# ЛАБОРАТОРНАЯ РАБОТА №17

*Тема:* Создание и использование представлений в языке SQL

*Цель:* Пробрести навыки создания и использования представлений в языке SQL.

**Теоретическией сведения**

Представление (view), или виртуальная таблица в базе данных Oracle Database — это специфический образ таблицы или набора таблиц, определенный оператором SELECT. Представление не существует физически как обычная таблица, являющаяся частью табличного пространства. Фактически представление создает виртуальную таблицу или подтаблицу только с теми строками и/или столбцами, которые нужно показать пользователю.

Представление Oracle — результат хранимого запроса, поэтому в словаре данных сохраняется только определение представления. При экспорте базы данных Oracle можно видеть предложение “exporting views” (“экспорт представлений”), но под этим имеется в виду только определения представлений, а не физические объекты.

При условии, что пользователь имеет соответствующие права доступа к лежащим в основе представления таблицам, можно запрашивать представления или даже модифицировать, удалять либо добавлять данные с использованием операторов UPDATE, DELETE и INSERT. Например, если выдать только привилегию INSERT для базовой таблицы, на которой определено представление, то этот пользователь сможет только вставлять строки в таблицу, но не сможет ни выбирать, ни вставлять, ни удалять строки.

Кончено же у читателя моего блога встает самый главный и насущный вопрос. Зачем все же нужны представления в базе данных Oracle? Представления Oracle используются в приложениях по нескольким причинам, включая перечисленные ниже:

- уменьшение сложности;
- повышение безопасности;
- повышение удобства;
- переименование столбцов таблицы;
- настройка данных для пользователей;
- защита целостности данных.

Представления создаются с помощью оператора SQL, описывающего композицию представления. При вызове представления выполняется запрос, по которому оно определено, и затем возвращается результат. Запрос, адресованный к представлению,выглядит в точности как обычный запрос, но база данных преобразует его в идентичный запрос к лежащим в его основе таблицам. Чтобы создать представление в своей схеме, необходимо иметь системную привилегию CREATE VIEW, а чтобы создать представление в любой схеме, а не только в собственной, понадобится системная привилегия CREATE ANY VIEW. Вдобавок нужно либо владеть лежащими в основе таблицами, либо иметь права на операции SELECT, INSERT, UPDATE и DELETE со всеми таблицами, на которых определено представление. Представление можно использовать для добавления к таблице мер безопасности уровня столбца или уровня значения. Безопасность уровня столбца обеспечивается созданием представлений, которые дают доступ лишь к избранным столбцам таблицы. Безопасность уровня значений включает применение конструкции WHERE в определении представления, которое отображает лишь избранные строки базовых таблиц. Чтобы использовать представление, пользователю нужны привилегии для доступа к нему, а не к базовым таблицам, на которых оно определено.

Следующий оператор создает представление по имени MY_EMPLOYEES, которое выдает информацию только о сотрудниках, подчиненных конкретному менеджеру:

```
CREATE VIEW my_employees AS
SELECT employee_id, first_name, last_name, salary
```

```
FROM employees
WHERE manager_id=122;
```

Совет. Добавление к оператору CREATE VIEW конструкции WITH READ ONLY гарантирует, что пользователи смогут только осуществлять выборку данных из представления. Это означает, что пользователи не смогут модифицировать представление и тем самым неявно обновлять, вставлять или удалять строки базовых таблиц. В противном случае по умолчанию Oracle позволяет обновлять представление.

Теперь менеджер с идентификатором 122 сможет опрашивать представление my_employees, как если бы это была обычная таблица, но содержащая лишь сотрудников, подчиненных этому менеджеру. В листинге показан вывод, полученный в результате запроса к представлению.

```
SELECT * FROM my_employees;
EMPLOYEE_ID   FIRST_NAME  LAST_NAME  SALARY
-----------   ----------  ---------  ------
        133   Jason       Mallin     3300
        134   Michael     Rogers     2900
        135   Ki          Gee        2400
        136   Hazel       Philtanker 2200
        188   Kelly       Chung      3800
        189   Jennifer    Dilly      3600
        190   Timothy     Gates      2900
        191   Randall     Perkins    2500
```

При создании представления в конструкции FROM можно специфицировать несколько базовых таблиц или даже других представлений. Созданные подобным образом представления называются соединенными (joined views), и следующий пример демонстрирует создание такого представления:

```
 CREATE VIEW view_1 AS

SELECT ename, empno, job,dname
FROM employee, dept
WHERE employee.deptno IN (10, 60)
AND employee.deptno = DEPT.DEPTNO;
```

Хотя представления используются в основном для запросов, при некоторых обстоятельствах их можно также применять в командах INSERT, DELETE и UPDATE. Например, допускается выполнять операции DML над представлениями, которые не имеют в своем определении конструкций GROUP BY, START WITH или CONNECT BY, либо каких-то под-запросов в своей конструкции SELECT. Однако поскольку представление в действительности не существует как отдельная физическая сущность, на самом деле происходит модификация данных лежащих в его основе таблиц, и само представление будет, таким образом, субъектом тех же ограничений целостности, что и таблицы, на которых оно основано. В следующем примере показано, как вставлять строки в представление по имени sales_view, которое зависит от таблицы employees.

```
SQL> INSERT INTO sales_view
VALUES (1234, 'ALAPATI', 99);
```

Приведенный оператор вставляет новую строку в базовую таблицу по имени employees. Обновления, удаления и вставки в представления подчиняются нескольким ограничениям. Например, при использовании ограничения CHECK при создании представления не получится вставить строку или обновить базовую таблицу этой строкой,если данное представление не может извлечь упомянутую строку из базовой таблицы.

