索引原理与表设计

# 索引原理

索引是有效使用数据库的基础，但你的数据量很小的时候，或许通过扫描整表来存取数据的性能还能接受，但当数据量极大时，当访问量极大时，就一定需要通过索引的辅助才能有效地存取数据。一般索引建立的好坏是性能好坏的成功关键。

## InnoDb数据与索引存储细节

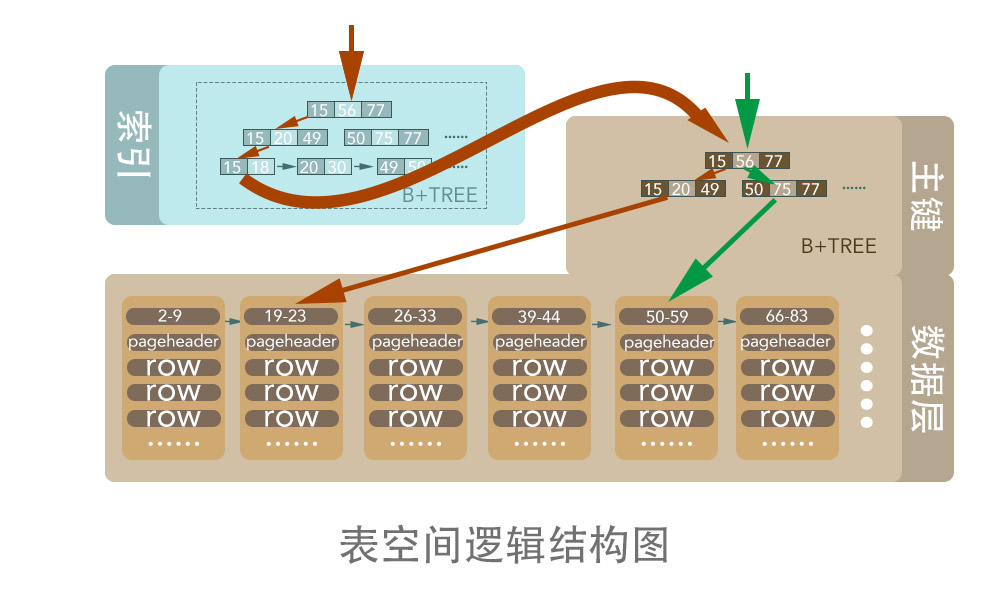
使用InnoDb作为数据引擎的Mysql和有聚集索引的SqlServer的数据存储结构有点类似，虽然在物理层面，他们都存储在Page上，但在逻辑上面，我们可以把数据分为三块：数据区域，索引区域，主键区域，他们通过主键的值作为关联，配合工作。默认配置下，一个Page的大小为16K。

一个表数据空间中的索引数据区域中有很多索引，每一个索引都是一颗B+Tree，在索引的B+Tree中索引的值作为B+Tree的节点的Key，数据主键作为节点的Value。

在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶节点数据域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。

可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

表数据都以Row的形式放在一个一个大小为16K的Page中，在每个数据Page中有都页头信息和一行一行的数据。其中页头信息中主要放置的是这一页数据中的所有主键值和其对应的OFFSET，便于通过主键能迅速找到其对应的数据位置。



## 索引优化检索的原理

索引是数据库的灵魂，如果没有索引，数据库也就是一堆文本文件，存在的意义并不大。索引能让数据库成几何倍数地提高检索效率。使用Innodb作为数据引擎的Mysql数据库的索引分为聚集索引（也就是主键）和普通索引。上节我们已经讲解了这两种索引的存储结构，现在我们仔细讲解下索引是如何工作的。

聚集索引和普通索引都有可能由多个字段组成，我们又称这种索引为复合索引，1.2.3将为大家解析这种索引的性能情况。

### 聚集索引

从上节我们知晓，Innodb的所有数据是按照聚集索引排序的，聚集索引这种存储方式使得按主键的搜索十分高效，如果我们SQL语句的选择条件中有聚集索引，数据库会优先使用聚集索引来进行检索工作。

数据库引擎根据条件中的主键的值，迅速在B+Tree中找到主键对应的叶节点，然后把叶节点所在的Page数据库读取到内存，返回给用户，如上图绿色线条的流向。下面我们来运行一条SQL，从数据库的执行情况分析一下：

**select \* from UP\_User where userId = 10000094;**

......

