4

分区、视图和其它模式对象

虽然表和索引是最重要和经常使用的模式对象，但数据库还支持许多其它类型的模式对象，其中最常见会在本章中讨论。

本章包含以下部分：

- 分区概述
- 视图概述
- 物化视图概述
- 序列概述
- 维度概述
- 同义词概述

分区概述

分区（Partitioning）使你能够将非常大的表和索引分解成为更小且更易于管理被称为**分区（partitions）**的片中。每个分区都是拥有自己的名称和自身可选的存储特性的独立对象。

对说明分区打个比方，假设人力资源经理有一个包含员工文件夹的大箱子。每个文件夹都列出员工的雇佣日期。查询经常使用在一个特定月份中对员工的雇佣日期。一种满足这种需求的途径是在员工雇佣日期上创建索引，指定散落到每个箱子的文件夹位置。与此相反，一个分区策略使用许多小箱子，每个箱子包含在给定月份的雇佣员工的文件夹。

使用更小的箱子有几个优点。当访问检索在六月中雇佣员工的文件夹，人力资源经理可以检索六月的箱子。此外，如果任一小箱子暂时受损，而其他的小箱子仍然可用。移动办公也变得更简单，因为经理可以移动几个小箱子，而不是移动单个沉重的箱子。

从应用的角度看，仅有一个模式对象存在。DML 语句无需修改就可访问分区表。

对于许多不同类型的数据库应用来说分区是有用的，特别是管理大量数据。有点包括：

- 提高可用性

  一个分区不可用不代表这个对象不可用。查询**优化器（optimizer）**自动从

查询计划（query plan）中移除未引用分区，因此当分区不可用时，查询没有影响。

- 更轻松的管理模式对象
  分区对象包含可以被集体或个人管理的片。DDL 语句可以操纵分区而不是整个表或索引。因此，你可以拆分资源密集型任务，如重建索引或表。例如，你可以在同一时间移动一个表的分区。如果出现问题，那么只有移动的分区必须重做，而不是移动的表。另外，删除分区避免了执行大量的 DELETE 语句。
- 减少在 OLTP 系统中共享资源争用
  在一些 OLTP 系统中，分区可以减少共享资源的争用。例如，DML 分布跨越许多段而不是一个段中。
- 增强在数据仓库中的查询性能
  在**数据仓库**（data warehouse）中，分区可以加速处理即席查询。例如，一张包含百万行的销售表可以按照季度进行分区。

参见：Oracle 数据库 VLDB 和分区指南 对于分区的导论

分区特性

表或索引的每一个分区必须有相同的逻辑属性，如**列**（column）名、**数据类型**（data types）和约束。例如，在表中所有分区共享相同的列和约束定义，并且在索引中的所有分区共享相同的索引列。然而，每个分区可以有不同的物理属性，如它所属的表空间。

分区键

分区键（partition key）是一列或多列的集合，来确定分区其中的每一**行**（row）应该去的分区。每一行明白的分配给单个分区。

在 sales 表中，你可以指定 time_id 列作为范围分区的键。数据库将行分配到基于日期是否在这个列中落入到指定范围的分区。Oracle 数据库自动直接插入、更新和删除操作到使用分区键的相应分区。

分区策略

Oracle 分区提供了几种分区策略来控制数据库如何放置数据到分区中。基本策略有范围、列表和散列分区。

单一级别（single-level）分区策略仅使用一种数据分布方法，例如，只有列表分区或只有范围分区。在**组合分区**（composite partitioning）中，一张分区表由一种数据分配方法并且每一个分区使用第二种数据分配方法更进一步分成子分区。例如，你可以对 channel_id 使用列表分区并且对 time_id 使用范围子分区。

范围分区 在**范围分区**（range partitioning）中，数据库映射行到基于范围值得分区键的分区中。范围分区是最常用的分区类型并且经常和日期使用。

假设你想要用带有在例 4–1 中所示的 sales 行填充分区表。

例 4–1 分区表的示例行集合

```
PROD_ID CUST_ID TIME_ID CHANNEL_ID PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
116 11393 05-JUN-99 2 999 1 12.18
40 100530 30-NOV-98 9 33 1 44.99
118 133 06-JUN-01 2 999 1 17.12
133 9450 01-DEC-00 2 999 1 31.28
36 4523 27-JAN-99 3 999 1 53.89
125 9417 04-FEB-98 3 999 1 16.86
30 170 23-FEB-01 2 999 1 8.8
24 11899 26-JUN-99 4 999 1 43.04
35 2606 17-FEB-00 3 999 1 54.94
45 9491 28-AUG-98 4 350 1 47.45
```

你可以创建 time_range_sales 作为分区表使用在列 4–2 中的语句。time_id 列是分区键。

例 4–2 范围分区表

```
CREATE TABLE time_range_sales
 ( prod_id NUMBER(6)
 , cust_id NUMBER
 , time_id DATE
 , channel_id CHAR(1)
 , promo_id NUMBER(6)
 , quantity_sold NUMBER(3)
 , amount_sold NUMBER(10,2)
 )
PARTITION BY RANGE (time_id)
 (PARTITION SALES_1998 VALUES LESS THAN
(TO_DATE('01-JAN-1999','DD-MON-YYYY')),
 PARTITION SALES_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
 PARTITION SALES_2000 VALUES LESS THAN (TO_DATE('01-JAN-2001','DD-MON-YYYY')),
 PARTITION SALES_2001 VALUES LESS THAN (MAXVALUE)
 );
```

之后，你从例 4-1 中加载带行的 time_range_sales。图 4-1 显示了行分布在四个分区中。数据库为每个基于 time_id 值的行根据在 PARTITION BY RANGE 子句中指定的规则选择分区。