Уничтожается представление с помощью команды DROP VIEW, как показано ниже:

```
SQL> DROP VIEW my_employees;
View dropped.
```

Вместо уничтожения и пересоздания представления можно воспользоваться конструкцией OR REPLACE для переопределения представления, например:

```
SQL> CREATE OR REPLACE VIEW view1 AS
SELECT empno, ename, deptno
FROM employee
WHERE deptno=50;
```

Если в базе данных есть другие представления, которые зависят от заменяемого, они станут недействительными. Недействительные

представления следует перекомпилировать с помощью оператора ALTER VIEW. Если программная единица PL/SQL, такая как процедура или функция, зависит от представления, то база данных может сделать ее недействительной, если изменения в новом представлении касаются количества столбцов или их имен либо же их типов данных.

## Использование материализованных представлений

Всякий раз, когда нужен доступ к представлению, Oracle должен выполнить запрос, по которому определено представление, и вернуть результат. Этот процесс наполнения представления называется разрешением представления (view resolution) и он повторяется при каждом обращении пользователя к представлению. Если вы имеете дело с представлениями с множеством конструкций JOIN и GROUP BY, то этот процесс разрешения представления может потребовать очень длительного времени. Если нужно часто обращаться к представлению, будет весьма неэффективно каждый раз повторять разрешение представления.

Материализованные представления Oracle предлагают выход из этого затруднения. Упомянутые представления можно воспринимать как специализированные представления, в отличие от обычных представлений, имеющие физическое воплощение. Они занимают место и требуют хранения подобно обычным таблицам. Материализованные представления можно даже секционировать и при необходимости создавать на них индексы.

На заметку! Представление всегда вычисляется на лету, и его данные не хранятся отдельно от таблиц, на которых оно определено. Таким образом, запросы, использующие представления, по определению гарантированно вернут самые свежие данные. Материализованные представления в базе данных Oracle Database, с другой стороны, являются статическими объектами, которые наследуют свои данные от лежащих в их основе базовых таблиц. Если вы будете обновлять свои материализованные представления

нечасто, то данные в них могут устареть по отношению к данным таблиц, на которых они основаны.

Традиционно хранилища данных и прочие крупные базы данных для выполнения своей работы всегда нуждались в итоговых или агрегатных таблицах. Определение таких итоговых таблиц и постоянное поддержание их в актуальном состоянии — непростая задача. При каждом добавлении данных к таблице деталей необходимо вручную обновлять итоговые таблицы и их индексы. Материализованные представления Oracle предлагают способ упрощения управления итоговой информацией в крупных базах данных. Материализованные представления в таких средах называются также итогами (summaries), поскольку хранят итоговые данные.

В качестве источника для материализованного представления могут служить таблицы, представления, а также другие материализованные представления. Исходные таблицы называются главными таблицами (master tables), а в средах хранилищ данных их часто также называют таблицами деталей. При создании материализованного представления Oracle автоматически создает внутреннюю таблицу для хранения данных этого материализованного представления. Таким образом, материализованное представление будет занимать физическое место в базе данных, в то время как обычное представление — нет, поскольку последнее является всего лишь выводом запроса SQL.

Над материализованными представлениями можно делать следующие действия:

- создавать индексы на материализованном представлении;
- создавать материализованное представление на секционированной таблице;
- секционировать само материализованное представление.

Совет. Индекс для доступа к материализованному представлению можно использовать непосредственно, как это делается в отношении таблицы. Аналогично, можно также обращаться к материализованному представлению непосредственно в операторе INSERT, UPDATE или DELETE. Однако, в Oracle не рекомендуют поступать подобным образом; напротив, следует позволить стоимостному оптимизатору Oracle (Cost Based Optimizer — CBO) принять решения относительно необходимости переписать обычные запросы, что обеспечит возможность воспользоваться преимуществами материализованного представления. Если план выполнения, применяющий материализованное представление, имеет меньшую стоимость доступа по сравнению с прямым обращением к таблицам, то Oracle автоматически использует его.

В материализованном представлении допустимы различные типы агрегации, вроде SUM, COUNT(*), AVG, MIN и MAX. В определении материализованного представления так-же можно использовать соединения множества таблиц.

Создание материализованного представления очень просто, но его оптимизация может оказаться довольно сложной. Оптимизация материализованного представления включает как проверку того, переписывает ли стоимостной оптимизатор Oracle запросы пользователя для использования ранее созданного материализованного представления, так и поддержку данных материализованного представления в актуальном состоянии. Давайте кратко рассмотрим эти два аспекта оптимизации материализованных представлений.

## Переписывание запросов

В крупных базах данных Oracle с интенсивными действиями, затратными по времени и вычислительной мощности процессоров, такими как соединение таблиц и использование агрегатных функций вроде SUM, материализованные представления ускоряют запросы. Материализованные

представления обеспечивают более быстрое выполнение запросов за счет перерасчета и хранения результатов дорогостоящих соединений и агрегатных операций. Прелесть материализованных представлений Oracle заключается в том, что при их создании можно указать, что база данных должна автоматически обновлять материализованные представления, когда происходят изменения в положенных в их основу таблицах. Материализованные представления полностью прозрачны для пользователей. Если пользователи пишут запросы с обращением к лежащим в основе таблицам, то Oracle автоматически переписывает их для использования материализованных представлений, и такая техника оптимизации запросов называется переписыванием запроса (query rewrite). Стоимостной оптимизатор Oracle автоматически распознает необходимость в переписывании запроса для использования материализованного представления вместо исходных таблиц, если оценочная стоимость такого запроса оказывается ниже. Под стоимостью запроса здесь подразумевается объем ввода-вывода, а также затраты времени процессора и памяти, связанные с обработкой SQL-запроса. Сложные соединения таблиц обходятся в этом смысле дорого, а применение материализованных представлений позволяет использовать уже сохраненную информацию в предварительно вычисленном виде, и запросы требуют гораздо меньше ресурсов и потому выполняются намного быстрее.