**# Query\_time: 0.000399** Lock\_time: 0.000101 Rows\_sent: 1 Rows\_examined: 1 Rows\_affected: 0

**# Bytes\_sent: 803** Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# InnoDB\_trx\_id: 1D4D

# QC\_Hit: **No Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# Filesort: No Filesort\_on\_disk: No Merge\_passes: 0

# InnoDB\_IO\_r\_ops: 0 InnoDB\_IO\_r\_bytes: 0 InnoDB\_IO\_r\_wait: 0.000000

# InnoDB\_rec\_lock\_wait: 0.000000 InnoDB\_queue\_wait: 0.000000

**# InnoDB\_pages\_distinct: 2**

SET timestamp=1451104535;

select \* from UP\_User where userId = 10000094;

我们可以看到，数据库读从磁盘取了两个Page就把809Bytes的数据提取出来返回给客户端。

下面我们试一下，如果试一下选取条件没有包含主键和索引的情况：

**select \* from `UP\_User` where bigPortrait = '5F29E883BFA8903B';**

**# Query\_time: 0.002869** Lock\_time: 0.000094 Rows\_sent: 1 Rows\_examined: 1816 Rows\_affected: 0

**# Bytes\_sent: 792** Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: Yes** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 25**

可以看到如果使用主键作为检索条件，检索时间花了0.3ms，只读取了两个Page，而不使用主键作为检索条件，检索时间花了2.8ms，读取了25个Page，全局扫描才把这条记录给找出来。这还是一个只有1000多行的表，如果更大的数据量，对比更加强烈。

对于这两个Page，一个是主键B+Tree的数据，一个是10000094这条数据所在的数据页。

### 普通索引

我们用普通索引作为检索条件来搜索数据需要检索两遍索引：首先检索普通索引获得主键，然后用主键到主索引中检索获得记录。如上图红色线条的流向。

下面我用一个例子来看看数据库的表现：

**select \* from UP\_User where userName = 'fred';**

**# Query\_time: 0.000400** Lock\_time: 0.000101 Rows\_sent: 1 Rows\_examined: 1 Rows\_affected: 0

**# Bytes\_sent: 803** Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 4**

我们可以看到数据库用了0.4ms来检索这条数据，读取了4个page，比用主键作为检索条件多用了0.1ms，多读取了两个Page，而这两个Page就是userName这个普通索引的B+Tree所在的数据页。

### 复合索引

聚集索引和普通索引都有可能由多个字段组成，多个字段组成的索引的叶节点的Key由多个字段的值按照顺序拼接而成。这种索引的存储结构是这样的，首先按照第一个字段建立一棵B+Tree，叶节点的Key是第一个字段的值，叶节点的value又是一棵小的B+Tree，逐级递减。对于这样的索引，用排在第一的字段作为检索条件才有效提高检索效率。排在后面的字段只能在排在他前面的字段都在检索条件中的时候才能起辅助效果。下面我们用例子来说明这种情况。

我们在UP\_User表上建立一个用来测试的复合索引，建立在 (`nickname`,`regTime`)两个字段上，下面我们测试下检索性能：

**select \* from UP\_User where nickName='fredlong';**

**# Query\_time: 0.000443** Lock\_time: 0.000101 Rows\_sent: 1 Rows\_examined: 1 Rows\_affected: 0

# Bytes\_sent: 778 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 4**

我们看到索引起作用了，和普通索引的效果一样都用了0.43ms，读取了四个Page就完成了任务。

**select \* from UP\_User where regTime = '2015-04-27 09:53:02';**

**# Query\_time: 0.007076** Lock\_time: 0.000286 Rows\_sent: 1 Rows\_examined: 1816 Rows\_affected: 0

# Bytes\_sent: 803 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: Yes** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 26**

从这次选择的执行情况来看，虽然regTime在刚才建立的复合索引中，还是做了全局扫描。因为这个复合索引排在regTime字段前面的nickname字段没有出现在选择条件中，所以这个索引我们没用用到。

那么我们什么情况下会用到复合索引呢。我一般在两种情况下会用到：

1. 需要复合索引来排重的时候。
2. 用索引的第一个字段选取出来的结果不够精准，需要第二个字段做进一步的性能优化。

我基本上没有建立过三个以上的字段做复合索引，如果出现这种情况，我觉的你的表设计可能出现了大而全的问题，需要在表设计层面调优，而不是通过增加复杂的索引调优。所有复杂的东西都是大概率有问题的。