图 4–1 范围分区

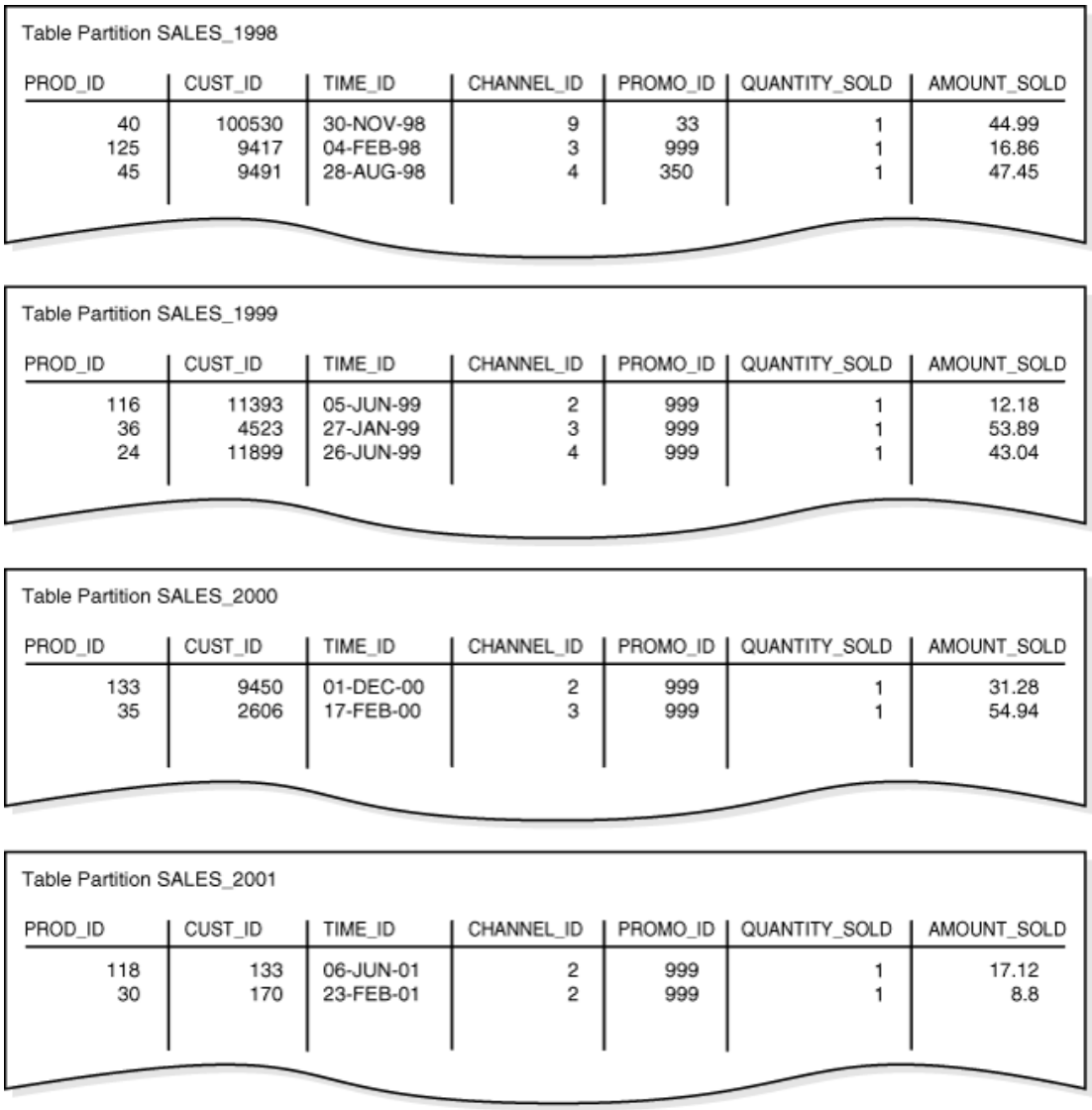


Table Partition SALES_1998

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
125	9417	04-FEB-98	3	999	1	16.86
45	9491	28-AUG-98	4	350	1	47.45

Table Partition SALES_1999

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
36	4523	27-JAN-99	3	999	1	53.89
24	11899	26-JUN-99	4	999	1	43.04

Table Partition SALES_2000

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
133	9450	01-DEC-00	2	999	1	31.28
35	2606	17-FEB-00	3	999	1	54.94

Table Partition SALES_2001

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
118	133	06-JUN-01	2	999	1	17.12
30	170	23-FEB-01	2	999	1	8.8

范围分区键值决定范围分区的高值，被称为**过渡点（transition point）**。在图 4–1 中,SALES_1998 分区包含带有分区键 time_id 值的行小于转折点 01-JAN-1999。数据库为超过过渡点的数据创建**间隔分区（interval partitions）**。间隔分区扩展分区范围通过指示数据库创建指定范围的分区或者当数据插入到表中超过所有范围分区时自动间隔。在图 4–1 中，SALES_2001 分区包含带有分区键 time_id 值的行大于或等于 01-JAN-2001。

列表分区 在**列表分区**中，数据库为每个分区使用离散值列表作为分区键。你可以使用列表分区来控制各行如何映射到指定分区中。当键不是方便有序的用来识别它们时，通过使用列表，你可以分组并组织相关数据集。

假设你使用在例 4–3 中的语句创建 list_sales 为列表分区表。channel_id 列作为分区键。

例 4–3 列表-分区表

```
CREATE TABLE list_sales
 ( prod_id NUMBER(6)
 , cust_id NUMBER
 , time_id DATE
 , channel_id CHAR(1)
 , promo_id NUMBER(6)
 , quantity_sold NUMBER(3)
 , amount_sold NUMBER(10,2)
 )
 PARTITION BY LIST (channel_id)
 (PARTITION even_channels VALUES (2,4),
 PARTITION odd_channels VALUES (3,9)
 );
```

之后，你从例 4-1 中加载带有行的表。图 4-2 显示了在两个分区中的行分布。数据库基于 channel_id 的值根据在 PARTITION BY LIST 子句中指定的规则为每一行选择分区。带有 channel_id 值为 2 或 4 的行被存储在 EVEN_CHANNELS 分区中，同时带有 channel_id 值为 3 或 9 的行被存储在 ODD_ CHANNELS 分区中。

图 4–2 列表分区

Table Partition EVEN_CHANNELS

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
118	133	06-JUN-01	2	999	1	17.12
133	9450	01-DEC-00	2	999	1	31.28
30	170	23-FEB-01	2	999	1	8.8
24	11899	26-JUN-99	4	999	1	43.04
45	9491	28-AUG-98	4	350	1	47.45

Table Partition ODD_CHANNELS

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
36	4523	27-JAN-99	3	999	1	53.89
125	9417	04-FEB-98	3	999	1	16.86
35	2606	17-FEB-00	3	999	1	54.94

散列分区 在**散列分区（hash partitioning）**中，数据库将行映射到基于**散列（hashing）**算法的分区，且数据库应用由用户指定的分区键。行的目的地是由数据库应用到行的内部**哈希函数（hash function）**决定的。散列算法被设计成将行均匀分布到各个设备中，因此每个分区包含大约相同数量的行。散列分区在提高可管理性方面对分解大表十分有用。而不是来管理一张大表，你有几个更小的碎片。单个散列分区的丢失不影响其余的分区，并且可以独立的恢复。散列分区在有大量更新争用的 OLTP 系统中也是非常有用的。

例如，一个段被分解为多个片，每个片都可以被更新，而不是单一段去经历争用。

假设你使用在例 4-4 中的语句创建分区 hash_sales 表。prod_id 列为分区键。

例 4–4 散列-分区表

```
CREATE TABLE hash_sales
 ( prod_id NUMBER(6)
 , cust_id NUMBER
 , time_id DATE
 , channel_id CHAR(1)
 , promo_id NUMBER(6)
 , quantity_sold NUMBER(3)
 , amount_sold NUMBER(10,2)
 )
PARTITION BY HASH (prod_id)
PARTITIONS 2;
```

之后，你从例 4–1 中加载带有行的表。图 4–3 显示了在两个分区中可能的行分配。请注意这些分区的名称是系统产生的。

如你插入行，数据库尝试随机并均匀的将它们分布到分区中。你不能指定将行放置到哪个分区中。数据库应用散列函数，散列函数会确定分区包含哪行的结果。如果你改变分区的数量，那么数据库会跨越所有分区重新分配数据。

图 4–3 散列分区

Table Partition SYS_P33

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
40	100530	30-NOV-98	9	33	1	44.99
118	133	06-JUN-01	2	999	1	17.12
36	4523	27-JAN-99	3	999	1	53.89
30	170	23-FEB-01	2	999	1	8.8
35	2606	17-FEB-00	3	999	1	54.94

Table Partition SYS_P34

PROD_ID	CUST_ID	TIME_ID	CHANNEL_ID	PROMO_ID	QUANTITY_SOLD	AMOUNT_SOLD
116	11393	05-JUN-99	2	999	1	12.18
133	9450	01-DEC-00	2	999	1	31.28
125	9417	04-FEB-98	3	999	1	16.86
24	11899	26-JUN-99	4	999	1	43.04
45	9491	28-AUG-98	4	350	1	47.45

