# 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" \t "_blank)(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。

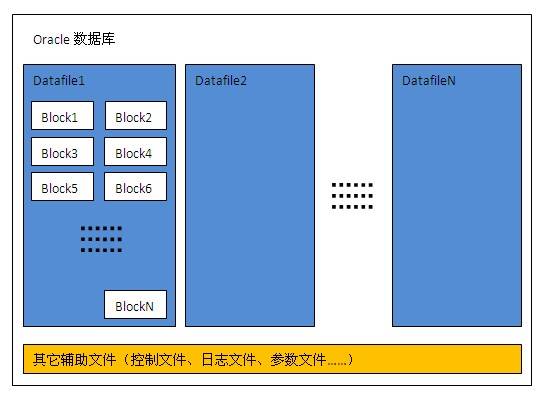
#### 尽量避免大事务操作

尽量避免大事务操作，提高系统并发能力。

## Oracle

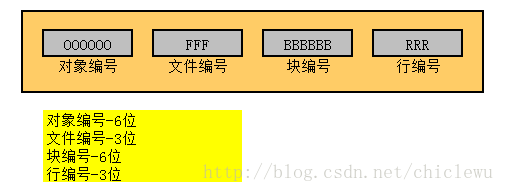
### 数据块(Block)

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" \t "_blank" \o "Oracle知识库)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。

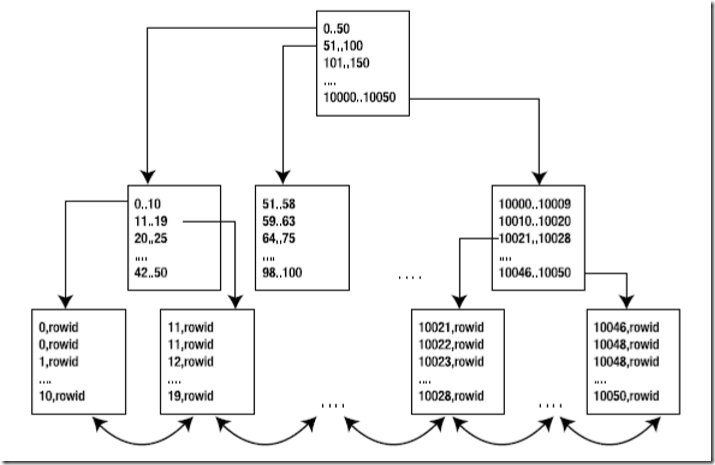
让我们再回到 TABLE ACCESS BY ROWID 来：

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

### 数据库引擎

#### MyISAM

MySql的默认引擎，**不支持事务，行级锁和外键**。

当执行Insert插入和Update**更新语句时**，即执行写操作的时候需要**锁定整个表**，**导致效率降低**。

不过和Innodb不同的是，MyIASM引擎是**保存了表的行数**，于是当进行Select count(\*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的**读操作远远多于写操作**时，并且**不需要事务的支持**，可以**选MyIASM作为数据库引擎**。

##### 数据库表文件

**.MYD**文件：即MY Data，表数据文件

**.MYI**文件：即MY Index，索引文件

**.log**文件：日志文件

#### InnoDB

Innodb引擎提供了**对数据库ACID事务的支持**。

**提供行级锁和外键的约束**。它的设计的目标就是处理大数据容量的数据库系统。它本身实际上是基于Mysql后台的完整的系统。Mysql运行的时候，Innodb会在内存中**建立缓冲池，用于缓冲数据和索引**。但是，该引擎是**不支持全文搜索**的。同时，启动也比较的慢，它**不保存表的行数**。当进行Select count(\*) from table指令的时候，需要进行扫描全表。所以当需要使用数据库的事务时，该引擎就是首选。由于**锁的粒度小**，**写操作不会锁定全表**。所以在并发度较高的场景下使用会提升效率的。