## 批量选择的效率

我们业务经常这样的诉求，选择某个用户发送的所有消息，这个帖子所有回复内容，所有昨天注册的用户的UserId等。这样的诉求需要从数据库中取出一批数据，而不是一条数据，数据库对这种请求的处理逻辑稍微复杂一点，下面我们来分析一下各种情况。

### 根据主键批量检索数据

我们有一张表PW\_Like，专门存储Feed的所有赞（Like），该表使用feedId和userId做联合主键，其中feedId为排在第一位的字段。该表一共有19个Page的数据。下面我们选取feedId为11593所有的赞：

**select \* from PW\_Like where feedId = 11593;**

**# Query\_time: 0.000478** Lock\_time: 0.000084 **Rows\_sent: 58** Rows\_examined: 58 Rows\_affected: 0

# Bytes\_sent: 865 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 2**

我们花了0.47ms取出了58条数据，但一共只读取了2个Page的数据，其中一个Page还是这个表的主键。说明这58条数据都存储在同一个Page中。这种情况，对于数据库来说，是效率最高的情况。

### 根据普通索引批量检索数据

还是刚才那个表，我们除了主键以外，还在userId上建立了索引，因为有时候需要查询某个用户点过的赞。那么我们来看看只通过索引，不通过主键来检索批量数据时候，数据库的效率情况。

**select \* from PW\_Like where userId = 80000402;**

**# Query\_time: 0.002892** Lock\_time: 0.000062 **Rows\_sent: 27** Rows\_examined: 27 Rows\_affected: 0

# Bytes\_sent: 399 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 15**

我们可以看到的结果是，虽然我们只取出了27条数据，但是我们读取了15个数据Page，花了2.8毫秒，虽然没有进行全局扫描，但基本上也把一般的数据块读取出来了。因为PW\_Like中的数据由于是按照feedId物理排序的，所以这27条数据分别分布在13个Page中（有两个Page是索引和主键），所以数据库需要把这13个Page全部从磁盘中读取出来，哪怕某一个Page（16K）上只有一条数据（15Bytes），也需要把这个数据Page读出来才能取出所有的目标Row。

通过普通索引来检索批量数据，效率明显比通过主键来检索要低得多。因为数据是分散的，所以需要分散地读取数据Page进行拼接才能完成任务。但是索引对主键而言还是非常有必要的补充，比如上面这个例子，当用户量达到100万的时候，检索某一个用户点的所有的赞的成本也只是大概读取15个Page，花2ms左右。

### 选取一段时间范围内的数据

选取一定范围内的数据是我们经常要遇到的问题。在海量数据的表中检索一定范围内的数据，很容易引起性能问题。我们遇到的最常见的需求有以下几种：

1. **选取一段时间内注册的用户信息**  
   这种时候，时间肯定不会是用户表的主键，如果直接用时间作为选择条件来检索，效率会非常差，如何解决这种问题呢？我采取的办法是，把注册时间作为用户表的索引，每次先把需要检索的时间的两端的userId都差出来，然后用这两个userId做选择条件做第三次查询就是我们想要的数据了。这三次检索我们只需要读取大约10个Page就能解决问题。这种方法看起来很麻烦，但是是表的数据量达到亿级的时候的唯一解决方案。
2. **选取一段时间内的日志**  
   日志表是我们最常见的表，如何设计好是经常聊到的话题。日志表之所以不好设计是因为大家都希望用时间作为主键，这样检索一段时间内的日志将非常方便。但是用时间作为主键有个非常大的弊端，当日志插入速度很快的时候，就会出现因为主键重复而引起冲突。  
   对于这种情况，我一般把日志生成时间和一个自增的Id作为日志表的联合主键，把时间作为第一个字段，这样即避免了日志插入过快引起的主键唯一性冲突，又能便捷地根据时间做检索工作，非常方便。下面是这种日志表的检索的例子，大家可以看到性能非常好。还有就是日志表最好是每天一个表，这样能更便利地管理和检索。  
     