参见：

- *Oracle 数据库 VLDB 和分区指南* 来了解如何创建分区

■ *Oracle 数据库 SQL 语言参考手册* 关于 CREATE TABLE ... PARTITION BY 的例子

分区表

分区表（partitioned table）由一个或多个分区构成，分区可以单独管理并可以单独操作于其他分区。一张表可以是分区的或者未分区的。即使分区表由一个分区组成，这张表时与非分区表不同的，非分区表是不能增加分区的。在 4-2 页的“分区特性”给出了分区表的例子。

分区表是由一个或多个表的分区段构成。如果你创建一张分区表叫 hash_products，那么没有表**段（segment）**分配给这张表。相反的，数据库在它自身的分区段中为每个表分区存储数据。每个表分区段包含表数据的一部分。

堆-组织表的一些或所有分区可以被存储在压缩格式中。压缩节省空间并且可以加速查询执行。因此，压缩在如插入和更新操作很少的数据仓库的环境和 OLTP 环境中十分有用。

表压缩（table compression）的属性可以被声明在表空间、表或者表分区。如果声明在表空间级别，那么在表空间中创建表默认为压缩的。你可以修改表的压缩属性，在这种情况下仅应用于进入到那张表的新数据。因此，单表或者分区可能包含压缩和非压缩块，从而保证因压缩数据大小不会增长。如果压缩可以增长块的大小，那么数据库不将它应用到块中。

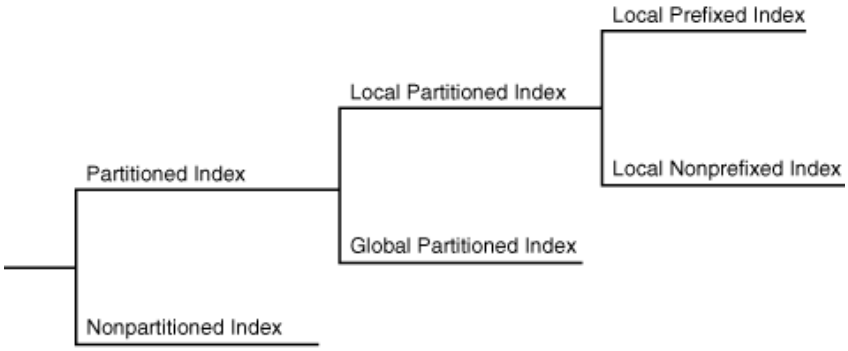
参见：

- “表压缩”在 2-19 页和“段概述”在 12-21 页
- *Oracle 数据库数据仓库指南* 来了解关于在数据仓库中的表压缩

分区索引

分区索引（partitioned index）是一种索引，类似于分区表，被分解到更小且更易于管理的片中。**全局索引（Global indexes）**在表的独立分区之上创建的，而**本地索引（local indexes）**是自动的链接到一张表的分区方法。像分区表、分区索引可以提高可管理性、可用性、性能和可扩展性。

下图显示了索引分区选项。



参见：

- “索引概述”在 3-1 页
- *Oracle 数据库 VLDB 和分区指南* 和 *Oracle 数据库性能调优指南* 来获取更多关于分区索引和如何决定使用类型的信息

本地分区索引

在**本地分区索引（local partitioned index）**中，索引在同一列上被分区，并带有这张表的相同的分区数量和相同的分区边界。每个索引分区准确的关联本表的每一个分区，这样在索引分区中的所有键仅指向在单表分区中的行存储。通过这种方式，数据库自动同步索引分区和它们相关的表分区，使每个表-索引对独立。本地分区索引在数据仓库环境中是常见的。本地索引具有以下优点：

- 可用性增加，因为使数据无效或不可用的动作仅在分区中影响这个分区。
- 分区维护工作简化。当移动表的分区，或者当数据年龄超出分区时，只有关联本地索引分区必须重建或维护。在全局索引中，所有索引分区必须重建或维护。
- 如果一个分区发生**时间点恢复（point-in-time recovery）**，那么索引可以被恢复到恢复的时间（参见“数据文件恢复”在 18-14 页）。整个索引不需要被重建。

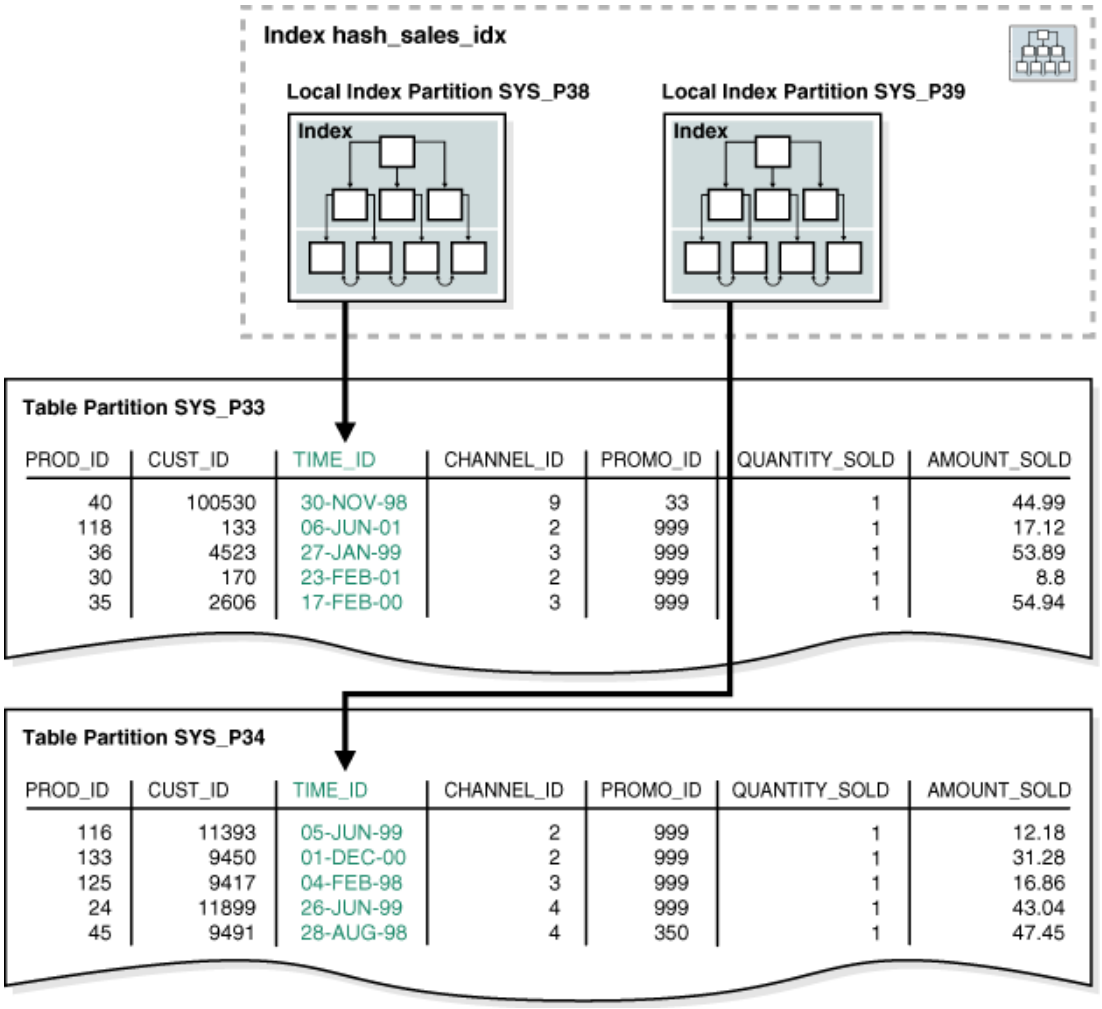
例 4–4 s 显示了分区的 hash_sales 表的创建语句，使用 prod_id 列作为分区键。例 4–5 创建本地分区索引在 hash_sales 表的 time_id 列上。

