索引分区

一、索引

在oracle中，索引是一种供服务器在表中快速查找一个行的数据库结构。通俗的来讲，索引在表中的作用，相当于书的目录对书的作用。索引直接指向包含所查询值的行的位置，减少磁盘I/O开销，提高查询时间。Oracle 自动使用并维护索引，插入、删除、更新表后，自动更新索引。

1.1 作用

在数据库中建立索引主要有以下作用

- 快速存取数据
- 既可以改善数据库性能，又可以保证列值的唯一性。
- 实现表与表之间的参照完整性
- 在使用orderby、groupby子句进行数据检索时，利用索引可以减少排序和分组的时间。

1.2 优点

- 索引是与表相关的一个可选结构
  - 一个表中可以存在索引，也可以不存在索引，不做硬性要求。
- 用以提高 SQL 语句执行的性能
  - 快速定位需要查找的表的内容（物理位置），提高sql语句的执行性能。
- 减少磁盘I/O
  - 取数据从磁盘上取到数据缓冲区中，再交给用户。磁盘IO非常不利于表的查找速度（效率的提高）。
- 使用 CREATE INDEX 语句创建索引
- 在逻辑上和物理上都独立于表的数据
  - 索引与表完全独立，表里的内容是我们真正感兴趣的内容，而索引则是做了一些编制 ，索引和数据可以存，在不同的表空间下面，可以存放在不同的磁盘下面。
- Oracle 自动维护索引
  - 当对一个建立索引的表的数据进行增删改的操作时，oracle会自动维护索引，使得其仍然能够更好的工作。

1.3 分类

在关系数据库中，每一行都由一个行唯一标识RowID。RowID包括该行所在的文件、在文件中的块数和块中的行号。索引中包含一个索引条目，每一个索引条目都有一个键值和一个RowID，其中键值可以是一列或者多列的组合。用户可以在Oracle中创建多种类型的索引，以适应各种表的特点。

- 按照索引列的唯一性可以分为**唯一索引**和**非唯一索引**；
- 按照索引列的个数可以分为**单列索引**和**复合索引。**
- 按照索引数据的存储方式可以将索引分为**B树索引**、**位图索引**、**反向键索引**和**基于函数的索引**；

1.4 创建

基本语法

```
CREATE UNIUQE | BITMAP INDEX <schema>.<index_name>
ON <schema>.<table_name>
   (<column_name> | <expression> ASC | DESC,<column_name> | <expression> ASC |
DESC,...)
TABLESPACE <tablespace_name>
STORAGE <storage_settings>
LOGGING | NOLOGGING
COMPUTE STATISTICS
NOCOMPRESS | COMPRESS<nn>
NOSORT | REVERSE
PARTITION | GLOBAL PARTITION<partition_setting>
```

说明

- UNIQUE | BITMAP：指定UNIQUE为唯一值索引，BITMAP为位图索引，省略为B-Tree索引
- <column_name> | ASC | DESC：可以对多列进行联合索引，当为expression时即“基于函数的索引”
- TABLESPACE：指定存放索引的表空间(索引和原表不在一个表空间时效率更高)
- STORAGE：可进一步设置表空间的存储参数
- LOGGING | NOLOGGING：是否对索引产生重做日志(对大表尽量使用NOLOGGING来减少占用空间并提高效率)
- COMPUTE STATISTICS：创建新索引时收集统计信息
- NOCOMPRESS | COMPRESS：是否使用“键压缩”(使用键压缩可以删除一个键列中出现的重复值)
- NOSORT | REVERSE：NOSORT表示与表中相同的顺序创建索引，REVERSE表示相反顺序存储索引值
- PARTITION | NOPARTITION：可以在分区表和未分区表上对创建的索引进行分区

B树索引

B树索引的存储结构类似书的索引结构，有分支和叶两种类型的存储数据块，分支块相当于书的大目录，叶块相当于索引到的具体的书页。Oracle用B树机制存储索引条目，以保证用最短路径访问键值。

默认情况下大多使用B树索引，**唯一索引、反向键索引都属于B树索引。**

示例

```
/*
语法：
CREATE INDEX index_name
ON table_name(column_name);

具体列值：  索引相关列上的值必须唯一，但可以不限制NULL值。
*/
```

唯一索引

唯一索引确保在定义索引的列中**没有重复值**，Oracle 自动在表的主键列上创建唯一索引

示例：

```
/*
语法：
CREATE UNIQUE INDEX index_name
ON table_name(column_name);

具体列值： 索引相关列上的值必须唯一，但可以不限制NULL值。
*/
```

