ?

INDEX\_COLUMN > ?

INDEX\_COLUMN >= ?

INDEX\_COLUMN < ?

INDEX\_COLUMN <= ?

INDEX\_COLUMN between ? and ?

INDEX\_COLUMN in (?,?,...,?)

INDEX\_COLUMN like ?||'%'（后导模糊查询）

T1. INDEX\_COLUMN=T2. COLUMN1（两个表通过索引字段关联）

#### 索引不使用条件

|  |  |
| --- | --- |
| 查询条件 | 不能使用索引原因 |
| INDEX\_COLUMN <> ?  INDEX\_COLUMN not in (?,?,...,?) | 不等于操作不能使用索引 |
| function(INDEX\_COLUMN) = ?  INDEX\_COLUMN + 1 = ?  INDEX\_COLUMN || 'a' = ? | 经过普通运算或函数运算后的索引字段不能使用索引 |
| INDEX\_COLUMN like '%'||?  INDEX\_COLUMN like '%'||?||'%' | 含前导模糊查询的Like语法不能使用索引 |
| INDEX\_COLUMN is null | B-TREE索引里不保存字段为NULL值记录，因此IS NULL不能使用索引 |
| NUMBER\_INDEX\_COLUMN='12345'  CHAR\_INDEX\_COLUMN=12345 | Oracle在做数值比较时需要将两边的数据转换成同一种数据类型，如果两边数据类型不同时会对字段值隐式转换，相当于加了一层函数处理，所以不能使用索引。 |
| a.INDEX\_COLUMN=a.COLUMN\_1 | 给索引查询的值应是已知数据，不能是未知字段值。 |

##经过函数运算的字段要使用索引可以使用**函数索引**，这种需求建议与DBA沟通。

有时候我们会使用多个字段的组合索引，如果查询条件中第一个字段不能使用索引，那整个查询也不能使用索引。

如：我们company表建了一个id+name的组合索引，以下SQL是不能使用索引的

select \* from company where name=?

Oracle9i后引入了一种index skip scan的索引方式来解决类似的问题，但是通过index skip scan提高性能的条件比较特殊，使用不好反而性能会更差。

#### 什么字段需要索引

主键及外键通常都要有索引。

字段出现在查询条件中，并且查询条件可以使用索引。

语句执行频率高，一天会有几千次以上。

通过字段条件可筛选的记录集很小，那数据筛选比例是多少才适合。

这个没有固定值，需要根据表数据量来评估，以下是经验公式，可用于快速评估：

小表(记录数小于10000行的表)：筛选比例<10%。

大表：(筛选返回记录数)<(表总记录数\*单条记录长度)/10000/16。

单条记录长度≈字段平均内容长度之和+字段数\*2。

以下是一些字段是否需要建B-TREE索引的经验分类：

|  |  |  |
| --- | --- | --- |
|  | 字段类型 | 常见字段名 |
| 需要建索引的字段 | 主键 | ID,PK |
| 外键 | PRODUCT\_ID,COMPANY\_ID,MEMBER\_ID,  ORDER\_ID,TRADE\_ID,PAY\_ID |
| 有对象或身份标识意义字段 | HASH\_CODE,USERNAME,IDCARD\_NO,  EMAIL,TEL\_NO,IM\_NO |
| 索引慎用字段,需要进行数据分布及使用场景详细评估 | 日期 | GMT\_CREATE,GMT\_MODIFIED |
| 年月 | YEAR,MONTH |
| 状态标志 | PRODUCT\_STATUS,ORDER\_STATUS,  IS\_DELETE,VIP\_FLAG |
| 类型 | ORDER\_TYPE,IMAGE\_TYPE,GENDER,  CURRENCY\_TYPE |
| 区域 | COUNTRY,PROVINCE,CITY |
| 操作人员 | CREATOR,AUDITOR |
| 数值 | LEVEL,AMOUNT,SCORE |
| 长字符 | ADDRESS,COMPANY\_NAME,  SUMMARY,SUBJECT |
| 不适合建索引的字段 | 描述备注 | DESCRIPTION,REMARK,MEMO,DETAIL |
| 大字段 | FILE\_CONTENT,EMAIL\_CONTENT |