Прием автоматической оптимизации с переписыванием запроса лежит в основе применения материализованных представлений. Параметр инициализации QUERY_REWRITE_ENABLED позволяет включать и отключать это средство на глобальном уровне.

Этот параметр может принимать следующие значения.

- *FALSE*. База данных не переписывает никакие запросы.
- *TRUE*. База данных сравнивает стоимость запроса с переписыванием и без, и выбирает наиболее дешевый метод.

- *FORCE*. База данных всегда переписывает запрос, не оценивая стоимости. Используйте вариант *FORCE*, если уверены, что это даст выигрыш во времени получения ответа.

Значением по умолчанию для этого параметра является TRUE, как в ситуации, если установить QUERY_REWRITE_ENABLED в 10.0.0 и выше (значение равно FALSE, если установить QUERY_REWRITE_ENABLED в 9.2.0 и ниже); это означает, что Oracle автоматически использует средство переписывания запроса. Когда упомянутый параметр установлен в TRUE, Oracle оценит стоимость запроса в исходном виде и в переписанном,и выберет вариант с минимальной стоимостью. Включение переписывания запросов действует на уровне системы, т.е. для всей базы данных.

Значение FORCE для параметра QUERY_REWRITE_ENABLED должно специфицироваться, только если есть абсолютная уверенность, что это принесет выгоду. Чтобы разрешить переписывание запроса для определенного материализованного представления, необходимо явно указать конструкцию ENABLE QUERY REWRITE при создании материализованного представления.

**Подсказка Rewrite_or_Error**

Предположим, что после создания нового материализованного представления обнаружено, что нужные запросы не переписываются, и преимущества нового материализованного представления не задействуются. Если выполнение запроса без материализованного представления требует слишком много времени, можно заставить Oracle прекратить выполнение запроса без материализованного представления. Чтобы заставить Oracle генерировать ошибку вместо выполнения непереписанного запроса, используется подсказка (создаваемая пользователем директива, которая служит указанием для стоимостного оптимизатора; это средство детально

рассматривается в главе 19).Подсказка называется REWRITE_ON_ERROR и применяется так:

```
SQL> SELECT /*+ REWRITE_OR_ERROR */
prod_id
SUM(quantity_sold) AS sum_sales_qty
FROM sales_data
GROUP BY prod_id
SQL>
```

Если запрос не переписывается, вы увидите следующую ошибку:

```
ORA-30393: a query block in the statement did not
write.
блок запроса в операторе не был переписан.
```

После получения такой ошибки с помощью процедуры DBMS_MVIEW.EXPLAIN_REWRITE можно узнать, почему запрос не переписывается, и решить проблему так, чтобы он был переписан и, таким образом, преимущества материализованного представления были задействованы.

**Целостность при переписывании**

После настройки переписывания запроса Oracle по умолчанию использует только свежие данные из материализованных представлений. Затем он использует только ограничения первичного, уникального или внешнего ключа типа ENABLED VALIDATED. Параметр инициализации QUERY_REWRITE_INTEGRITY задает поведение оптимизатора в этом отношении. Поведение по умолчанию известно как режим ENFORCED. Кроме этого режима параметр QUERY_REWRITE_INTEGRITY может принимать еще два значения.

TRUSTED. В этом режиме оптимизатор принимает во внимание несколько отношений помимо тех, что приняты в режиме ENFORCED. Так, например, оптимизатор принимает наряду с декларируемыми и принудительные отношения, но не ограничения первичного или уникального ключа ENABLED VALIDATED. Поскольку вы позволяете оптимизатору

принимать отношения на веру (не принудительно), то большинство запросов могут быть подвергнуты переписыванию.

STALE_TOLERATED. Оптимизатор будет принимать свежие и старые данные до тех пор, пока они действительны. Конечно, в этом режиме перепишется больше запросов, но вы также рискуете получить некорректные результаты, если старые данные неточно представляют истинную природу текущей таблицы.

**Обновление данных материализованного представления**

Поскольку материализованное представление определяется на основе главных таблиц, при изменении данных этих таблиц материализованное представление устаревает. Чтобы справиться с этой проблемой, материализованные представления обновляются, синхронизируя их с содержимым главных таблиц. В следующих разделах описаны опции обновления материализованных представлений.

**Режим обновления**

При обновлении можно выбирать между режимами ON COMMIT и ON DEMAND.

- *ON COMMIT*. В этом режиме при всякой фиксации изменений данных в главных таблицах материализованное представление обновляется автоматически, отражая эти изменения.
- *ON DEMAND*. В этом режиме для обновления материализованного представления потребуется выполнить процедуру типа *DBMS_MVIEW.REFRESH*.

По умолчанию принимается режим ON DEMAND.

**Тип обновления**

Доступен выбор одного из следующих четырех типов обновлений.

- *COMPLETE*. Эта опция обновления полностью заново вычисляет запрос, лежащий в основе материализованного представления. Таким образом, материализованное представление, на создание которого ушло 12 часов, потребует почти такого же времени для перестройки. Очевидно, что не особенно эффективно использовать эту опцию при каждом обновлении, удалении или вставке данных в таблицы.
- *FAST REFRESH*. Для реализации механизма быстрого обновления Oracle использует журнал материализованного представления для регистрации всех изменений в главных таблицах. Затем журнал материализованного представления применяется для обновления этого материализованного представления. Упомянутый журнал представляет собой таблицу, основанную на ассоциированном материализованном представлении. Каждая из таких таблиц, соединенная с материализованным представлением, нуждается в собственном журнале материализованного представления, чтобы фиксировать изменения в таблицах. Для быстрого обновления материализованного представления Oracle также может использовать данные из операций обслуживания разделов или загрузки данных, выполненной с применением метода загрузки в прямом режиме SQL*Loader.
- *FORCE*. В случае выбора этой опции Oracle попытается применить механизм быстрого обновления (fast refresh). Если по некоторым причинам он не может быть использован, применяется метод полного обновления.
- *NEVER*. Эта опция никогда не обновляет материализованное представление. Очевидно, что это неподходящий вариант для материализованных представлений, чьи главные таблицы со временем подвергаются серьезным изменениям.