例 4–5 本地分区索引

```
CREATE INDEX hash_sales_idx ON hash_sales(time_id) LOCAL;
```

在图 4–4 中，hash_products 表有两个分区，所以 hash_sales_idx 有两个分区。每个索引分区都关联一个不同表分区。索引分区 SYS_P38 的索引行在表分区 SYS_P33 中，而索引分区 SYS_P39 的索引行在表分区 SYS_P34 中。

图 4–4 本地索引分区



你不能明确的添加一个分区到本地索引中。相反，只有当你添加一个新的分区到本表中时新分区才添加到本地索引。同样，你不能明确的从一个本地索引中删除一个分区。相反的，只有当你从本表中删除一个分区时本地索引分区才被删除。

像其他的索引，你可以在分区表上创建**位图索引（bitmap index）**。唯一的限制是位图索引必须是本地分区表——它们不能是全局索引。全局位图索引仅支持在非分区表上。

本地前缀和非前缀索引 本地分区索引被分为以下两类：

- 本地前缀索引

  在这种情况下，分区键是在索引定义的前沿上。在例 4–2 页 4-3 中，该表在 time_id 上按照范围分区。在这张表上的本地前缀索引将使用 time_id 作为它列表中的第一列。

- 本地非前缀索引

  在这种情况下，分区键不在索引列列表的前沿并且不根本不需要存在于列表中。在例 4–5 页 4-8 中，索引是本地非前缀的，因为分区键 product_id 不在前沿。

这两种类型的索引都可以获取**分区消除（partition elimination）**（也被称为**分区修剪（partition pruning）**）的优势，当优化器从考虑中排除分区来加速

数据访问时发生。无论一个**查询（query）**能依赖查询**谓词（predicate）**消除分区。使用本地前缀索引的查询通常允许索引分区消除，而使用本地非前缀索引的查询可能不会。

参见：*Oracle 数据库 VLDB 和分区指南* 来了解如何使用前缀和非前缀索引

本地分区索引存储 就像一个表分区一样，一个本地索引分区被存储在它自身的段中。每个段包含总索引数据的一部分。因此，一个由四个分区组成的本地索引不是存储在一个单一段中，而是四个独立段中。

参见：*Oracle 数据库 SQL 语言参考手册* 对于 CREATE INDEX ... LOCAL 的例子

全局分区索引

全局分区索引（global partitioned index）是一个 B-树索引，它是在相关表上创建的独立分区的。单个索引分区可以指向任意或者所有表分区，而在一个本地的分区索引中，一个一对一的比较存在于索引分区和表分区之间。

一般情况下，全局索引对 OLTP 应用是十分有用的，并在快速访问、数据完整性和可用性是非常重要的。在一个 OLTP 系统中，一张表可能按照一个键进行分区，例如，employees.department_id 列，但是应用程序可能需要用许多不同的键来访问数据，例如，按照 employee_id 或 job_id。全局索引可能在这种情况下非常有用。

你可以按照范围方式或者按照散列方式来分区一个全局索引。如果按照范围方式分区，那么数据库在列的列表中从你指定表列的范围值上分区全局索引。如果按照散列方式分区，那么数据库在分区键列的值上使用散列函数来指定行到达的分区。

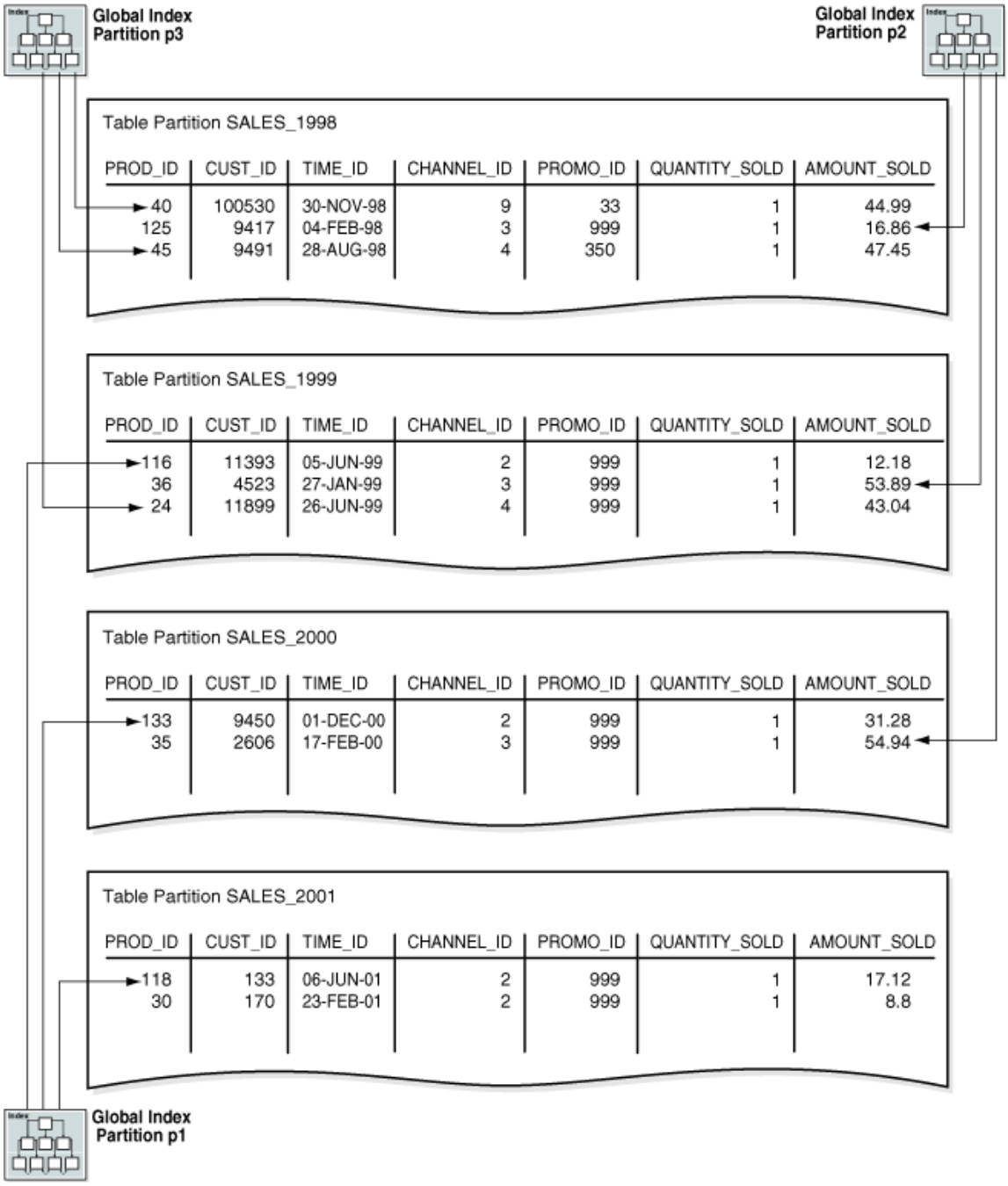
作为一个例子，假设你从例 Example 4–2 在 time_ range_sales 表上创建一个全局分区索引。在这张表中，从 1998 年开始销售的行被存储在一个分区中，从 1999 年开始销售的行在另一个分区中，以此类推。例 4–6 按照范围方式创建了一个全局索引分区在 channel_id 列上。