复合索引

复合索引是在表的**多个列上**创建的索引， 索引中列的顺序是任意的，如果 SQL 语句的 WHERE 子句中引用了组合索引的所有列或大多数列，则可以提高检索速度

```
/*
语法：
CREATE UNIQUE INDEX index_name
ON table_name(column_name1, column_name2);

具体列值：该索引中的元组由两列共同确定一行
*/
```

反向键索引

反向键索引也逆序索引。该索引同样保持列按顺序排列，但是颠倒已索引的每列的字节。反向键索引反转索引列键值的每个字节，为了实现索引的均匀分配，避免b树不平衡，通常建立在值是连续增长的列上，使数据均匀地分布在整个索引上。

创建索引时使用**REVERSE**关键字

```
/*
语法：
CREATE INDEX index_name
ON table_name(column_name1)
REVERSE;

具体列值：适用于某列值前面相同，后几位不同的情况
CLASSID：1001 1002 1003 1004 1005 1006 1007
索引转化：1001 2001 3001 4001 5001 6001 7001
*/
```

位图索引

位图索引适合创建在低基数列（数值固定）上，位图索引不直接存储ROWID，而是存储字节位到ROWID的映射节省空间占用，减少oracle对数据块的访问。它采用位图偏移方式来与表的行ID号对应，采用位图索引一般是重复值太多的表字段。位图索引之所以在实际密集型OLTP（联机事物处理）中用的比较少，是因为OLTP会对表进行大量的删除、修改、新建操作。

如果索引列被经常更新的话，不适合建立位图索引，总体来说，位图索引适合于数据仓库中，不适合OLTP中

```
/*
语法：
CREATE BITMAP INDEX index_name
ON table_name(column_name);

具体列值：不适用于经常更新的列，适用于条目多但取值类别少的列，例如性别列
*/
```

基于函数的索引

基于一个或多个列上的函数或表达式创建的索引。索引中的一列或者多列是一个函数或者表达式，索引根据函数或表达式计算索引列的值。可以将基于函数的索引建立创建成位图索引。

- 表达式中不能出现聚合函数
- 不能在LOB类型的列上创建
- 创建时必须具有 QUERY REWRITE 权限

```
语法：create index index_name on table_name （函数(column_name)）;
具体列值：  不能在LOB类型的列上创建，用户在该列上对该函数有经常性的要求。
    例如：用户不知道存储时候姓名是大写还是小写，使用
    select * from student where upper(sname)='TOM';

/*
语法：
CREATE INDEX index_name
ON table_name(函数(column_name));

具体列值：不能在LOB类型的列上创建，用户在该列上对该函数有经常性的要求。
例如：用户不知道存储时候姓名是大写还是小写，使用
select * from student where upper(sname)='TOM';
*/
```

1.5 创建索引的原则

1. 一般不需要为数据量很小的表创建索引
2. 对于数据量比较大的表，如果**经常需要查询的记录数小于表中所有记录数的10%**，则可以考虑为该表创建索引
3. 应该为大部分列值不重复的列创建索引
4. 对于取值范围较大的列（如ename列），应该创建B树索引；对于取值范围较小的列（如sex列），应该创建位图索引
5. 对于包含很多个NULL值，但是经常需要查询所有非NULL值记录的列，应当创建索引
6. 不能在CLOB或BLOB等大对象数据类型的列上创建索引
7. 如果在大部分情况下只需要对表执行只读操作，就可以为该表创建更多的索引以提高查询速度
8. 如果在大部分情况下需要对表执行更新操作，则应该为少创建一些索引，以提高更新速度

二、分区

Oracle在实际业务生产环境中，经常会遇到随着业务量的逐渐增加，表中的数据行数的增多，Oracle对表的管理和性能的影响也随之增大。对表中数据的查询、表的备份的时间将大大提高，以及遇到特定情况下，要对表中数据进行恢复，也随之数据量的增大而花费更多的时间。这个时候，Oracle数据库提供了分区这个机制，通过把一个表中的行进行划分，归为几部分。可以减少大数据量表的管理和性能问题。利用这种分区方式把**表数据进行划分的机制**称为表分区，各个分区称为分区表。

- 分区是将一个表或索引物理地分解为多个更小、更可管理的部分。
- 分区对应用透明，即对访问数据库的应用而言，逻辑上讲只有一个表或一个索引（相当于应用“看到”的只是一个表或索引），但在物理上这个表或索引可能由数十个物理分区组成。
- 每个分区都是一个独立的对象，可以独自处理，也可以作为一个更大对象的一部分进行处理。