   **select \* from log\_test where logTime > "2015-12-27 11:53:05" and logTime < "2015-12-27 12:03:05";**

**# Query\_time: 0.001158** Lock\_time: 0.000084 **Rows\_sent: 599** Rows\_examined: 599 Rows\_affected: 0

# Bytes\_sent: 4347 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

**# InnoDB\_pages\_distinct: 3**

## Select结果排序原理

我们在批量检索数据的时候，对选择的结果在大多数情况下，都需要数据是排好序的。排序的性能也是日常需要注意到的，下面我们分三种情况来分析下数据库是如何排序的。

### ORDER BY 主键

对于ORDER BY 主键这种情况，数据库是非常乐于见到的，基本上不会有额外性能的损耗，因为数据本来就是按照主键顺序存储的，取出来直接返回即可。

下面的所有关于排序的例子是在UP\_MessageHistory表上做的实验，这个表一共有195个page，35417行数据，主键建立在字段id上，在sendUserId和destUserId上都建立了索引。

首先我们先做一个没有排序的检索：

**select \* from `CU`.`UP\_MessageHistory` where sendUserId = 92;**

**# Query\_time: 0.016135** Lock\_time: 0.000084 Rows\_sent: 3572 Rows\_examined: 3572 Rows\_affected: 0

# Bytes\_sent: 95600 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 125

然后我们在相同的条件下对id进行排序：

**select \* from `CU`.`UP\_MessageHistory` where sendUserId = 92 order by id;**

**# Query\_time: 0.016259** Lock\_time: 0.000086 Rows\_sent: 3572 Rows\_examined: 3572 Rows\_affected: 0

# Bytes\_sent: 95600 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 125

从上面的数据可以看出，性能和没有加order by差不多，基本没有额外的性能损耗。接下来我们对索引进行排序：

**select \* from `CU`.`UP\_MessageHistory` where sendUserId = 92 order by destUserId;**

**# Query\_time: 0.018107** Lock\_time: 0.000083 Rows\_sent: 3572 Rows\_examined: 7144 Rows\_affected: 0

# Bytes\_sent: 103123 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 125

接下来我们用普通字符串字段做排序再看看：

**select \* from `CU`.`UP\_MessageHistory` where sendUserId = 92 order by content;**

**# Query\_time: 0.023611** Lock\_time: 0.000085 Rows\_sent: 3572 Rows\_examined: 7144 Rows\_affected: 0

# Bytes\_sent: 105214 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 125

然后我们再用普通的数字类型字段排序看看情况：

**select \* from `CU`.`UP\_MessageHistory` where sendUserId = 92 order by sentTime;**

**# Query\_time: 0.018522** Lock\_time: 0.000107 Rows\_sent: 3572 Rows\_examined: 7144 Rows\_affected: 0

# Bytes\_sent: 95709 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 125

针对以上的实验结果，我们可以得出以下结论：

1. **针对主键做排序操作不会有性能损耗；**
2. **针对不在选择条件中的索引字段做排序操作，索引不会起优化排序的作用；**
3. **针对数值类型字段排序会比针对字符串类型字段排序的效率要高很多。**