Типом обновления по умолчанию является FORCE.

**Использование пакета DBMS_MVIEW**

Даже после того, как вы специфицируете механизм обновления запроса, стоимостной оптимизатор Oracle не всегда сможет автоматически переписать запрос, и обратится к главным таблицам вместо материализованного представления. Таким образом, даже несмотря на наличие материализованного представления, оптимизатор игнорирует его, сводя на нет смысл создания и обслуживания материализованного представления. Оптимизатор Oracle поступает так потому, что некоторые условия для переписи запросов могут быть не выполнены. Для диагностики этой и прочих проблем материализованного представления служит поставляемый Oracle пакет DBMS_MVIEW.

Процедуры пакета DBMS_MVIEW используются следующим образом.

- Процедура *EXPLAIN_MVIEW* применяется для того, чтобы увидеть, какие типы переписывания запросов возможны.
- Процедура *EXPLAIN_REWRITE* служит для определения, почему определенный запрос не переписан с использованием материализованного представления.
- Процедура *TUNE_MVIEW* используется для включения переписывания запроса. Эта процедура подскажет, как следует изменить материализованное представление, чтобы сделать его доступным для переписывания запросов. Процедура *TUNE_MVIEW* также сообщит, каким образом удовлетворить требования быстро обновляемого материализованного представления. Процедура примет ввод и построит сценарий создания материализованного представления (вместе со всеми необходимыми журналами материализованного представления), готового к реализации.

**Создание материализованных представлений**

В этом разделе будет показано, как создать базовое материализованное представление с использованием некоторых опций, описанных в предыдущих разделах.

Для ввода в действие материализованного представления необходимо выполнить следующие три шага, хотя само его создание достаточно просто.

1. Выдать необходимые привилегии.
2. Создать журнал материализованного представления (предполагая использование опции обновления FAST).
3. Создать материализованное представление.

### Выдача необходимых привилегий

Первым делом потребуется выдать необходимые привилегии пользователю, создающему материализованные представления. Главные привилегии — это те, что позволяют создавать материализованное представление. Вдобавок необходимо выдать пользователю привилегию QUERY REWRITE, используя для этого либо привилегию GLOBAL QUERY REWRITE, либо специфические привилегии QUERY REWRITE для каждого объекта, не являющегося частью пользовательской схемы. Ниже приведены операторы GRANT, которые позволяют пользователю создавать материализованное представление в его схеме:

```
GRANT CREATE MATERIALIZED VIEW TO salapati;
GRANT QUERY REWRITE TO salapati;
```

В дополнение, если пользователь еще не имеет их, потребуется выдать права на создание таблиц с помощью следующего оператора GRANT:

```
GRANT CREATE ANY TABLE TO salapati;
```

Если пользователь не владеет никакими главными таблицами, являющимися частью определения материализованного представления, необходимо выдать ему привилегию на SELECT в отношении этих индивидуальных таблиц или же сделать так:

```
GRANT SELECT ANY TABLE TO salapati;
```

### Создание журнала материализованного представления

Давайте включим механизм быстрого обновления для материализованного представления. В большинстве случаев для этого необходимо создать журнал материализованного представления. Разумеется, это потребует создания двух журналов материализованного представления, фиксирующих изменения в двух главных таблицах, которые станут основой нашего материализованного представления.

Для использования механизма быстрого обновления материализованного представления сначала нужно создать журналы материализованного представления для каждой из таблиц — частей этого материализованного представления. В нашем случае это таблицы products и sales. В дополнение необходимо специфицировать конструкцию ROWID в операторе CREATE MATERIALIZED VIEW LOG. Также следует перечислить все столбцы, упоминаемые в материализованном представлении, и предусмотреть конструкции SEQUENCE и INCLUDING NEW VALUES, например:

```
SQL> CREATE MATERIALIZED VIEW LOG
ON products WITH SEQUENCE, ROWID
(prod_id, prod_name, prod_desc, prod_subcategory,
prod_subcategory_desc,
prod_category, prod_category_desc, prod_weight_class,
prod_unit_of_measure,   prod_pack_size,   supplier_id,
prod_status,
prod_list_price, prod_min_price)
INCLUDING                NEW                VALUES;

SQL> CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id,
quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Этот пример демонстрирует создание двух журналов материализованного представления для фиксации изменений в главных таблицах products и sales. В следующем разделе будет показано, как создать само материализованное представление.

**Создание материализованного представления**

Теперь все готово для создания материализованного представления. В примере, показанном в листинге 7.17, с помощью конструкции FAST REFRESH специфицируется механизм обновления материализованного представления.

Совет. Если в базе данных уже есть таблицы, содержащие некоторого рода агрегаты или итоговые результаты, можно воспользоваться оператором CREATE MATERIALIZED VIEW с конструкцией ON PREBUILT TABLE для регистрации имеющейся итоговой таблицы в качестве материализованного представления.