##### 数据库表文件

InnoDB采用表空间（tablespace）来管理数据，存储表数据和索引，

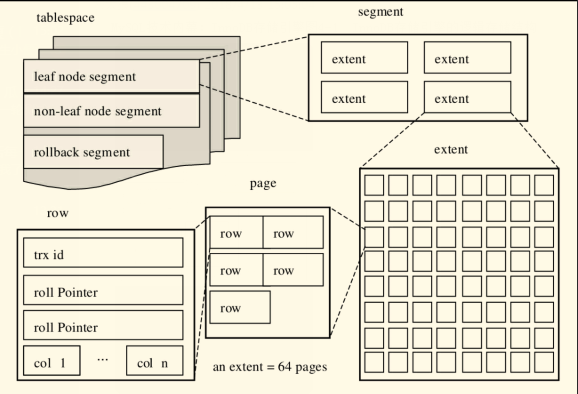
InnoDB数据库文件（即InnoDB文件集，ib-file set）：

**ibdata1**、**ibdata2**等：系统表空间文件，存储InnoDB系统信息和用户数据库表数据和索引，所有表共用

**.ibd**文件：单表表空间文件，**每个表使用一个表空间文件**（file per table），存放用户数据库表数据和索引

日志文件： ib\_logfile1、ib\_logfile2

##### 逻辑存储结构



1. **表空间（tablespace）**

InnoDB存储引擎的逻辑存储结构和Oracle大致相同，所有数据都被逻辑地存放在一个空间中，我们称之为**表空间（tablespace）**，

表空间又由**段（segment）**、**区（extent）**、**页（page）**组成。页在一些文档中有时也称为**块（block）。**

默认情况下InnoDB存储引擎有一个共享表空间ibdata1，即所有数据都放在这个表空间内。如果我们启用了参数**innodb\_file\_per\_table**，则每张表内的数据可以单独放到一个表空间内。

**mysql>show variables like 'innodb\_file\_table'\G;**

**mysql>system ls -lh /usr/local/mysql/data/ib\***

对于启用了innodb\_file\_per\_table的参数选项，需要注意的是，每张表的表空间内存放的只是数据、索引和插入缓冲，其他类的数据，如撤销（Undo）信息、系统事务信息、二次写缓冲（double write buffer）等还是存放在原来的共享表空间内。这也就说明了另一个问题：即使在启用了参数innodb\_file\_per\_table之后，共享表空间还是会不断地增加其大小。

1. **段（segment）**
2. **区（extent）**
3. **页（page）**

##### 插入缓冲（insert buffer)

插入缓冲,并不是缓存的一部分,而是物理页,对于非聚集索引的插入或更新操作,不是每一次直接插入索引页.而是先判断插入的非聚集索引页是否在缓冲池中.如果在,则直接插入,如果不再,则先放入一个插入缓冲区中.然后再以一定的频率执行插入缓冲和非聚集索引页子节点的合并操作。

使用条件:**非聚集索引**,**非唯一**。

##### 二次写(double write)

##### 自适应哈希索引(ahi)

##### 预读(read ahead)

#### MyISAM与InnoDB的区别

|  |  |  |
| --- | --- | --- |
|  | MyISAM | InnoDB |
| 事务 | 不支持 | **支持** |
| 行级锁 | 不支持 | **支持** |
| MVCC | 不支持 | **支持** |
| 外键 | 不支持 | **支持** |
| 多种行格式 | 不支持 | **支持** |
| 全文索引 | **支持** | 不支持 |
| 拷贝表文件到另一台机器 | **支持** | 不支持 |
| 表类型 | 堆表 | 索引组织表 |