例 4–6 全局分区索引

```
CREATE INDEX time_channel_sales_idx ON time_range_sales (channel_id)
 GLOBAL PARTITION BY RANGE (channel_id)
 (PARTITION p1 VALUES LESS THAN (3),
 PARTITION p2 VALUES LESS THAN (4),
 PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

如图 4–5 所示，一个全局索引分区可以包含指向多个表分区的条目。索引分区 p1 指向带有 channel_id 为 2 的行，索引分区 p2 指向带有 channel_id 为 3 的行，并且索引分区 p3 指向带有 channel_id 为 4 或 9 的行。

图 4–5 全局分区索引



参见：

- *Oracle 数据库 VLDB 和分区指南* 来了解如何使用全局分区索引
- *Oracle 数据库 SQL 语言参考手册* 对于 CREATE INDEX ... GLOBAL 的例子

视图概述

分区索引-组织表

你可以按照范围方式、列表方式和散列方式分区一个**索引-组织表**（index-organized table） (IOT)。分区对于 IOT 提供改进可管理性、可用性和性能是十分有用的。此外，使用 IOT 的数据磁带可以获得用来划分它们存储数据的能力优势。

注意分区 IOT 的以下特征：

- 分区列必须是主键列的一个子集。
- 二级索引可以是本地和全局分区。
- **溢出**（OVERFLOW）数据段总是与表分区等分区的（equipartitioned）。

Oracle 数据库支持在分区和未分区的索引-组织表上的位图索引。对于创建在索引-组织表上的位图索引需要一张映射表。

参见：“索引-组织表概述”在 3-20 页

视图概述

视图（view）是一张或多张表的逻辑表示。从本质上说，一个视图是一个被存储的查询。一个视图从它基于的表上衍生出它的数据，该表称为**基表**（base tables）。基表可以是表或其他视图。视图上所有的执行操作都会实际影响基表。你可以在使用表的多数地方使用视图。

注意：物化视图使用与标准视图不同的数据结构。参见“物化视图概述”在 4-16 页。

视图使你能够修整数据的展现方式来给不同类型的用户。视图经常被用来：

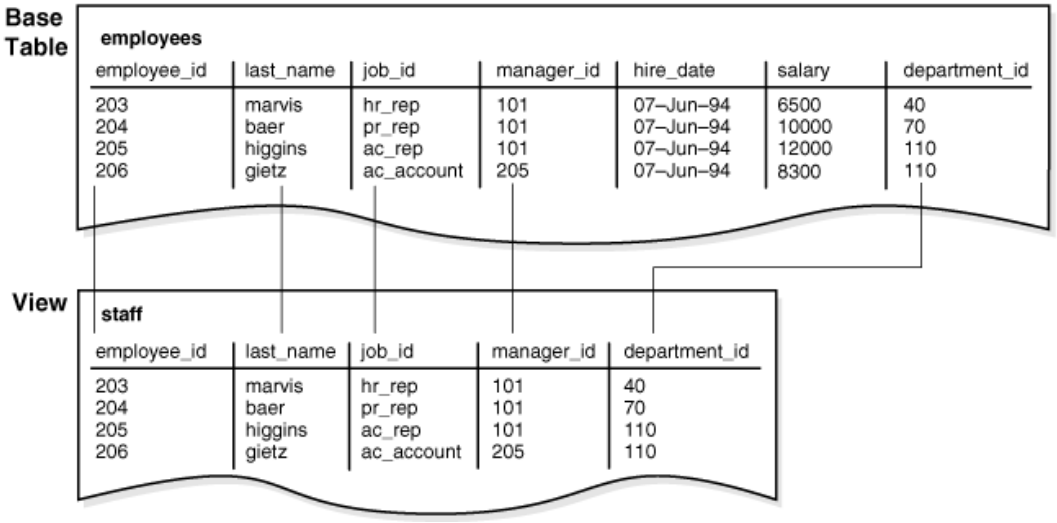
- 通过限制访问预定表的行或列的集合来提供额外的表级安全。
  例如，图 4–6 显示了 staff 视图如何不显示基表 employees 的 salary 或 commission_pct 列。
- 隐藏数据的复杂性
  例如，单个视图可以定义为一个在多表中相关列或行集合的**连接**（join）。然而，视图隐藏了这个信息实际上是从几张表起源的事实。查询也可能会对表中信息进行大量计算。因此，用户可以在不知道如何连接或计算的情况下查询一个视图。
- 当前的数据是源自基表的不同视角
  例如，一个视图的列可以不影响视图所基于的表进行重命名。
- 从变化基表的定义中隔离应用程序
  例如，如果一个视图的定义查询引用了一张四列表的三个列，并且第五列被添加到表中，那么视图的定义不受影响，并且使用视图的所有应用程序不受影响。

举个使用视图的例子，研究 hr.employees 表，其中有几个列和大量的行。为了让用户只看见这些列的五个或者指定的行，你可以创建如下的视图：

```
CREATE VIEW staff AS
 SELECT employee_id, last_name, job_id, manager_id, department_id
 FROM employees;
```

如带有所有的**子查询（subqueries）**，查询定义了一个视图不能包含 FOR UPDATE 子句。图 4–6 形象的说明了名叫 staff 的视图。注意视图仅显示了在基表中的五列。

图 4–6 视图



参见：

- *Oracle 数据库管理员指南* 来了解如何管理视图
- *Oracle 数据库 SQL 语言参考手册* 对于 CREATE VIEW 的语法和语义

视图的特点

不同于一张表，一个视图不分配存储空间，也不包含数据。相反的，一个视图由一个提取或源于该视图所引用的基表数据的查询来定义的。因为视图是基于其他对象的，它不需要存储对于在**数据字典（data dictionary）**中定义的视图查询的存储。

视图依赖于它引用的对象，这是由数据库自动处理的依赖关系。例如，如果你删除并重新创建视图的基表，那么数据库会确定新的基表是否是可以被接受的视图定义。

在视图中的数据操纵

因为视图源于表，所以它们有许多相似之处。例如，一个视图可以包含多达 1000 列，就像一张表。用户可以查询视图，并限制在视图上他们可以执行的 DML。

在视图上执行的操作会影响一些视图的基表，并受到基表的完整性约束和触发器。下面的例子创建了 hr.employees 表的视图：

```
CREATE VIEW staff_dept_10 AS
SELECT employee_id, last_name, job_id,
 manager_id, department_id
FROM employees
WHERE department_id = 10
WITH CHECK OPTION CONSTRAINT staff_dept_10_cnst;
```

定义查询值引用部门为 10 的行。CHECK OPTION 创建带有约束的视图,以便和语句发出反对视图不能选择的行的输出结果。因此，对于在部门 10 的员工的行可以被插入，而不能用于部门 30 的行。

> **参见：***Oracle 数据库 SQL 语言参考手册* 来了解关于子查询在 CREATE VIEW 语句中的限制

在视图中数据是如何访问的

Oracle 数据库在数据字典中通过存储视图的定义视图的查询文本来定义。当你在 SQL 语句中引用一个视图时，Oracle 数据库执行如下的任务：

1. 合并查询（尽可能），针对于查询有定义视图和任何相关视图的视图。

Oracle 数据库优化合并后的查询，如果你发出不带有参考视图的查询。因此，Oracle 数据库可以使用在任何引用的基表的列上索引，无论列是否在视图定义中或者在用户针对的查询视图中被引用。

有时 Oracle 数据库不能合并带有用户查询的视图定义。在这种情况下，Oralce 数据库可能无法使用在引用列上的所有索引。

2. 解析在**共享 SQL 区（shared SQL area）**中的合并语句

Oracle 数据库解析语句，该语句在新的共享 SQL 区中引用了一个视图***除非***不存在包含相似语句的共享 SQL 区。因此，视图提供减少相关共享 SQL 内存使用的益处。

3. 执行 SQL 语句

下面的例子说明当一个视图被查询时数据的访问。假设你创建 employees_view 基于 employees 和 departments 表：