```
SQL CREATE MATERIALIZED VIEW product_sales_mv

TABLESPACE test1
STORAGE (INITIAL 8k NEXT 8k PCTINCREASE 0)
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS    SELECT    p.prod_name,    SUM(s.amount_sold)    AS
dollar_sales,
COUNT(*) AS cnt, COUNT(s.amount_sold) AS cnt_amt
FROM sales s, products p
WHERE s.prod_id = p.prod_id GROUP BY p.prod_name;GROUP
BY p.prod_name;
SQL>
```

Рассмотрим некоторые важные конструкции оператора CREATE MATERIALIZED VIEW.

- *BUILD IMMEDIATE* немедленно наполняет материализованное представление; эта опция принята по умолчанию. Альтернатива заключается в использовании опции *BUILD DEFERRED*, которая в действительности загружает материализованное представление данными позднее, в указанное время.
- *REFRESG FAST* специфицирует, что материализованное представление должно использовать метод обновления *FAST*, что для фиксации всех изменений главных таблиц требует наличия двух журналов

материализованных представлений, которые были созданы на предыдущем шаге. Часть *COMMIT* конструкции *REFRESH* указывает на то, что все зафиксированные изменения главных таблиц распространялись на материализованное представление немедленно после фиксации этих изменений.

- *ENABLE QUERY REWRITE* означает, что оптимизатор Oracle прозрачно перепишет все запросы для использования материализованных представлений вместо лежащих в основе главных таблиц.
- Подзапрос *AS* определяет материализованное представление. Oracle сохранит вывод этого подзапроса в материализованном представлении, которое вы создаете.Здесь допустим любой подзапрос SQL.
- Последние четыре строки кода содержат действительный запрос, определяющий материализованное представление; они извлекают вывод из главных таблиц и делают его частью материализованного представления.

На заметку! Из-за ограниченности объема книги здесь был представлен только простейший пример создания материализованного представления и его журналов. В действительности, чтобы иметь возможность создавать такие объекты, может понадобиться удовлетворить дополнительным требованиям. Например, чтобы иметь возможность создания быстро обновляемых материализованных представлений с журналами, вы должны удовлетворять специальным требованиям. Полный список этих требований можно найти в руководствах Oracle (в частности, в Data Warehousing Guide).

Обратите внимание на две возможности включения переписывания запросов: указание конструкции ENABLE QUERY REWRITE при создании материализованного представления (см. листинг 7.16) или применение оператора ALTER MATERIALIZED VIEW с этой конструкцией после того, как материализованное представление уже существует.

Для просмотра предложенного плана выполнения запроса вместо процедуры EXPLAIN_REWRITE из пакета DBMS_MVIEW можно воспользоваться инструментом EXPLAIN PLAN. При этом EXPLAIN PLAN не должен отображать никаких ссылок на лежащие в основе базовые таблицы. Если запрос был действительно переписан с использованием нового материализованного представления, там должно присутствовать обращение к этому представлению.

Совет. Соберите статистику оптимизатора (см. главу 19) для материализованного представления сразу после его создания. Это поможет Oracle оптимизировать процесс переписывания запросов.

Если материализованное представление не нужно, можно уничтожить его с помощью оператора DROP MATERIALIZED VIEW:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

Порядок выполнения работы

1.Создать в облачной системе apex.oracle.com Представление, которое объединяет информацию о работниках из разных таблиц.

```
CREATE OR REPLACE VIEW emp_details_view
  (employee_id,
   job_id,
   manager_id,
   department_id,
   location_id,
   country_id,
   first_name,
   last_name,
   salary,
   commission_pct,
   department_name,
   job_title,
   city,
   state_province,
   country_name,
   region_name)
AS SELECT
  e.employee_id,
```

```
  e.job_id,
  e.manager_id,
  e.department_id,
  d.location_id,
  l.country_id,
  e.first_name,
  e.last_name,
  e.salary,
  e.commission_pct,
  d.department_name,
  j.job_title,
  l.city,
  l.state_province,
  c.country_name,
  r.region_name
FROM
  employees e,
  departments d,
  jobs j,
  locations l,
  countries c,
  regions r
WHERE e.department_id = d.department_id
  AND d.location_id = l.location_id
  AND l.country_id = c.country_id
  AND c.region_id = r.region_id
  AND j.job_id = e.job_id
WITH READ ONLY
```

2. Проверить работоспособность созданного представления. С этой целью составить несколько запросов к БД, основанных на использовании созданного представления.

3. Заменить в представлении механизм объединения таблиц на более современный, основанный на использовании ключевого слово JOIN.

4. Переформатировать создание представления, таким образом, что использовались параметры в операторе создания представления.

5. Создать БД Order Entry – как дополнение к БД HR

Для выполнить следующие шаги

А) Выполнить оператор

CREATE OR REPLACE TYPE phone_list_typ

AS VARRAY(5) OF VARCHAR2(25);

Б) Выполнить оператор

```
CREATE OR REPLACE TYPE cust_address_typ
  AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
```

В) Выполнить скрипт 1_.sql

Г) Выполнить оператор

```
CREATE OR REPLACE TRIGGER insert_ord_line
  BEFORE INSERT ON order_items
  FOR EACH ROW
  DECLARE
    new_line number;
  BEGIN
    SELECT (NVL(MAX(line_item_id),0)+1) INTO new_line
      FROM order_items
      WHERE order_id = :new.order_id;
    :new.line_item_id := new_line;
  END;
```

Д) Выполнить скрипт 2_.sql

Е) Выполнить скрипт 3_.sql

Ж) Выполнить скрипт 4_.sql

З) Выполнить скрипт 5_.sql

И) Выполнить скрипт 6_.sql

К) Выполнить скрипт 7_.sql

Л) Выполнить скрипт 8_.sql

6. Проверить корректность создание БД

7. Создать представление

