Oracle数据库设计规范

# 1.目的

本规范的主要目的是希望规范数据库设计，尽量提前避免由于数据库设计不当而产生的麻烦；同时好的规范，在执行的时候可以培养出好的习惯，好的习惯是软件质量的很好的保证。

数据库设计是指对于一个给定的应用环境，构造最优的数据库模式，建立数据库及其应用系统，有效存储数据，满足用户信息要求和处理要求。

# 2.适用范围

本规范的适用人员范围包括我司的所有应用开发人员以及在我司承接数据库应用开发的软件人员。

本规范的适用IT范围包括数据库对象的命名规范、设计原则、SQL语句的设计和使用、SQL语句的性能优化建议、其他与性能有关的设计原则以及设计工具的选择。

# 3.数据对象的命名规范

## 3.1.通用规范

### 3.1.1.使用英文

使用英文：要用简单明了的英文单词，不要用拼音，特别是拼音缩写。主要目的很明确，让人容易明白这个对象是做什么用的；

### 3.1.2.表名一律大写

表名一律大写：有些数据库，表的命名乃至其他数据对象的命名是大小写敏感的，为了避免不必要的麻烦，并且尊重通常的习惯，最好一律用大写。

## 3.2.数据库对象命名规范

### 3.2.1.表的命名

#### 3.2.1.1.表名的前缀

表名的前缀：T\_前缀\_表名，为表的名称增加一个或者多个前缀，前缀名不要太长，可以用缩写，最好用下划线与后面的单词分开.

### 3.2.2视图命名

V\_相关表名（或者根据需要另取名字）

### 3.2.3程序包命名

PKG\_程序包名（用英文表达程序包意义）

### 3.2.4.存储过程命名

PRO\_存储过程名（用英文表达存储过程意义）

### 3.2.5.函数命名

函数命名:FUN\_函数名称（用英文表达函数作用）

### 3.2.6.触发器命名

触发器命名:TRI\_触发器名称（用英文表达触发器作用）

### 3.2.7.索引命名

索引命名：IDX\_表名\_字段名(如果存在多字段索引，取每字段前三个字符加下划线组合，如在 custom， cutting， curtail 上建立联合索引，命名为 IDX\_表名\_cus\_cut\_cur，如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如在 custid， custom，custname上建立联合索引，就命名为IDX\_表\_tid\_tom\_tna;

### 3.2.8.唯一索引命名

唯一索引命名：UNI\_表名\_字段名 (如果存在多字段唯一索引，取每字段前三个字符加下划线组合，如在 custom， cutting， curtail上建立唯一索引，命名为 UNI\_表名\_ cus\_cut\_cur，如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如：在 custid， custom，custname上建立唯一索引，命名：UNI\_表\_tid\_tom\_tna;

### 3.2.9.主键命名

主键命名：PK\_表名\_字段名 (如果存在多字段主键，取每字段前三个字符加下划线组合，如在 custom， cutting， curtail上建立主键，命名为 PK\_表名\_cus\_cut\_cur，如果前三个截取字符相同，就从字段名称中不同的字符开始取三个字符加下划线组合，如在 custid， custom，custname上建立主键，命名：PK\_表\_tid\_tom\_tna.

### 3.2.10.外键命名

外键命名：FK\_表名\_主表名\_字段名

### 3.2.11.序列命名

序列命名：SEQ\_表名\_列名（或者根据需要另取名字）

### 3.2.12.同义词命名

同义词命名：与对应的数据库对象同名

# 4.数据库对象设计原则

## 4.1.表的设计

### 4.1.1.主外键

#### 4.1.1.1.业务表要有主键.

每个业务表原则需要有主键，主键是每行数据的唯一标识，保证主键不可随意更新修改，在不知道是否需要主键的时候，请加上主键，它会为你的程序以及将来查找数据中的错误等等，提供一定的帮助.

#### 4.1.1.2.业务表不建议用外键

外键虽然可以维护数据一致性，但是影响数据性能和数据操作，一般生产库上建议不用；

### 4.1.2.列的设计

#### 4.1.2.1.字段的长度一定时间够用

字段的长度要在一定时间内足够用但也不要过宽，占用过多的存储空间，对于长度不确定的列，采用可变长度的数据类型如 varchar类型；

#### 4.1.2.2.字段的类型及长度在设计要以开发或者需求人员沟通

字段的类型及长度在设计，往往要与应用的设计，开发人员商讨，以得到双方认可的类型及宽度.