```
CREATE VIEW employees_view AS
 SELECT employee_id, last_name, salary, location_id
 FROM employees JOIN departments USING (department_id)
 WHERE departments.department_id = 10;
```

用户执行 employees_view 中接下来的查询：

```
SELECT last_name
FROM employees_view
WHERE employee_id = 9876;
```

Oracle 数据库合并视图和用户查询来构造下面的查询，它然后执行检索数据：

```
SELECT last_name
```

```
FROM employees, departments
WHERE employees.department_id = departments.department_id
AND departments.department_id = 10
AND employees.employee_id = 9876;
```

参见：

- “优化器概述”在 7-10 页和 *Oracle 数据库性能调优指南* 来了解关于查询优化
- “共享 SQL 区”在 14-16 页

可更新的连接视图

连接视图（join view）被定义为在它的 FROM 子句中有多张表或视图的视图。在例 4–7 中，staff_dept_10_30 视图连接 employees 和 departments 表，仅包括在部门 10 或 30 中的员工。

例 4–7 连接视图

```
CREATE VIEW staff_dept_10_30 AS
SELECT employee_id, last_name, job_id, e.department_id
FROM employees e, departments d
WHERE e.department_id IN (10, 30)
AND e.department_id = d.department_id;
```

可更新连接视图（updatable join view），也被称为**可修改连接视图（modifiable join view）**，设计两张或多张基表或视图并允许 DML 操作。可更新连接视图在顶层的 SELECT 语句的 FROM 子句中包含多表，并且不限制使用 WITH READ ONLY 子句。

要成为本质上可更新的，一个视图必须满足几个标准。例如，一般规则是在一个连接视图上的 INSERT、UPDATE 或 DELETE 操作仅可以一次修改一张基表。下面 USER_UPDATABLE_COLUMNS 数据字典的查询显示了在例 4–7 中创建的视图是可更新的：

```
SQL> SELECT TABLE_NAME, COLUMN_NAME, UPDATABLE
 2 FROM USER_UPDATABLE_COLUMNS
 3 WHERE TABLE_NAME = 'STAFF_DEPT_10_30';
TABLE_NAME COLUMN_NAME UPD
------------------------------ ------------------------------ ---
STAFF_DEPT_10_30 EMPLOYEE_ID YES
STAFF_DEPT_10_30 LAST_NAME YES
STAFF_DEPT_10_30 JOB_ID YES
STAFF_DEPT_10_30 DEPARTMENT_ID YES
```

连接视图的所有可更新列必须映射到键-保留表的列。在一个连接查询中的**键-保留表（key-preserved table）**是一张相关表的每一行最多一次出现在查询的输出中的表。在例 4–7 中，department_id 是 departments 表的主键，所以来自 employees 表的每一行最多出现一次在结果集中，使 employees 为键-保留的。表不是键-保留的，因为每个它的行可能在结果集中出现多次。

参见： *Oracle 数据库管理员指南* 来了解图和更新连接视图

对象视图

正如一个视图是一张虚拟表，一个**对象视图（object view）**是一张虚拟对象表。在视图中的每行是一个**对象（object）**，它是一个**对象类型（object type）**的实例。一个对象类型是用户定义的数据类型。

你可以检索、更新、插入和删除关系数据，如果它被粗处在一个对象类型中。你也可以用对象类型如对象、**参考（REF）**和集合（嵌套表和**可变数组（VARRAY）**）的列来定义视图。

像关系视图，对象视图可以只显示你想让用户看到的数据。例如，对象视图可以显示关于 IT 程序员的数据，但省略关于薪酬的敏感数据。下面的例子创建一个 employee_type 对象，然后 it_prog_view 视图基于这个对象：

```
CREATE TYPE employee_type AS OBJECT
(
 employee_id NUMBER (6),
 last_name VARCHAR2 (25),
 job_id VARCHAR2 (10)
);
/

CREATE VIEW it_prog_view OF employee_type
 WITH OBJECT IDENTIFIER (employee_id) AS
SELECT e.employee_id, e.last_name, e.job_id
FROM employees e
WHERE job_id = 'IT_PROG';
```

对象视图在原型或过渡到面向对象的应用程序中是有用的，因为在视图中的数据可以从关系表取出并访问，犹如表被定义为一个对象表。你可以不将现有表转换为不同的物理结构来运行面向对象应用程序。

参见：

- *Oracle 数据库对象-关系开发者指南* 来了解关于对象类型和对象视图
- *Oracle 数据库 SQL 语言参考手册* 来了解关于 CREATE TYPE 命令

物化视图概述

物化视图(Materialized views)是已存储的或“物理化”成模式对象为前提的查询结果。查询的 FROM 子句可以命名表、视图和物化视图。总之，这些对象被称为**主表（master tables）**（复制术语）或**详细表（detail tables）**（数据仓库术语）。

物化视图用于汇总、计算、复制和分发数据。它们适用于各种计算环境中，如下所示：

- 在数据仓库中，你可以使用物化视图来计算和存储来自聚合函数如求和和平均值产生的数据。

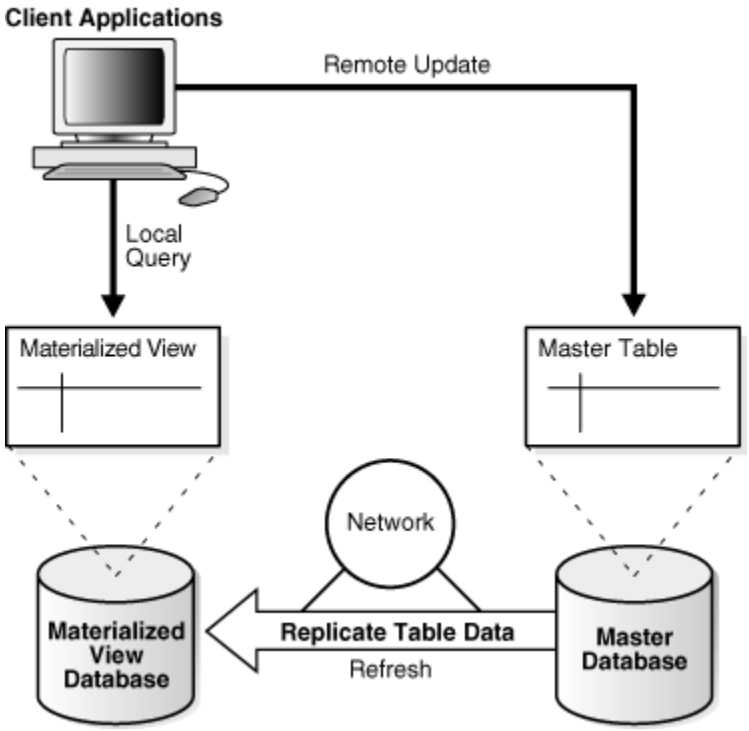
  汇总（summary）是一个由预计算连接和聚合操作并在表中存储结果来减少查询时间的聚合视图。物化视图相当于汇总（见“数据仓库体系结构（基础）”在 17-16 页）。你也可以使用物化视图来计算带或不带聚合的连接。如果兼容性被设置为或更高，那么物化视图对包括过滤器选择的查询是有用的。

■ 在物化视图的**复制（replication）**中，视图包含来自单点的准时的表的完整或部分复制。物化视图在分布式站点复制数据并同步在几个站点执行更新。当数据库不总是连接到网络时，这种复制的方式是适合如销售场景这种环境的。

■ 在移动计算环境中，你可以使用物化视图来下载一个数据子集从中央服务器到移动客户端，并从中央服务器定期刷新，由客户端到中央服务器传播更新。

在复制环境中，物化视图在不同的数据库中共享一张表的数据，称为**主数据库（master database）**。在主站端关联物化视图的表是**主表（master table）**。图 4–7 说明了在一个数据库中的物化视图基于在另一个数据库中的主表。更新的主表复制到物化视图的数据库。

图 4–7 物化视图



参见：

■ “信息共享”在 17-21 页来了解关于通过 Oracle 流的复制

■ *Oracle 数据库 2 天 + 数据复制和集成指南* 和 *Oracle 数据库高级复制* 来了解图和使用物化视图

■ *Oracle 数据库 SQL 语言参考手册* 来了解关于 CREATE MATERIALIZED VIEW 语句

物化视图的特性

物化视图共享一些非物化视图和索引的特性。物化视图在以下方面类似于索引：

■ 它们包含真实的数据并消耗存储空间

■ 在它们的主表改变时，它们可以被刷新。

- 当使用查询重写操作时，它们可以提高 SQL 的执行性能。
- 它们的存在对于 SQL 应用程序和用户是透明的。

物化视图类似于非物化视图，应为它代表在其他表和视图中的数据。不同于索引的是，用户可以直接的使用 SELECT 语句查询物化视图。视图也可以通过 DML 语句被更新，这取决于所需的刷新类型。

下面的例子在 sh 样例模式中创建并填充基于三张主表的物化聚合视图：