```
CREATE OR REPLACE VIEW products
AS
SELECT i.product_id
,      d.language_id
,      CASE WHEN d.language_id IS NOT NULL
           THEN d.translated_name
           ELSE TRANSLATE(i.product_name USING NCHAR_CS)
       END    AS product_name
,      i.category_id
,      CASE WHEN d.language_id IS NOT NULL
           THEN d.translated_description
           ELSE TRANSLATE(i.product_description USING NCHAR_CS)
       END    AS product_description
,      i.weight_class
,      i.warranty_period
,      i.supplier_id
,      i.product_status
,      i.list_price
,      i.min_price
,      i.catalog_url
FROM   product_information  i
,      product_descriptions d
WHERE  d.product_id  (+) = i.product_id
AND    d.language_id (+) = sys_context('USERENV','LANG');
```

И проверить его работоспособность

8. Создать представление для вывода товаров с указанием их количества, находящихся на складе в Сиднее

9. Составить представление для вывода,категории товаров, количесвта на складе, минимальная цена, максимальная цена.

Содержание отчета.

Название работы

1. Цель работы
2. Листинги созданных представлений
3. Листинги запросов, основанных на использовании представлений, и результатов их работы.
4. Выводы

Контрольные вопросы

1. Объяснить роль фразы **`WITH READ ONLY`** в первом представлении из порядка выполнения работы.
2. Описать назначение механизма представлений
3. Можно изменить данные в таблицах с использованием представлений?
4. Что такое материализованные представления? В чем их отличие от обычных (не материализованных) представлений?

Приложение

**OE (размещение заказов)**

Компания продает различные товары - компьютерное оборудование и программное обеспечение, музыкальные товары, одежду, рабочий ручной инструмент. Компания хранит и обрабатывает информацию об этих товарах – идентификационный номер товара, категорию, к которой относится товар, вводе заказов, весовая группа (для организации доставки), гарантийный период (если имеется), статус доступности товара, цена по каталогу, минимальная цена, по которой товар может быть продан, URL-ссылка на информацию производителя. Обо всех товарах также записывается учетная информация, в том числе, склад, где есть товар и его количество, имеющееся в наличии. Поскольку товары продаются по всему миру, компания хранит и обрабатывает названия и описания товаров на нескольких языках.

Для обслуживания покупателей компания содержит товарные склады в разных районах. Каждый склад имеет идентификационный номер, название, описание объекта и идентификатор места расположения.

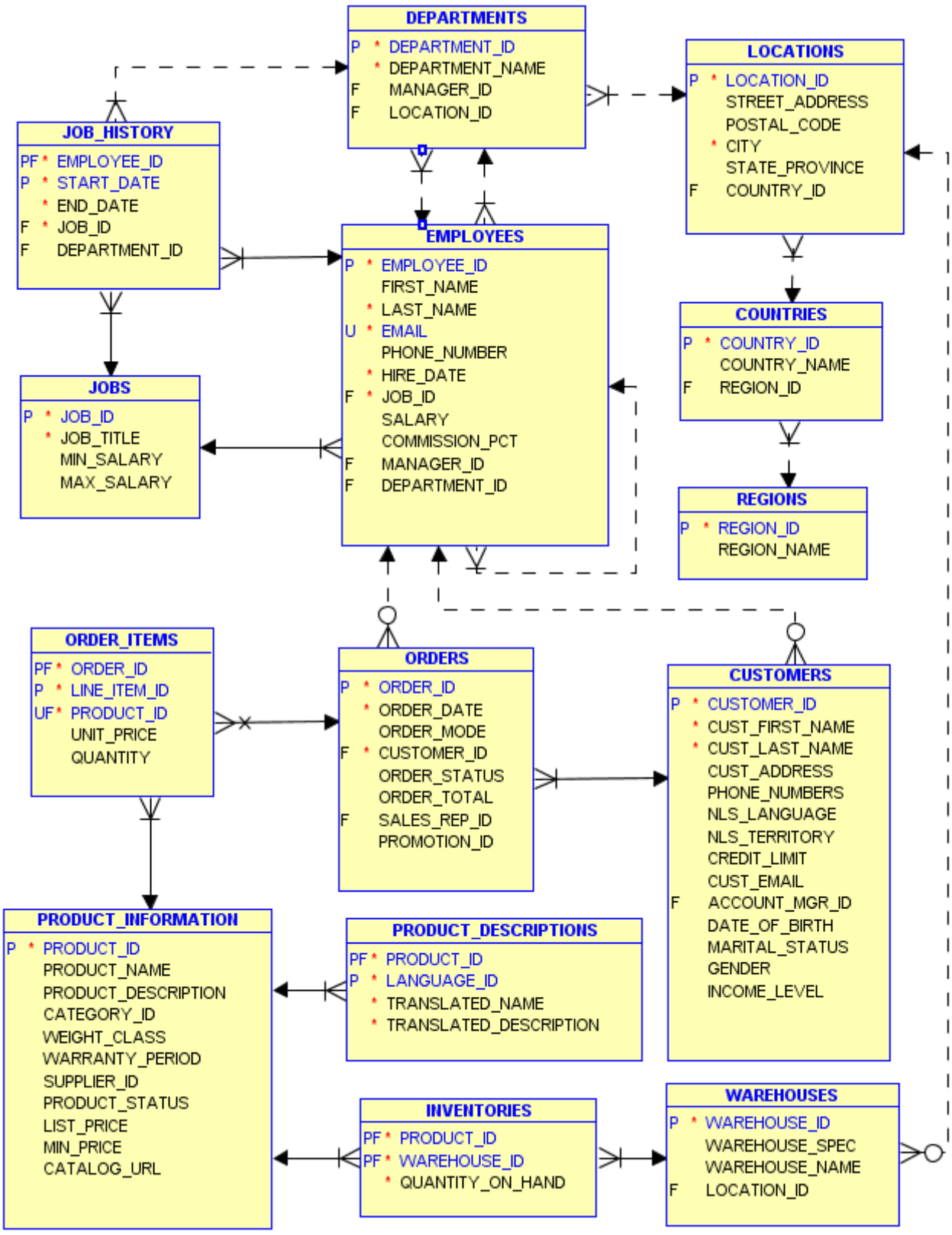
Также сохраняется информация о покупателях. Каждому покупателю присваивается идентификационный номер. Записывается имя покупателя, название улицы, город или район, страна, номера телефонов (до пяти номеров для каждого покупателя) и почтовый индекс. Некоторые покупатели вводят заказы через интернет, поэтому записываются также адреса их электронной почты. Записывается также родной язык покупателя и его страна, поскольку покупатели используют разные языки.

Для ограничения объема товаров, доступных для одноразовой покупки, компания устанавливает своим покупателям кредитный лимит. Некоторые покупатели имеют персонального менеджера, эта информация также записывается.

Когда покупатель вводит заказ, записывается дата заказа, способ размещения заказа, текущий статус заказа, способ доставки, сумма заказа и данные

торгового представителя, содействовавшего размещению заказа. Торговым представителем может быть, а может и не быть персональный менеджер покупателя. Если заказ размещен через интернет, то данные о торговом представителе отсутствуют. Также записывается информация о заказанных товарах (товарных позициях), заказанное количество единиц товара, цене единицы товара.

## Диаграмма Сущность-Связь


DEPARTMENTS
P * DEPARTMENT_ID
* DEPARTMENT_NAME
F MANAGER_ID
F LOCATION_ID
LOCATIONS
P * LOCATION_ID
STREET_ADDRESS
POSTAL_CODE
* CITY
STATE_PROVINCE
F COUNTRY_ID
JOB_HISTORY
PF * EMPLOYEE_ID
P * START_DATE
* END_DATE
F * JOB_ID
F DEPARTMENT_ID
EMPLOYEES
P * EMPLOYEE_ID
FIRST_NAME
* LAST_NAME
U * EMAIL
PHONE_NUMBER
* HIRE_DATE
F * JOB_ID
SALARY
COMMISSION_PCT
F MANAGER_ID
F DEPARTMENT_ID
COUNTRIES
P * COUNTRY_ID
COUNTRY_NAME
F REGION_ID
JOBS
P * JOB_ID
* JOB_TITLE
MIN_SALARY
MAX_SALARY
REGIONS
P * REGION_ID
REGION_NAME
ORDER_ITEMS
PF * ORDER_ID
P * LINE_ITEM_ID
UF * PRODUCT_ID
UNIT_PRICE
QUANTITY
ORDERS
P * ORDER_ID
* ORDER_DATE
ORDER_MODE
F * CUSTOMER_ID
ORDER_STATUS
ORDER_TOTAL
F SALES_REP_ID
PROMOTION_ID
CUSTOMERS
P * CUSTOMER_ID
* CUST_FIRST_NAME
* CUST_LAST_NAME
CUST_ADDRESS
PHONE_NUMBERS
NLS_LANGUAGE
NLS_TERRITORY
CREDIT_LIMIT
CUST_EMAIL
F ACCOUNT_MGR_ID
DATE_OF_BIRTH
MARITAL_STATUS
GENDER
INCOME_LEVEL
PRODUCT_INFORMATION
P * PRODUCT_ID
PRODUCT_NAME
PRODUCT_DESCRIPTION
CATEGORY_ID
WEIGHT_CLASS
WARRANTY_PERIOD
SUPPLIER_ID
PRODUCT_STATUS
LIST_PRICE
MIN_PRICE
CATALOG_URL
PRODUCT_DESCRIPTIONS
PF * PRODUCT_ID
P * LANGUAGE_ID
* TRANSLATED_NAME
* TRANSLATED_DESCRIPTION
INVENTORIES
PF * PRODUCT_ID
PF * WAREHOUSE_ID
* QUANTITY_ON_HAND
WAREHOUSES
P * WAREHOUSE_ID
WAREHOUSE_SPEC
WAREHOUSE_NAME
F LOCATION_ID

# Описание таблиц и скрипты для их создания

## REGIONS

Таблица REGIONS содержит сведения о регионах деятельности компании.