下面我们再研究下用选择条件中的索引字段排序，数据库是否会优化排序算法，我们任然用UP\_User表来研究。

**select \* from UP\_User where score > 10000;**

**# Query\_time: 0.001470** Lock\_time: 0.000130 Rows\_sent: 122 Rows\_examined: 122 Rows\_affected: 0

# Bytes\_sent: 9559 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 17

然后我们用选择条件中的索引字段做排序：

**select \* from UP\_User where score > 10000 order by score**

**# Query\_time: 0.001407** Lock\_time: 0.000087 Rows\_sent: 122 Rows\_examined: 122 Rows\_affected: 0

# Bytes\_sent: 9559 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 17

然后我们用选择条件中的非索引数值字段做排序：

**select \* from UP\_User where score > 10000 ORDER BY `securityQuestion`**

**# Query\_time: 0.002017** Lock\_time: 0.000104 Rows\_sent: 122 Rows\_examined: 244 Rows\_affected: 0

# Bytes\_sent: 9657 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 17

从上面三条查询语句的执行时间来分析，使用索引字段排序和不排序花的时间差不多，比使用普通字段排序花的时间少一些，因此我们可以得出第四条结论：

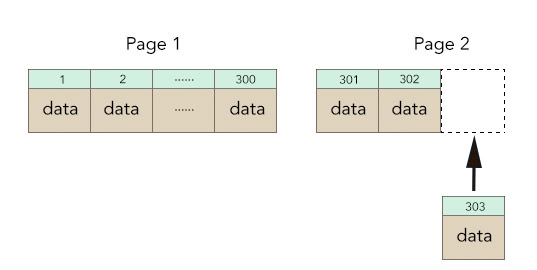
1. **针对在选择条件中的索引字段做排序操作，索引会起优化排序的作用。**

## 索引维护

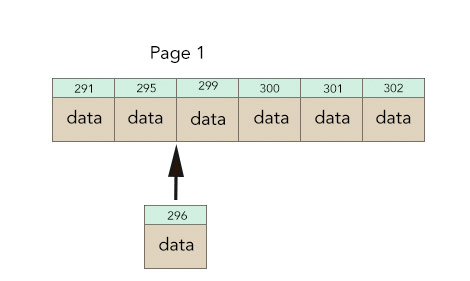
在前面我们可以看到所有的主键和索引都是排好序的，那么排序这件事情就需要销号资源，每次有新的数据插入，或者老的数据的数值发生变更，排序就需要调整，这里面是需要损耗性能的，下面我们分析一下。

自增型字段作为主键时，数据库对主键的维护成本非常低：

1. 每次新增加的值都是一个最大的值，追加到最后即可，其他数据不需要挪动；
2. 这种数据一般不做修改。



使用业务型字段作为主键时，主键维护成本会比较高。每次生成的新数据都有可能需要挪动其他数据的位置。



因为innodb主键和数据是在放在一块的，每次挪动主键，也需要挪动数据，维护的成本会比较搞，对于需要频繁写入的表，不建议使用业务字段作为主键的。

由于主键是所有索引的叶节点的值，也是数据排序的依据，如果主键的值被修改，那么需要修改所有相关索引，并且需要修改整个主键B+Tree的排序，损耗会非常大。避免频繁更新主键可以避免以上提到的问题。

**update UP\_User set userId = 100000945 where userId = 10000094;**

**# Query\_time: 0.010916** Lock\_time: 0.000201 Rows\_sent: 0 Rows\_examined: 1 Rows\_affected: 1

# Bytes\_sent: 59 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: No Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 11

从上面的数据可以看出来，索然SQL语句只修改了一条数据，却影响了11个Page。

相对于主键而已，索引就轻很多，它的叶节点的值是主键，很轻，维护起来成本比较低。但也不建议为一个表建立过多索引。维护一个索引成本低，维护8个就不一定低了，这种事需要均衡地对待。

## 设计索引原则

索引其实是一把双刃剑，用好了事半功倍，没用好，事倍功半。

主键的字段无特殊情况，一定要使用数值类型的，排序时占用计算资源少，存储时占用空间也少。若主键的字段值很大，则整个数据表的各种索引也会变得没有效率，因为所有的索引的叶节点的值都是主键。

并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

# 表设计原则

从经历过的大大小小的项目中提取出自己的数据表设计原则与大家分享，具有一定的个人风格，供大家参考。

## 数据库尽量作为数据仓库

数据库尽量作为数据仓库，业务逻辑在应用中实现，而不是通过复杂的SQL语句或存储过程实现业务逻辑。

在数据库中实现业务逻辑会增加业务应用与数据库之间的耦合性，也会增加数据的压力。数据库应该只是个落地的仓库，用的时候取出来，用完了放回去，而不是在仓库里面对数据进行加工。

对于事务控制这块，数据库中的逻辑处理有一定优势，但我一般选择利用redis的全局性来锁定某条数据来完成事物控制。

## 一张表对应且只对应一个逻辑主体

一张表只负责落地一个业务逻辑，干净利落。

## 尽量避免字段中允许NULL值

在MySQL中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值。

## 预留一个flag字段作为扩展字段

我习惯给一些有可能出现特定逻辑需求的表加上一个int32型flag字段，这样很多bool逻辑判断都可以放到这个这个flag字段的不同的位内来实现。比如我在CU\_Sticker（表情）表内冗余了字段flag，之后在这个flag上增加了以下的零散逻辑判断：