### 索引

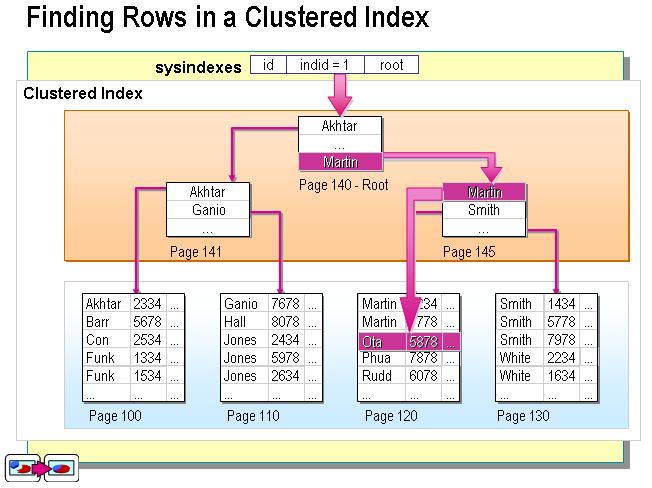
#### 聚集索引(聚簇索引)

##### 聚集索引

该索引中键值的逻辑顺序决定了表中相应行的物理顺序。**聚集索引**的叶节点就是数据节点。一个表只能建一个**聚集索引。**

innodb的聚集索引实际上是在同一个结构中保存了btree索引和数据行。

Innodb的存储索引是基于B+tree，理所当然，聚集索引也是基于B+tree。与非聚集索引的区别则是，聚集索引既存储了索引，也存储了行值。当一个表有一个聚集索引，它的数据是存储在索引的叶子页（leaf pages）。因此innodb也能理解为基于索引的表。



确定物理顺序

**聚集索引**确定表中数据的物理顺序。**聚集索引**类似于电话簿，后者按姓氏排列数据。由于**聚集索引**规定数据在表中的物理存储顺序，因此一个表只能包含一个聚集索引。但该索引可以包含多个列（组合索引），就像电话簿按姓氏和名字进行组织一样。

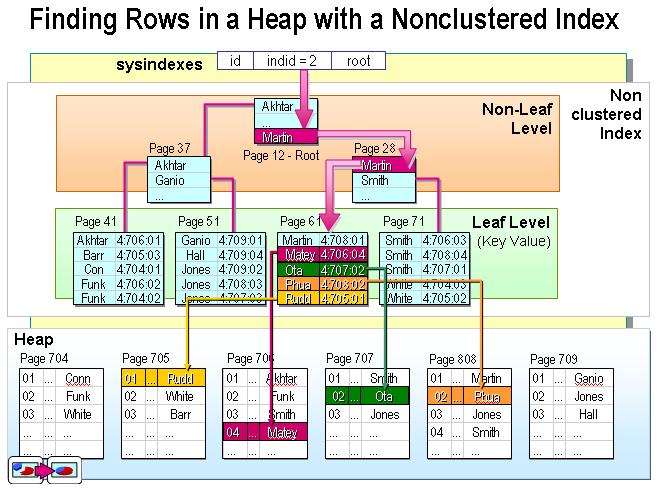
搜索范围值特别有效

**聚集索引**对于那些经常要搜索范围值的列特别有效。使用**聚集索引**找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。例如，如果应用程序执行 的一个查询经常检索某一日期范围内的记录，则使用聚集索引可以迅速找到包含开始日期的行，然后检索表中所有相邻的行，直到到达结束日期。这样有助于提高此 类查询的性能。同样，如果对从表中检索的数据进行排序时经常要用到某一列，则可以将该表在该列上聚集（物理排序），避免每次查询该列时都进行排序，从而节 省成本。

当索引值唯一时，使用聚集索引查找特定的行也很有效率。例如，使用唯一雇员 ID 列 emp\_id 查找特定雇员的最快速的方法，是在 emp\_id 列上创建聚集索引或 PRIMARY KEY 约束。