```
Name                      Null?    Type
------------------------- -------- ------------
REGION_ID                 NOT NULL NUMBER
REGION_NAME                        VARCHAR2(25)
```

REGION_ID – идентификатор региона (первичный ключ).

REGION_NAME – название региона.

```
CREATE TABLE regions(
   region_id      NUMBER
      CONSTRAINT region_id_nn NOT NULL
 , region_name    VARCHAR2(25) );

CREATE UNIQUE INDEX reg_id_pk ON regions(region_id);

ALTER TABLE regions
   ADD CONSTRAINT reg_id_pk
       PRIMARY KEY (region_id);
```

## COUNTRIES

Таблица COUNTRIES содержит сведения о странах, где работает компания.

```
Name                      Null?    Type
------------------------- -------- ------------
COUNTRY_ID                NOT NULL CHAR(2)
COUNTRY_NAME                       VARCHAR2(40)
REGION_ID                          NUMBER
```

COUNTRY_ID – идентификатор страны (первичный ключ).

REGION_NAME – название страны.

REGION_ID – идентификатор региона деятельности компании, к которому отнесена данная страна. Внешний ключ, ссылается на REGIONS.region_id.

```
CREATE TABLE countries(
   country_id      CHAR(2)
      CONSTRAINT country_id_nn NOT NULL
 , country_name    VARCHAR2(40)
 , region_id       NUMBER
 , CONSTRAINT    country_c_id_pk
      PRIMARY KEY (country_id) )
    ORGANIZATION INDEX;

ALTER TABLE countries
   ADD CONSTRAINT countr_reg_fk
          FOREIGN KEY (region_id)
          REFERENCES regions(region_id);
```

## LOCATIONS

Таблица LOCATIONS содержит сведения о местах расположения подразделений компании.

```
Name                          Null?    Type
----------------------------- -------- ------------
LOCATION_ID                   NOT NULL NUMBER(4)
STREET_ADDRESS                         VARCHAR2(40)
POSTAL_CODE                            VARCHAR2(12)
CITY                          NOT NULL VARCHAR2(30)
STATE_PROVINCE                         VARCHAR2(25)
COUNTRY_ID                             CHAR(2)
```

LOCATION_ID – идентификатор места расположения (местоположения) подразделения компании (первичный ключ).

STREET_ADDRESS – название улицы, номер дома и другие сведения об адресе местоположения.

POSTAL_CODE – почтовый индекс местоположения.

CITY – название города (населенного пункта), где находится местоположение.

STATE_PROVINCE – название штата (области, провинции), где расположен город.

COUNTRY_ID – идентификатор страны, где расположен город. Внешний ключ, ссылается на COUNTRYES.country_id.