优点

1. 分区技术使数据库的可管理性变得更加容易，如：用户可以往一个单独的分区中装载数据，而对其他分区没有任何影响；用户可以在单独的分区上创建索引等。
2. 分区可以提高表的查询性能，SQL语句的where子句会过滤掉不需要的分区，oracle不会再扫描那些不需要的分区。
3. 分区技术减少数据的不可用时间，用户可以单独维护一个分区中的数据，而不影响其他分区中数据的使用。
4. 分区技术在数据库级完成，几乎不需要对应用程序做任何修改。

分区方法

- 范围分区：根据表中列值的范围将整个表分成不同的部分，如按照时间进行范围分区。
- 列表分区：使用列表值将表划分成几部分。
- 哈希分区：使用哈希函数把表分成几部分。
- 复合分区：同时使用两种分区方法对表进行分区。

范围分区

范围分区将数据基于范围映射到每一个分区，这个范围是你在创建分区时指定的分区键决定的

特点

1. 每一个分区都必须有一个VALUES LESS THAN子句，它指定了一个不包括在该分区中的上限值。分区键的任何值等于或者大于这个上限值的记录都会被加入到下一个高一些的分区中。
2. 所有分区，除了第一个，都会有一个隐式的下限值，这个值就是此分区的前一个分区的上限值。
3. 在最高的分区中必须有一个MAXVALUE。代表了一个不确定的值。这个值高于其它分区中的任何分区键的值，可以理解为高于任何分区中指定的VALUE LESS THEN的值，同时包括空值

示例

```
/*
语法格式：
PARTITION BY RANGE(RANGE_COLNUM)(
 PARTITION part_1 VALUES LESS THAN (parameter) TABLESPACE space_name
 …
)
*/

CREATE TABLE CUSTOMER(
 CUSTOMER_ID NUMBER NOT NULL PRIMARY KEY,
 FIRST_NAME VARCHAR2(30) NOT NULL,
 LAST_NAME VARCHAR2(30) NOT NULL,
 PHONE VARCHAR2(15) NOT NULL
)
PARTITION BY RANGE(CUSTOMER_ID)(
 PARTITION CUS_PART1 VALUES LESS THAN(1000) TABLESPACE CUS_TS01,
 PARTITION CUS_PART2 VALUES LESS THAN(20000) TABLESPACE CUS_TS02,
 PARTITION CUS_PART3 VALUES LESS THAN(MAXVALUE) TABLESPACE CUS_TS03
)
```

```
-- 也可以在创建表时不指定表空间进行分区
CREATE TABLE tab_range_test (
 range_key_column date not null,
 data varchar2(20)
)
PARTITION BY RANGE (range_key_column) (
 PARTITION PART1 VALUES LESS THAN(to_date('01/01/2010','dd/MM/yyyy' )) ,
 PARTITION PART2 VALUES LESS THAN(to_date('01/01/2011','dd/MM/yyyy' )) ,
 PARTITION PART3 VALUES LESS THAN(MAXVALUE)
)
```

列表分区

该分区的特点是某列的值只有几个，基于这样的特点我们可以采用列表分区

示例

```
/*
语法格式:
PARTITION BY LIST(RANGE_COLNUM)(
 PARTITION part_1 VALUES(parameter) TABLESPACE space_name
 …
)
*/

CREATE TABLE CUSTOMER(
 CUSTOMER_ID NUMBER NOT NULL PRIMARY KEY,
 FIRST_NAME VARCHAR2(30) NOT NULL,
 CITY VARCHAR2(10) NOT NULL
)
PARTITION BY LIST(CITY)(
 PARTITION CUS_PART1 VALUES('河南') TABLESPACE CUS_TS01,
 PARTITION CUS_PART2 VALUES('河北') TABLESPACE CUS_TS02
)
```

散列分区

散列分区又称Hash分区，是在列的取值难以确定的情况下采用的一种分区方式。散列分区通过指定分区编号将数据均匀分布在磁盘设备上，使得这些分区大小一致，这就降低了I/O磁盘争抢的情况。

对一个表执行散列分区时，Oracle会对分区键应用一个散列（Hash）函数，以此确定数据应当放在 N 个分区中的哪一个分区中。

Oracle建议 N 是 2 的一个幂（如 N = 2、4、8、16 等），从而使表数据得到最佳的总体分布。

当列的值没有合适的范围条件时，建议使用散列分区。

```
/*
语法格式:
PARTITION BY HASH(RANGE_COLNUM)(
 PARTITION part_1 TABLESPACE space_name
 …
)
*/
```

```
CREATE TABLE CUSTOMER(
 CUSTOMER_ID NUMBER NOT NULL PRIMARY KEY,
 FIRST_NAME VARCHAR2(30) NOT NULL,
 CITY VARCHAR2(10) NOT NULL
)
PARTITION BY HASH(FIRST_NAME)(
 PARTITION CUS_PART1 TABLESPACE CUS_TS01,
 PARTITION CUS_PART2 TABLESPACE CUS_TS02
)
```