```
CREATE MATERIALIZED VIEW sales_mv AS
 SELECT t.calendar_year, p.prod_id, SUM(s.amount_sold) AS sum_sales
 FROM times t, products p, sales s
 WHERE t.time_id = s.time_id
 AND p.prod_id = s.prod_id
 GROUP BY t.calendar_year, p.prod_id;
```

下面的例子删除 sales_mv 的主表 sales 表，然后查询 sales_mv。查询选择数据，因为在行被存储（物化）于主表数据不同的地方。

```
SQL> DROP TABLE sales;
Table dropped.
SQL> SELECT * FROM sales_mv WHERE ROWNUM < 4;
CALENDAR_YEAR PROD_ID SUM_SALES
------------- ---------- ----------
1998 13 936197.53
1998 26 567533.83
1998 27 107968.24
```

物化视图可以被分区。你可以在分区表上定义物化视图和在物化视图上的一个和多个索引。

参见：*Oracle 数据库数据仓库指南* 来了解在数据仓库中如何使用物化视图

物化视图的刷新方法

数据库在物化视图中通过更改它们的主表之后刷新它们来维持数据。刷新方法可以是递进的，被称为**快速刷新（fast refresh）**或**完全刷新（complete refresh）**。当物化视图初始定义为 BUILD IMMEDIATE 时，发生一次完全刷新，除非物化视图引用了一张预先建立的表。刷新涉及对定义的物化视图执行查询。这个过程可能是缓慢的，尤其吐过数据库读和处理大量的数据。

快速刷新无需从头开始重建物化视图。因此，仅处理变更可以在非常短的刷新时间中得出结果。可以按照需求或者固定时间间隔刷新物化视图。另外，在相同数据库中的物化视图，作为其中的主表可以在每当一个事物提交改变主表是进行刷新。

对于物化视图使用快速刷新方法，**物化视图日志（materialized view log）**或**直接加载日志（direct loader log）**来保持改变主表的记录。

物化视图日志是一种记录修改主表数据的模式对象，以便在主表上定义的物化视图可以逐渐刷新。每个物化视图日志都关联单独的主表。物化视图日志存放在相同的数据库和模式中作为其主表。

参见：

- *Oracle 数据库数据仓库指南* 来了解如何刷新物化视图
- *Oracle 数据库高级复制* 来了解关于物化视图日志

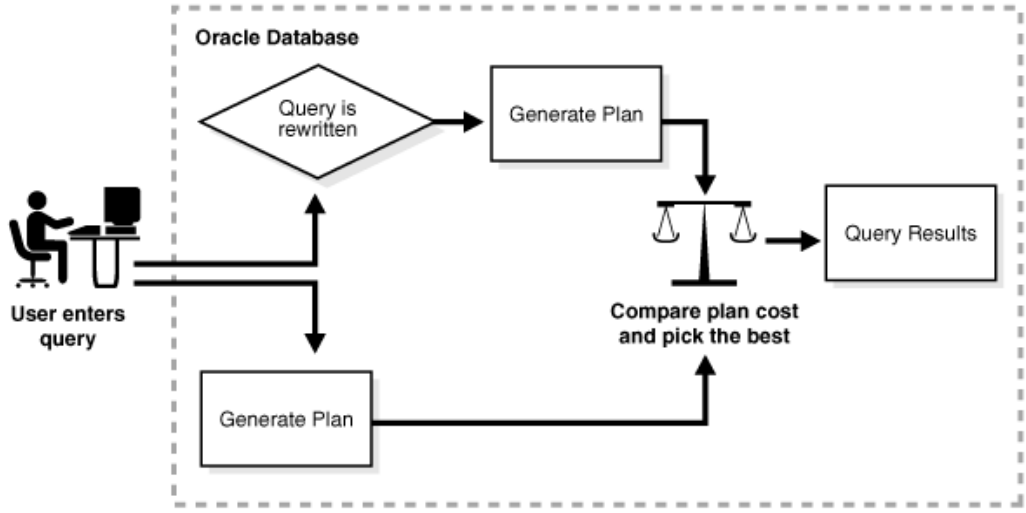
查询重写

查询重写（Query rewrite）是一种优化技术，就主表而言，这种技术将用户写入请求转换为一个语义上等价的请求，包括物化视图。当基表包含大量数据时，计算聚合或**连接（join）**是昂贵且费时的。因为物化视图包含预计算聚合和连接，查询重写可以使用物化视图快速回应查询。

优化查询转换器(optimizer query transformer)透明重写请求到使用的物化视图，请求无需用户干预并且无需在 SQL 语句中引用物化视图。因为查询重写是透明的，物化视图可以被添加或删除，无需在应用程序代码中禁用 SQL。

一般情况下，重写查询来使用物化视图而不是详细表提高了响应时间。图显示了数据库为原始和重写查询生成一个**执行计划（execution plan）**并选择成本最低的计划。

图 4–8 查询重写



参见：

- “优化器概述”在 7-10 页
- *Oracle 数据库数据仓库指南* 来了解图和使用查询重写

序列概述

序列（sequence）是一种来自多用户可以产生唯一整数的模式对象。序列生成器提供一个高度可扩展并且性能良好的方法来为数值数据类型产生代理键。

序列的特性

序列定义要说明的一般信息，如下所示：

- 序列的名称
- 序列是升序还是降序
- 数值之间的间隔
- 数据库是否应在内存中缓存生成序列数值的集合
- 当达到限制时，序列是否应该循环

下面的例子在样例模式 oe 中创建序列 customers_seq。当行被添加到 customers 表中时，应用程序可以使用这个序列来提供 ID 数值。