##### 非聚集索引

该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。



##### Innodb如何选择一个聚集索引

对于Innodb，主键毫无疑问是一个聚集索引。但是当一个表没有主键，或者没有一个索引，Innodb会如何处理呢。请看如下规则

1. 如果一个主键被定义了，那么这个主键就是作为**聚集索引。**
2. 如果没有主键被定义，那么该表的第一个**唯一非空索引**被作为**聚集索引。**
3. 如果没有主键也没有合适的唯一索引，那么innodb内部会生成**一个隐藏的主键**作为聚集索引，这个隐藏的主键是一个6个字节的列，改列的值会随着数据的插入自增。

还有一个需要注意的是：

**次级索引**的叶子节点并不存储行数据的物理地址。而是存储的该行的主键值。

所以：一**次级索引**包含了两次查找。一次是查找**次级索引**自身。然后查找主键（聚集索引）。

##### 何时使用聚集索引或非聚集索引

下面的表总结了何时使用聚集索引或非聚集索引（很重要）：

|  |  |  |
| --- | --- | --- |
| 动作描述 | 使用聚集索引 | 使用非聚集索引 |
| 列经常被分组排序 | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 一个或极少不同值 | 不应 | 不应 |
| 小数目的不同值（distinct 小） | 应 | 不应 |
| 大数目的不同值（distinct 大） | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
| 频繁修改索引列 | 不应 | 应 |

##### 聚集索引的优点

1. 区间查找非常快速。
2. 排序非常快。

如果一个表的某个字段经常被用来排序，如果该字段在聚集索引列，则排序的时间就是查询的时间。同时，因为B-Tree同时保存了索引和数据值。因此通过聚集索引的检索会快于一个非聚集索引的检索。

##### 聚集索引的缺点

1. 如果一个很大的聚集索引被定义。那么所有的次级索引也会显而易见的大，因为次级索引存储了聚集索引的key。因为数据的存储方式，次级索引的查找需要两次。
2. 聚集索引的列的值被更改后，聚集索引会对数据重排，因此主键被更改的开销是昂贵的。
3. 插入可能比较慢，如果数据不是按照主键的顺序插入。因此我们判断插入的数据的效率决定于插入的顺序。在innodb的表中，插入行按照主键的顺序插入式是最快的方式。

#### 次级索引（secondary index）

除了聚簇索引之外的其他索引类型都属于次级索引。

在innodb，每个次级索引都存储的主键key。因为innodb的数据结构决定了，次级索引存储的是主键的值。次级索引的叶子节点并不存储行数据的物理地址。而是存储的该行的主键值。因此，换句话说，主键扮演了行数据的指针。

一次级索引包含了两次查找。一次是查找次级索引自身。然后查找主键（聚集索引）。

#### 索引使用的误区

##### 主键就是聚集索引(误区)

这种想法笔者认为是极端错误的，是对聚集索引的一种浪费。虽然SQL SERVER默认是在主键上建立聚集索引的。

通常，我们会在每个表中都建立一个ID列，以区分每条数据，并且这个ID列是自动增大的，步长一般为1。我们的这个办公自动化的实例中的列Gid就是如此。此时，如果我们将这个列设为主键，SQL SERVER会将此列默认为聚集索引。这样做有好处，就是可以让您的数据在数据库中按照ID进行物理排序，但笔者认为这样做意义不大。

显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引的规则，这使得聚集索引变得更加珍贵。

从我们前面谈到的聚集索引的定义我们可以看出，使用聚集索引的最大好处就是能够根据查询要求，迅速缩小查询范围，避免全表扫描。在实际应用中，因为 ID号是自动生成的，我们并不知道每条记录的ID号，所以我们很难在实践中用ID号来进行查询。这就使让ID号这个主键作为聚集索引成为一种资源浪费。其次，让每个ID号都不同的字段作为聚集索引也不符合“大数目的不同值情况下不应建立聚合索引”规则；当然，这种情况只是针对用户经常修改记录内容，特别是索引项的时候会负作用，但对于查询速度并没有影响。