RESENDENT(0x1),//是不是作为本地表情

RECOMMAND(0x2),//是不是推荐的表情

IWATCHAWARD(0x4);//用户安装iwatch版本之后，是不是赠送此表情

## 尽量使用数字作为字段类型

越小的数据类型通常更好：越小的数据类型通常在磁盘、内存和CPU缓存中都需要更少的空间，处理起来更快。数据量多的表，枚举类型只允许存储值，不允许存储枚举字符串。

理论上，除了日志表，其他表的内容除了用户产生的内容，表内数据应该都是一些数字，日期。

# SQL语句撰写原则

数据库设计好了，如何合理地写SQL语句去访问数据库也有许多需要注意的地方。

## 不要对where后的字段做运算

对Where后面字段做运算会让索引失效，引发全表扫描。

**select \* from UP\_User where userId+1=10000095**

**# Query\_time: 0.002880** Lock\_time: 0.000125 Rows\_sent: 1 Rows\_examined: 1876 Rows\_affected: 0

# Bytes\_sent: 818 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: **No Full\_scan: Yes** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 26

## 不要对where后的字段使用函数

对Where后面字段使用函数会让索引失效，引发全表扫描。

**select \* from UP\_User where SUBSTRING(userName , 1 , 1) = 'F'**

**# Query\_time: 0.007103** Lock\_time: 0.003494 Rows\_sent: 73 Rows\_examined: 1876 Rows\_affected: 0

# Bytes\_sent: 6160 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: Yes** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 26

这种情况我们可以用下面这种写法替代：

**select \* from UP\_User where userName like 'F%';**

**# Query\_time: 0.001333** Lock\_time: 0.000149 Rows\_sent: 73 Rows\_examined: 73 Rows\_affected: 0

# Bytes\_sent: 6284 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# InnoDB\_trx\_id: F0AB

# QC\_Hit: No **Full\_scan: No** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 18

## 不要写负向查询

尽量不要用not,<>,not exists ， not in ， not like这些语句查询数据。

## 使用or运算一定要小心

使用or运算一定要小心，如果有一个条件中没有索引，那么可能前面索引一个都用不着，全表扫描。

**select \* from UP\_User where userName='fred' or userId = 10000088 or nickname='viking'**

# Query\_time: 0.004272 Lock\_time: 0.000127 Rows\_sent: 3 Rows\_examined: 1876 Rows\_affected: 0

# Bytes\_sent: 1043 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No **Full\_scan: Yes** Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 26

## 不要用like '%abc'

不要用like '%abc'，否则全表扫描。

**select \* from UP\_User where userName like '%iking'**

**# Query\_time: 0.002858** Lock\_time: 0.000128 Rows\_sent: 1 Rows\_examined: 1876 Rows\_affected: 0

# Bytes\_sent: 801 Tmp\_tables: 0 Tmp\_disk\_tables: 0 Tmp\_table\_sizes: 0

# QC\_Hit: No Full\_scan: Yes Full\_join: No Tmp\_table: No Tmp\_table\_on\_disk: No

# InnoDB\_pages\_distinct: 26

## 避免出现对海量数据库用没有where的select语句

避免出现对海量数据库用select语句，却没有where子句的情况，基本上执行就会JAVA内存爆掉。初学者常犯这种错误，在测试环境和本机测试没问题，一到生产环境就内存溢出了，多半是这种情况引起的。

## 海量数据迁移时一些小技巧

在做数据迁移的时候，有个方法可以非常有效地提高效率，先把索引删除了，再执行数据迁移，等数据迁移完成后再把索引建立好。事实证明，数据库和人一样，在专注做一件时间的时候会比较快，如果有索引，会在迁移数据的时候维护索引，同时还会检查数据的正确性，降低迁移效率。

清理数据库的时候，用truncate比用delete table要快很多。

# 讲师介绍



黄湘龙，北京新媒传信科技有限公司架构师、高级专家、海外解决方案顾问，原北京左岸风暴科技有限公司CTO。

飞信核心平台任职7年，带领飞信服务器侧核⼼开发团队⽀撑飞信产品迭代同时,保障亿级⽤用户、数百万在线系统的平稳运行。经过7年的一线血腥作战，全程参与飞信架构设计与核心代码编写，对承载亿级用户的高可用应用云平台架构有深刻理解。

组建、带领开发团队打造中东陌生人交友App，从开始写第一行代码到产品商用，全程主导产品的开发，包括通信协议设计、各平台协议栈实现、数据库设计、服务器高可用架构等。深度理解互联网应用开发技术，包括高效的二级制协议栈，服务器端高可用框架设计，面向海量数据的JAVA应用开发，WEB前端开发，JAVA性能调优等。