#### B-TREE索引

B-TREE索引也称为平衡树索引(Balance Tree)，它是一种按字段排好序的树形目录结构，主要用于提升查询性能和唯一约束支持。B-TREE索引的内容包括**根节点**、**分支节点**、**叶子节点**。

**叶子节点内容**：索引字段内容+表记录ROWID。

**根节点，分支节点内容**：当一个数据块中不能放下所有索引字段数据时，就会形成树形的根节点或分支节点，根节点与分支节点保存了索引树的顺序及各层级间的引用关系。



##### 复合索引(组合索引)

一个索引由多个字段组成，称为组合索引。

对于复合索引，在Where限制条件中出现所有被索引的列时，优化器会选择走索引；

对于复合索引，在Where限制条件中出现联合索引中前导的列，即创建联合索引时前导的列时，优化器会选择走索引；

对于创建复合索引时，应考虑被索引字段的优先顺序，应将经常作为限制条件的字段放在首位；重复值少，即高基数(high-cardinaltiy)的列往前靠，而重复值多的字段往后靠；

##### 反向索引

##### 函数索引

#### 位图索引

#### 全文索引

### 执行计划

#### 执行计划常用列字段解释

基数（Rows） ##Oracle估计的当前操作的返回结果集行数。

字节（Bytes） ##执行该步骤后返回的字节数。

耗费（COST）、CPU耗费 ##Oracle估计的该步骤的执行成本，用于说明SQL执行的代价，理论上越小越好（该值可能与实际有出入）。

时间（Time） ##Oracle估计的当前操作所需的时间。

#### 表访问的几种方式

##### TABLE ACCESS FULL（全表扫描）

Oracle会读取表中所有的行，并检查每一行是否满足SQL语句中的 Where 限制条件；

全表扫描时可以使用多块读（即一次I/O读取多块数据块）操作，提升吞吐量；

使用建议：数据量太大的表不建议使用全表扫描，除非本身需要取出的数据较多，占到表数据总量的 5% ~ 10% 或以上。

##### **TABLE ACCESS BY ROWID（通过ROWID的表存取）**

ROWID是由Oracle自动加在表中每行最后的一列伪列，既然是伪列，就说明表中并不会物理存储ROWID的值。

你可以像使用其它列一样使用它，只是不能对该列的值进行增、删、改操作；

一旦一行数据插入后，则其对应的ROWID在该行的生命周期内是唯一的，即使发生行迁移，该行的ROWID值也不变。

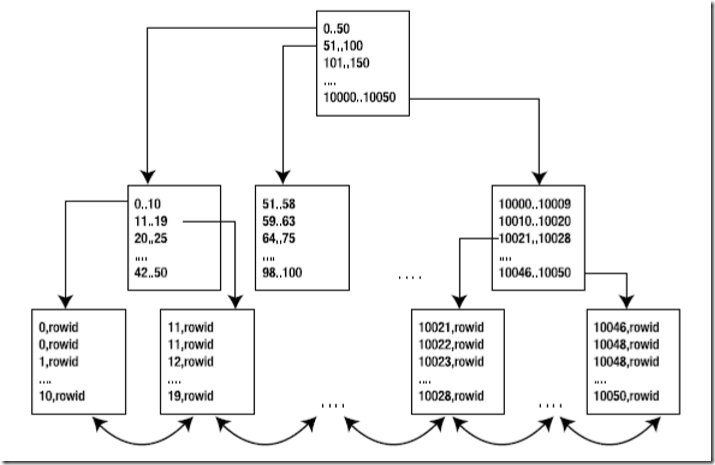
让我们再回到 TABLE ACCESS BY ROWID 来：

行的ROWID指出了该行所在的数据文件、数据块以及行在该块中的位置，所以通过ROWID可以快速定位到目标数据上，这也是Oracle中存取**单行**数据最快的方法。