#### 4.1.2.3.减少冗余列

所谓冗余列，除非必要否则尽量不加冗余列。所谓冗余列，是指能通过其他列计算出来的列，或者是与某列表达同一含义的列，或者是从其他表复制过来的列等等。冗余列需要应用程序来维护一致性，相关列的值改变的时候，冗余列也需要随之修改，而这一规则未必所有人都知道，就有可能因此发生不一致的情况。如果是应用的特殊需要，或者是为了优化某些逻辑很复杂的查询等操作，可以加冗余列；

#### 4.1.3.3.部分数据类型少用

这些数据类型除非必要，尽量不使用LONG， TEXT， BLOB， CLOB， NCLOB， LONG， LONG RAW这一类的数据类型，而是使用其他可以替代的数据类型；优先使用varchar2类型替代CHAR类型，除非列宽有严格的要求而且得到应用严格支持，BLOB类型：如存放照片，文件等放到其它文件服务器或者缓存服务器，而不是直接存放数据库中.

### 4.1.3.表记录数

#### 4.1.3.1.单表的记录数一般控制在两千万条

参考值：2000万，业务系统可以根据实际情况进行适量调整

#### 4.1.3.2.记录数在两千万和两亿条之间的表一定要采用分区技术

记录数在两千万和两亿条之间的表一定要采用分区技术并根据应用的使用情况创建合适的分区标准，单个分区内的记录数一般控制在两千万条(参考值，各应用可以根据实际情况进行适量调整)以内，同时表的索引使用对应的分区索引；

#### 4.1.3.3.记录数超过两亿条的表一定要考虑信息生命周期

记录数超过两亿条的表一定要考虑信息生命周期，必须考虑历史数据的剥离，并在应用设计中完成对历史数据的相应处理功能（历史数据的剥离规则须经业务使用部门的确认）；

### 4.1.4.表和字段中文注释

表和字段名要有中文注释说明

## 4.2.索引的设计

索引是从数据库中获取数据的最高效方式之一。95%的数据库性能问题都可以采用索引技术得到解决。但大量的DML操作会增加系统对索引的维护成本，对性能会有一定影响，对于插入相当频繁的表要慎重建索引，索引也会占相当的存储空间，所以要根据硬件环境和应用需求在空间和时间上达到最好的平衡点，主要原则：

### 4.2.1.适当利用索引

适当利用索引提高查询速度：当数据量比较大，了解应用程序的会有哪些查询，依据这些查询需求建相应的索引；最好亲自试验一下，模拟一下生产环境的数据量，在此数据量下，比较一下建索引前后的查询速度；索引对性能会有一定影响，对于DML频繁列的索引要定期维护（重建）。但是，索引的结构对于索引的更新（比如在插入数据的时候）是有一定优化的，所以不要在没有试验前过分夸大它对性能的影响,最终还是以试验为准.

### 4.2.2.不要建实际用不上的索引

不要建实际用不上的索引，与上条相关，如果建的索引并不提高任何一应用中的查询速度，则要把它删除；有些数据库有相关工具可以发现实际未被使用的索引，可以利用一下.

### 4.2.3.索引类型的选择

索引类型的选择：要根据数据分布及应用来决定如何建立索引，一般的高基数数据列（高基数数据列是指该列有很多不同的值）时 ，建立BTree索引（一般数据库索引的缺省类型）；当低基数数据列（该列有大量相同的值）时，可以考虑建立位图索引（如果所选数据库支持的话），但位图索引是压缩类型索引，所以DML（增、删、改）的代价更高，要综合考虑；

### 4.2.4.索引列的选择

索引列的选择：如果检索条件有可能包含多列，创建联合主键或者联合索引，把最常用于检索条件的列放在最前端，其他的列排在后面；不要索引使用频繁的小型表，假如这些小表有频繁的DML就更不要建立索引，维护索引的代价远远高于扫描表的代价；

### 4.2.5.主键索引在建立的时候要明确指定名称

主键索引在建立的时候一定要明确的指定名称，不能让系统默认建立主键索引（可能有些数据库无法指定主键名，则例外）.

### 4.2.6.外键必须需建索引

不建议外键，索引略.