```
CREATE SEQUENCE customers_seq
START WITH 1000
INCREMENT BY 1
NOCACHE
NOCYCLE;
```

第一次引用 customers_seq.nextval 会返回 1000。第二次返回 1001。随后的每一次引用都返回一个比前一个引用值大 1 的值。

参见：

- *Oracle 数据库 2 天开发者指南* 和 *Oracle 数据库管理员指南* 来了解如何管理序列
- *Oracle 数据库 SQL 语言参考手册* 对于 CREATE SEQUENCE 的语法和语义

并发访问序列

相同的序列生成器可以为多个表产生数值。在这种方式中，数据库可以自动的产生主键并协调键跨越多行或多表。例如，一个序列可以为 orders 表和 customers 表产生主键。

序列生成器在多用户环境下对不带有磁盘 I/O 开销或事务锁而产生的唯一数值是有用的。例如，两个用户同时插入新行到 orders 表中。通过使用序列对 order_id 列产生唯一数值，则用户必须等待其他的进入下一个可用顺序号。序列自动的为每个用户产生正确的数值。

引用序列的每个用户都可以访问他或她的当前序列号，这是在**会话（session）**中最后生成的序列。一个用户可以发出语句来产生新的序列值，或者由会话使用最近产生的当前值。会话中的语句生成一个序列数之后，它仅可用于这个会话。如果在一个被完全回滚的事务中生成并使用时，个别的序列值可能被跳过。

警告：如果你的应用程序需要无间隙的一个数字集合，那么你不能使用 Oracle 序列。你必须在数据库中使用你自己的代码来串行化这些活动。

参见：第 9 章，“数据的并发性和一致性”

维度概述

一个典型的**数据仓库（data warehouse）**有两个重要的组成部分：维度和事实。**维度（dimension）**是在特定的业务问题中使用的任何类别，例如，时间、地理信息、产品、部门和分销渠道。**事实（fact）**是一个事件或实体相关的维度值的部分集合，例如，已售单位和收益。

多维度请求的示例包含以下方面：

- 在提高地理维度的聚合级别时显示横跨所有产品的总销量，从州到乡村再到区域，在 2007 年和 2008 年里。
- 创建一个交叉表格分析按领土分类显示在南美洲的 2007 年和 2008 年我们操作的开支。
- 根据 2008 年汽车产品的销售收入列出在亚洲前 10 名销售代表，并排序他们的佣金。

许多多维度问题需要汇总数据并比较数据集，往往要跨越时间、地理或预算。

创建一个维度允许更广泛的使用查询重写特性。通过透明的重写查询来使用**物化视图（materialized views）**，数据库可以提高查询性能。

参见：“数据仓库和商务智能概述”在 17-14 页

维度的层次结构

维度表（dimension table）是一种定义两列或列集之间层次关系的逻辑结构。维度不含有数据存储分配给它。维度的信息被存放在维度表中，而事实信息被存储在**事实表（fact table）**。

在客户维度内，客户可以上卷到城市、州、乡村、次区域和区域。数据分析在维度层次中通常起始于更高级别，并逐步的下钻，如果情况允许这样分析。

在子级的每个值都关联一个且仅一个在父级的值。层次关系是在层次中一个层次级别到下一个层次级别的**函数依赖（functional dependency）**。

参见：

- *Oracle 数据库数据仓库指南* 来了解关于维度
- *Oracle OLAP 用户指南* 来了解如何创建维度

维度的创建

维度使用 SQL 语句来创建。CREATE DIMENSION 语句指定：

- 多个 LEVEL 子句，在维度中的每个都标识出单列或列集
- 一个或多个 HIERARCHY 子句指明相邻级别之间的父/子关系
- 可选 ATTRIBUTE 子句，每个都标识出一个关联单一级别的额外的列或列集

在样例模式 sh 中，使用下面语句来创建 customers_dim 维度：

```
CREATE DIMENSION customers_dim
 LEVEL customer IS (customers.cust_id)
 LEVEL city IS (customers.cust_city)
 LEVEL state IS (customers.cust_state_province)
 LEVEL country IS (countries.country_id)
 LEVEL subregion IS (countries.country_subregion)
 LEVEL region IS (countries.country_region)
 HIERARCHY geog_rollup (
 customer CHILD OF
 city CHILD OF
 state CHILD OF
 country CHILD OF
 subregion CHILD OF
 region
JOIN KEY (customers.country_id) REFERENCES country )
ATTRIBUTE customer DETERMINES
(cust_first_name, cust_last_name, cust_gender,
 cust_marital_status, cust_year_of_birth,
 cust_income_level, cust_credit_limit)
ATTRIBUTE country DETERMINES (countries.country_name);
```

在维度中的列可以来自同一个表(**非规范化（denormalized）**)或者来自多个表(**全部（fully）**或者**部分规范化（partially normalized）**)。例如，一个规范化时间的维度可以包括一张日期表、一张月份表和一张年份表，并带有连接条件，连接每一天的行到一个月份的行，每个月份的行到一个年份的行。在一个完全非正规化时间维度中，日期、月份和年份列在相同的表中。无论是规范化的还是非规范化的，列之间的层次关系必须在 CREATE DIMENSION 语句中指定。

参见：

- *Oracle 仓库生成器数据模型、ETL 和数据质量指南* 关于维度如何被使用于仓库环境的信息
- *Oracle 数据库 SQL 语言参考手册* 对于 CREATE DIMENSION 语法和语义

同义词概述

同义词(synonym)是一个模式对象的别名。例如，你可以为表和视图、序列、PL/SQL 程序单元、用户定义对象类型或者另一个同义词创建一个别名。因为同义词简单的作为一个别名，它不需要任何存储除了它在数据字典中的定义。

同义词可以为数据库用户简化 SQL 语句。同义词也对隐藏底层模式对象的标识和位置非常有用。

如果底层对象必须被重命名或者移动，那么只有同义词必须被重新定义。基于同义词的应用程序可以无修改的继续工作。

你可以同时创建私有和公有同义词。**私有（private）**同义词是在一个指定用户的模式对象中，谁有将它的可用性给其他用户的控制权限。**公有（public）**同义词是由名叫 PUBLIC 的用户组所有且可以由每个数据库用户访问。

在例 4–9 中，数据库管理员创建 hr.employees 表的公有同义词 people。然后用户连接到 oe 模式并通过同义词统计在表中引用的行数。

例 4–8 公有同义词

```
SQL> CREATE PUBLIC SYNONYM people FOR hr.employees;
Synonym created.
SQL> CONNECT oe
Enter password: password
Connected.
SQL> SELECT COUNT(*) FROM people;
 COUNT(*)
----------
 107
```

请谨慎使用公有同义词，因为它们是数据库整合更加困难。如例 4–9 所示，如果另一位管理员试图创建公有同义词 people，那么会创建失败，因为在数据库中只能有一个公有同义词 people。公有同义词的过度使用会导致应用程序之间的命名空间冲突。

例 4–9 公有同义词

```
SQL> CREATE PUBLIC SYNONYM people FOR oe.customers;
CREATE PUBLIC SYNONYM people FOR oe.customers
 *
ERROR at line 1:
ORA-00955: name is already used by an existing object
SQL> SELECT OWNER, SYNONYM_NAME, TABLE_OWNER, TABLE_NAME
 2 FROM DBA_SYNONYMS
 3 WHERE SYNONYM_NAME = 'PEOPLE';
OWNER SYNONYM_NAME TABLE_OWNER TABLE_NAME
---------- ------------ ----------- ----------
PUBLIC PEOPLE HR EMPLOYEES
```

同义词本身是不安全的。当你对同义词授予对象权限时，你实际上对底层对象授予了权限。在 GRANT 语句中，同义词的作用仅作为对象的一个别名。

参见：

- *Oracle 数据库管理员指南* 来了解如何管理同义词
- *Oracle 数据库 SQL 语言参考手册* 对于 CREATE SYNONYM 的语法和语义