##### **TABLE ACCESS BY INDEX SCAN（索引扫描）**

在索引块中，既存储每个索引的键值，也存储具有该键值的行的ROWID。

一个数字列上建索引后该索引可能的概念结构如下图：



索引扫描分为两步：

扫描索引得到对应的ROWID。

通过ROWID定位到具体的行读取数据。

#### 索引扫描类型

##### INDEX UNIQUE SCAN（索引唯一扫描）

针对唯一性索引（UNIQUE INDEX）的扫描，每次至多只返回一条记录；

表中某字段存在 UNIQUE、PRIMARY KEY 约束时，Oracle常实现唯一性扫描；

##### INDEX RANGE SCAN（索引范围扫描）

使用一个索引存取多行数据；

发生索引范围扫描的三种情况：

在唯一索引列上使用了范围操作符（如：>   <   <>   >=   <=   between）。

在组合索引上，只使用部分列进行查询（查询时必须包含前导列，否则会走全表扫描）。

对非唯一索引列上进行的任何查询。

##### INDEX FULL SCAN（索引全扫描）

进行全索引扫描时，查询出的数据都必须从索引中可以直接得到（注意全索引扫描只有在[CBO](#_CBO（Cost-Based_Optimization）_基于代价的优)模式下才有效）。

##### INDEX FAST FULL SCAN（索引快速扫描）

扫描索引中的所有数据块，与 [INDEX FULL SCAN](#_INDEX_FULL_SCAN（索引全扫描）) 类似，但是一个显著的区别是它不对查询出的数据进行排序（即数据不是以排序顺序被返回）。

##### INDEX SKIP SCAN（索引跳跃扫描）

Oracle 9i后提供，有时候[复合索引](#_组合索引(复合索引))的前导列（索引包含的第一列）没有在查询语句中出现，oralce也会使用该复合索引，这时候就使用的INDEX SKIP SCAN;

什么时候会触发 INDEX SKIP SCAN 呢？

前提条件：表有一个复合索引，且在查询时有除了前导列（索引中第一列）外的其他列作为条件，并且优化器模式为CBO时

当Oracle发现前导列的唯一值个数很少时，会将每个唯一值都作为常规扫描的入口，在此基础上做一次查找，最后合并这些查询；

例如：

假设表emp有ename（雇员名称）、job（职位名）、sex（性别）三个字段，并且建立了如 create index idx\_emp on emp (sex, ename, job) 的复合索引；

因为性别只有 '男' 和 '女' 两个值，所以为了提高索引的利用率，Oracle可将这个复合索引拆成 ('男', ename, job)，('女', ename, job) 这两个复合索引；

当查询 select \* from emp where job = 'Programmer' 时，该查询发出后：

Oracle先进入sex为'男'的入口，这时候使用到了 ('男', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

再进入sex为'女'的入口，这时候使用到了 ('女', ename, job) 这条复合索引，查找 job = 'Programmer' 的条目；

最后合并查询到的来自两个入口的结果集。

### 优化器

#### RBO（Rule-Based Optimization） 基于规则的优化器

RBO有严格的使用规则，只要按照这套规则去写SQL语句，无论数据表中的内容怎样，也不会影响到你的执行计划；

换句话说，RBO对数据“不敏感”，它要求SQL编写人员必须要了解各项细则；

RBO一直沿用至ORACLE 9i，从ORACLE 10g开始，RBO已经彻底被抛弃。

#### CBO（Cost-Based Optimization） 基于代价的优化器

CBO是一种比RBO更加合理、可靠的优化器，在ORACLE 10g中完全取代RBO；

CBO通过计算各种可能的执行计划的“代价”，即COST，从中选用COST最低的执行方案作为实际运行方案；

它依赖数据库对象的统计信息，统计信息的准确与否会影响CBO做出最优的选择，也就是对数据“敏感”。

## MySQL