### 4.2.7.当有联合主键或者联合索引时不要建重复的索引

#### 4.2.7.1.情况1

表EMPLOYEES，它的主键是建立在列DEPARTID和EMPLOYEEID上的联合主键，并且创建主键的语句中DEPARTID在前，EMPLOYEEID在后。在这样一个表里，通常就没有必要再为DEPARTID建一个索引了；联合索引的情况也一样；

#### 4.2.7.2.情况2

更复杂的情况，比如表EMPLOYEES，有一个索引建立在列CORPID， DEPARTID， EMPLOYEEID三列上，在创建语句中也依据上述顺序，就没有必要再为CORPID建立索引；也没有必要再建立以CORPID在前，DEPARTID在后的联合索引；如果EMPLOYEEID需要索引，那么为EMPLOYEEID建立一个索引是不与上面的索引重复的；DEPARTID列也类似；

### 4.2.8 表的索引数量小于五个

表的索引不是越多越好，一般1个表的索引不要超过5个

## 4.3 视图的设计

### 4.3.1.在不太清楚视图用法的情况下尽量不建

在不太清楚视图用法的情况下，尽量不建因为一旦建了，就有被滥用的危险；

### 4.3.2.视图也需要写入数据库设计中

如果需要建视图，只要是打算长期使用的，请写入数据库设计中,明确它的用途、目的；

### 4.3.3.建立视图时要明确写出所有要选择出的列名

建立视图时要明确写出所有要选择出的列名而不要以SELECT \*来代替，可以使结构清晰可读性增强，也不会增加它对表的所有字段的依赖，而表是很可能修改的，特别是增加字段。就很有可能导致使用该视图的应用程序出错.

### 4.4.存储过程&函数&触发器的设计

### 4.4.1.触发器设计

一般不建议用触发器，影响数据库性能，略

### 4.4.2.程序包&存储过程&函数&定时任务.

同应用程序一同加入，进行版本控制，因为此四者包含了代码，应用程序对他们的依赖程度比对表、视图的依赖程度更高.

### 4.4.3.适量使用程序包&存储过程&函数&定时任务

使用这些数据库对象带来的的影响，根据需要合理使用这些数据库对象.

#### 4.4.3.1.提高性能

可以减少数据库与客户端的交互，提高性能

#### 4.4.3.2.提高速度

有的数据库还对他们进行了某种程度的编译，在执行的时候，不用再对其中的SQL等语句进行解析，从而提高速度.

#### 4.4.3.3.复杂逻辑共享

如果有多个应用，使用了不同的开发语言，当有某些关键的或者复杂逻辑希望共享，则可以考虑使用存储过程或者函数。因为存储过程等在数据库一级是共享的；

#### 4.4.3.4.增加维护和迁移难度

增强了应用对数据库的依赖，如果打算将来移植数据库的话，使用得越多，则移植的困难越大；数据库中的业务逻辑越多（存储过程等），应用以及存储过程等的维护难度也会增大；

#### 4.4.3.5.不容易设计出易于扩展的结构

通常存储过程等没有面向对象的特性，不容易设计出易于扩展的结构。当存储过程比较复杂时，或者它们相互间的调用关系比较复杂时，可能难于维护；

# 5.SQL的设计和使用

## 5.1.SQL书写规范

### 5.1.1.尽量不要写复杂的SQL

尽量不要写复杂的SQL：过于复杂的SQL可以用存储过程或函数来代替，效率更高；甚至如果能保证不造成瓶颈的话，把条SQL拆成多条也是可以的。这与一般的编码规范很相似的，首先是要易懂。易懂也就意味着容易维护，对较为复杂的sql语句加上注释，说明算法、功能注释风格：注释单独成行、放在语句前面。

### 5.1.2 应对不易理解的分支条件表达式加注释

### 5.1.3.对重要的计算应说明其功能

### 5.1.4.过长的函数实现应按实现的功能分段加以说明

### 5.1.5.每条复杂SQL语句均应有注释说明

每条复杂SQL语句均应有注释说明:表名,字段名主要是说明此句SQL执行的作用及所取得结果集的意义

### 5.1.6.常量及变量注释时，应注释被保存值的含义和合法取值的范围

### 5.1.7.可采用单行/多行注释

可采用单行/多行注释：-- 或 /\* \*/ 方式，不同数据库可能语法不同

### 5.1.8.连接符

连接符:or、in、and、以及＝、<=、>=等前后加上一个空格

### 5.1.9.不要用SELECT \*

不要用SELECT \*：SELECT语句中写出必要的要选择的全部列名，增强语句可读性，避免不必要的选择；SELECT \* 增加了对所有字段的依赖，当表增加了字段后，有可能发生错误；此外还可能增加了数据的流量，查询了一些实际不需要的字段；