查看分区情况

```
SELECT * FROM USER_TAB_PARTITIONS WHERE TABLE_NAME = 'CUSTOMER';
```

查看分区数据

```
SELECT * FROM customer PARTITION(P1);
```

删除分区

```
alter table tablename drop partiton p4;
```

三、数据库优化

3.1 为什么要对SQL进行优化

我们开发项目上线初期，由于业务数据量相对较少，一些SQL的执行效率对程序运行效率的影响不太明显，而开发和运维人员也无法判断SQL对程序的运行效率有多大，故很少针对SQL进行专门的优化，而随着时间的积累，业务数据量的增多，SQL的执行效率对程序的运行效率的影响逐渐增大，此时对SQL的优化就很有必要。、

3.2 SQL优化方式

- 服务器硬件优化
- 服务器优化
- SQL本身优化
- 索引优化
- 反范式设计优化

3.2.1 服务器硬件优化

硬盘、CPU、内存

3.2.2 服务器优化

提高缓冲区的效率

修改游标参数，用户任何时点拥有的打开光标的最大数

实现分布式快速存取和充实内存是很重要的

3.2.3 SQL本身优化

1. 避免全表扫描
    1. 应尽量避免在 where 子句中对字段进行 null 值判断（可以设置默认值0）
    2. 应尽量避免在 where 子句中使用!=或<>操作符（=）
    3. 应尽量避免在 where 子句中使用 or 来连接条件（可以使用union all）
    4. 避免使用in 和 not in 也要慎用（用 exists 代替 in ）
    5. 减少使用like
    6. 应尽量避免在 where 子句中对字段进行表达式操作
    7. 应尽量避免在where子句中对字段进行函数操作
2. 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
3. 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
4. 尽量使用数字型字段
5. 用具体的字段列表代替“*”
6. 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。
7. 写大写

3.2.4 索引优化

适当添加索引

3.2.5 反范式设计优化

数据库设计三大范式

第一范式（1NF）：要求数据库表的每一列都是不可分割的原子数据项

用户信息表

编号	姓名	性别	年龄	联系电话	省份	城市	详细地址
1	张红欣	男	26	0378-23459876	河南	开封	朝阳区新华路23号
2	李四平	女	32	0751-65432584	广州	广东	白云区天明路148号
3	刘志国	男	21	0371-87659852	河南	郑州	二七区大学路198号
4	郭小明	女	27	0371-62556789	河南	郑州	新郑市薛店北街218号

第二范式（2NF）：在1NF的基础上，非码属性必须完全依赖于候选码（在1NF基础上消除非主属性对主码的部分函数依赖）

订单编号	商品编号	商品名称	数量	单位	价格	客户	所属单位	联系方式
001	1	挖掘机	1	台	1200000￥	张三	上海玖智	020-1234567
001	2	冲击钻	8	把	230￥	张三	上海玖智	020-1234567
002	3	铲车	2	辆	980000￥	李四	北京公司	010-1234567

这样就产生一个问题：这个表中是以订单编号和商品编号作为联合主键。这样在该表中商品名称、单位、商品价格等信息不与该表的主键相关，而仅仅是与商品编号相关。所以在这里违反了第二范式的设计原则。

而如果把这个订单信息表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中，就非常完美了。如下所示。

这样设计，在很大程度上减小了数据库的冗余。如果要获取订单的商品信息，使用商品编号到商品信息表中查询即可。

第三范式（3NF）：在2NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。如下面这两个表所示的设计就是一个满足第三范式的数据库表。

订单信息表

订单编号	订单项目	负责人	业务员	订单数量	客户编号
001	挖掘机	刘明	李东明	1台	1
002	冲击钻	李刚	霍新峰	8个	2
003	铲车	郭新一	艾美丽	2辆	1

客户信息表

客户编号	客户名称	所属公司	联系方式
1	李聪	五一建设	13253661015
2	刘新明	个体经营	13285746958

这样在查询订单信息的时候，就可以使用客户编号来引用客户信息表中的记录，也不必在订单信息表中多次输入客户信息的内容，减小了数据冗余。

反范式设计优化

范式化设计减少了表之间的数据冗余（保存的数据量减少），增强了表之间的关联姓，查询变得复杂

- 针对范式化而言
- 为了性能和读取效率的考虑而适当的对数据进行
- 允许少量的冗余，使用空间换取时间