在办公自动化系统中，无论是系统首页显示的需要用户签收的文件、会议还是用户进行文件查询等任何情况下进行数据查询都离不开字段的是“日期”还有用户本身的“用户名”。

通常，办公自动化的首页会显示每个用户尚未签收的文件或会议。虽然我们的where语句可以仅仅限制当前用户尚未签收的情况，但如果您的系统已建立了很长时间，并且数据量很大，那么，每次每个用户打开首页的时候都进行一次全表扫描，这样做意义是不大的，绝大多数的用户1个月前的文件都已经浏览过了，这样做只能徒增数据库的开销而已。事实上，我们完全可以让用户打开系统首页时，数据库仅仅查询这个用户近3个月来未阅览的文件，通过“日期”这个字段来限制表扫描，提高查询速度。如果您的办公自动化系统已经建立的2年，那么您的首页显示速度理论上将是原来速度8倍，甚至更快。

##### 只要建立索引就能显著提高查询速度(误区)

  （1）仅在主键上建立聚集索引，并且不划分时间段：

Select gid,fariqi,neibuyonghu,title from tgongwen

用时：128470毫秒（即：128秒）

（2）在主键上建立聚集索引，在fariq上建立非聚集索引：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：53763毫秒（54秒）

（3）将聚合索引建立在日期列（fariqi）上：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：2423毫秒（2秒）

事实上，我们可以发现上面的例子中，第2、3条语句完全相同，且建立索引的字段也相同；不同的仅是前者在fariqi字段上建立的是非聚合索引，后者在此字段上建立的是聚合索引，但查询速度却有着天壤之别。所以，并非是在任何字段上简单地建立索引就能提高查询速度。

从建表的语句中，我们可以看到这个有着1000万数据的表中fariqi字段有5003个不同记录。在此字段上建立聚合索引是再合适不过了。在现实中，我们每天都会发几个文件，这几个文件的发文日期就相同，这完全符合建立聚集索引要求的：“既不能绝大多数都相同，又不能只有极少数相同”的规则。由此看来，我们建立“适当”的聚合索引对于我们提高查询速度是非常重要的。

##### 把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度

如果仅用聚集索引的起始列作为查询条件和同时用到复合聚集索引的全部列的查询速度是几乎一样的，甚至比用上全部的复合索引列还要略快（在查询结果集数目一样的情况下）；而如果仅用复合聚集索引的非起始列作为查询条件的话。同时，请记住：无论您是否经常使用聚合索引的其他列，但其前导列一定要是使用最频繁的列。

### 数据库优化

#### 开启慢查询日志

查看所有日志状态： show variables like '%quer%';

查看慢查询状态：show variables like 'show%'

vim /etc/my.cnf

[mysqld]

slow-query-log = on ##开启慢查询功能

slow\_query\_log\_file = /usr/local/mysql/data/slow-query.log ##慢查询日志存放路径与名称

long\_query\_time = 5 ##查询时间超过5s的查询语句

log-queries-not-using-indexes = on ##列出没有使用索引的查询语句

#### ****explain（执行计划）****

explain返回各列的含义

**extra**列需要注意的返回值

**Using filesort**:看到这个的时候，查询就需要优化了。MYSQL需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行

**Using temporary**看到这个的时候，查询需要优化了。这里，MYSQL需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上

**table**:显示这一行的数据是关于哪张表的。

**type**:这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq\_reg、ref、range、index 和ALL。

**possible\_keys**:显示可能应用在这张表中的索引。如果为空，没有可能的索引。

**key**:实际使用的索引。如果为NULL,则没有使用索引。

**keyjen**:使用的索引的长度。在不损失精确性的情况下，长度越短越好。

**ref**:显示索引的哪一列被使用了，如果可能的话，是一个常数。

**rows**: MYSQL认为必须检查的用来返回请求数据的行数。

#### sql优化

##### ****子查询的优化****

通常情况下把子查询优化为join查询，但在优化的时候需要注意关联建是否有一对多的关系，要特别注意重复数据