### 5.1.10.避免长事务（Transaction）

避免长事务（Transaction）：长事务容易造成死锁，应该避免，单个事务使用的数据库和系统资源不宜超过总资源1-2%（参考值，各应用可以根据实际情况进行适量调整，这种情况不适用于数据仓库）

### 5.1.11.行最长不能超过80字符

行最长不能超过80字符，同一语句不同字句之间逗号以后空格，其他分割符前空格 where子句书写时，每个条件占一行，语句令起一行时，以保留字或者连接符开始，连接符右对齐

### 5.1.12.多表连接时使用表的别名来引用列

### 5.1.13. SQL中对视图的引用

SQL中对视图的引用：在不太清楚视图用法的情况下尽量不用。只是因为视图中有自己想要的字段就拿来用，是相当普遍和错误的用法。原因如下：

#### 5.1.13.1.对数据库资源会有过多的消耗

增加了不必要的数据流量，对你的实际需求，那很可能是一个非常复杂的视图，有大量你不需要的字段，并且关联了很多你实际不需要的表，对数据库资源会有过多的消耗；

#### 5.1.13.2.增加了应用程序对视图的依赖

增加了应用程序对视图的依赖，不必要的依赖是越少越好的。当有应用程序依赖了某个视图，不久可能其他人因为某种原因会修改此视图，原来的应用有可能会受到不同程度的影响；

## 5.2. SQL 性能优化建议

### 5.2.1.量小的表作为驱动表

系统可能选择基于规则的优化器，所以将结果集返回数据量小的表作为驱动表，即将结果集返回数据量小的表放在FROM后边最后一个表

### 5.2.2.大量的排序操作影响系统性能

大量的排序操作影响系统性能,所以尽量减少order by和group by排序操作.

### 5.2.3 如必须使用排序操作，排序尽量建立在有索引的列上

### 5.2.4.索引的使用

#### 5.2.4.1.尽量避免对索引列进行计算

尽量避免对索引列进行计算。如对索引列计算较多，请提请数据库管理员建立函数索引

#### 5.2.4.2..尽量注意比较值与索引列数据类型的一致性

尽量注意比较值与索引列数据类型的一致性:number与number比较、char与char比较，避免使用数据库的类型自动转换功能；如下数值和字符自动转避免：

-- id是数值型

SELECT \* FROM category

WHERE id = ‘123’;

#### 5.2.4.3.对于复合索引，SQL语句必须使用第1个列

复合索引必须要用到第1个索引列，这个复合索引才能使用到

#### 5.2.4.4 索引字段中，尽量避免使用NULL值

#### 5.2.4.5.对于索引的比较，尽量避免使用不等于符合

如运算符合:<>,!=,not in等

#### 5.2.4.6.查询列和排序列与索引列次序保持一致

### 5.2.5.尽量避免相同语句由于书写格式的不同

尽量避免相同语句由于书写格式的不同，而导致多次语法分析(减少数据库的硬分析)，如可使用TOAD的格式化工具对SQL语句进行格式化处理

### 5.2.6.查询的WHERE过滤原则，应使过滤记录数最多的条件放在最前面

### 5.2.7.在WHERE中数据库函数&计算表达式等等，要尽可能将放在等号右边

在WHERE中，数据库函数、计算表达式等等，要尽可能将放在等号右边。否则会使所比较的字段上的索引失效；如：

SELECT　\*

FROM　service\_promotion

WHERE TO\_CHAR(gmt\_modified，’yyyy-mm-dd’)=’2001-09-01’;

而应使用：

SELECT \*

FROM service\_promotion

WHERE gmt\_modified= TO\_DATE(‘2001-09-01’,’yyyy-mm-dd’)

### 5.2.8. in&or子句常会使索引失效，尽可能不使用in、or

使用 IN 或 NOT IN 或者or子句时，特别是当子句中有多个值且表数据较多时，速度会明显下降,可以采用连接查询或外连接查询来提高性能。

### 5.2.9.尽量避免在循环中使用SQL语句

### 5.2.10.在循环中尽量使用动态SQL语句提高执行性能

### 5.2.11.复杂SQL

对于非常复杂的SQL（特别是有多层嵌套，带子句或相关子查询的），应该先考虑是否设计不当引起的，对于一些复杂SQL可以考虑使用程序实现，而不是写很复杂很长的SQL完成功能.

### 5.2.12.参数的传递

SQL语句的编写，变量尽量使用“?”绑定变量。

# 6.数据库相关工具

为需要写SQL的开发人员提供方便的、图形化的客户端软件，可以用来执行SQL，看到表结构、索引等等，如TOAD， PL/SQL Develop软件。