```
CREATE TABLE locations(
   location_id    NUMBER(4)
 , street_address VARCHAR2(40)
 , postal_code    VARCHAR2(12)
 , city           VARCHAR2(30)
     CONSTRAINT loc_city_nn NOT NULL
 , state_province VARCHAR2(25)
 , country_id     CHAR(2));

CREATE UNIQUE INDEX loc_id_pk ON locations(location_id);

ALTER TABLE locations
   ADD (CONSTRAINT loc_id_pk
            PRIMARY KEY (location_id)
      , CONSTRAINT loc_c_id_fk
            FOREIGN KEY (country_id)
            REFERENCES countries(country_id) );
```

**DEPARTMENTS**

Таблица DEPARTMENTS содержит сведения о подразделениях компании.

```
Name                          Null?    Type
----------------------------- -------- ------------
DEPARTMENT_ID                 NOT NULL NUMBER(4)
DEPARTMENT_NAME               NOT NULL VARCHAR2(30)
MANAGER_ID                             NUMBER(6)
LOCATION_ID                            NUMBER(4)
```

DEPARTMENT_ID – идентификатор подразделения компании (первичный ключ).

DEPARTMENT_NAME – название подразделения компании.

MANAGER_ID – идентификатор руководителя подразделения. Внешний ключ, ссылается на EMPLOYEEES.employee_id.

LOCATION_ID – идентификатор места расположения подразделения компании. Внешний ключ, ссылается на LOCATIONS. location_id.

```
CREATE TABLE departments(
   department_id    NUMBER(4)
 , department_name  VARCHAR2(30)
      CONSTRAINT  dept_name_nn  NOT NULL
 , manager_id       NUMBER(6)
 , location_id      NUMBER(4) ) ;

CREATE UNIQUE INDEX dept_id_pk ON departments(department_id);

ALTER TABLE departments
   ADD (CONSTRAINT dept_id_pk
            PRIMARY KEY (department_id)
      , CONSTRAINT dept_loc_fk
            FOREIGN KEY (location_id)
            REFERENCES locations (location_id) );
```

**JOBS**

Таблица JOBS содержит сведения о должностях, которые могут занимать сотрудники компании.

```
Name                          Null?    Type
----------------------------- -------- ------------
JOB_ID                        NOT NULL VARCHAR2(10)
JOB_TITLE                     NOT NULL VARCHAR2(35)
MIN_SALARY                             NUMBER(6)
MAX_SALARY                             NUMBER(6)
```

JOB_ID – идентификатор должности (первичный ключ).

JOB_TITLE – название должности.

MIN_SALARY – минимальный оклад для данной должности.

MAX_SALARY – максимальный оклад для данной должности.

```
CREATE TABLE jobs(
   job_id         VARCHAR2(10)
 , job_title      VARCHAR2(35)
      CONSTRAINT job_title_nn  NOT NULL
 , min_salary     NUMBER(6)
 , max_salary     NUMBER(6) );

CREATE UNIQUE INDEX job_id_pk ON jobs (job_id) ;

ALTER TABLE jobs
   ADD CONSTRAINT job_id_pk
          PRIMARY KEY(job_id);
```

**EMPLOYEES**

Таблица EMPLOYEES содержит сведения о сотрудниках компании.

```
Name                      Null?    Type
------------------------- -------- ----------------------------
EMPLOYEE_ID               NOT NULL NUMBER(6)
FIRST_NAME                         VARCHAR2(20)
LAST_NAME                 NOT NULL VARCHAR2(25)
EMAIL                     NOT NULL VARCHAR2(25)
PHONE_NUMBER                       VARCHAR2(20)
HIRE_DATE                 NOT NULL DATE
JOB_ID                    NOT NULL VARCHAR2(10)
SALARY                             NUMBER(8,2)
COMMISSION_PCT                     NUMBER(2,2)
MANAGER_ID                         NUMBER(6)
DEPARTMENT_ID                      NUMBER(4)
```

EMPLOYEE_ID – идентификатор сотрудника (первичный ключ).

FIRST_NAME – имя сотрудника.

LAST_NAME – фамилия сотрудника.

EMAIL - адрес электронной почты сотрудника.

PHONE_NUMBER – номер служебного телефона сотрудника.

HIRE_DATE – дата, когда сотрудник был принят на работу.

JOB_ID – идентификатор должности, которую занимает сотрудник. Внешний ключ, ссылается на JOBS.job_id.

SALARY – оклад, установленный сотруднику.

COMMISSION_PCT – установленный размер комиссионных (в процентах от оклада).

MANAGER_ID – идентификатор непосредственного руководителя сотрудника. Внешний ключ, ссылается на EMPLOYEES.employee_id.

DEPARTMENT_ID – идентификатор подразделения, к которому приписан сотрудник. Внешний ключ, ссылается на DEPARTMENTS.department_id.

```
CREATE TABLE employees(
   employee_id    NUMBER(6)
 , first_name     VARCHAR2(20)
 , last_name      VARCHAR2(25)
      CONSTRAINT emp_last_name_nn  NOT NULL
 , email          VARCHAR2(25)
      CONSTRAINT emp_email_nn  NOT NULL
 , phone_number   VARCHAR2(20)
 , hire_date      DATE
      CONSTRAINT emp_hire_date_nn  NOT NULL
 , job_id         VARCHAR2(10)
      CONSTRAINT     emp_job_nn  NOT NULL
 , salary         NUMBER(8,2)
 , commission_pct NUMBER(2,2)
 , manager_id     NUMBER(6)
 , department_id  NUMBER(4)
 , CONSTRAINT emp_salary_min
      CHECK (salary > 0)
 , CONSTRAINT emp_email_uk
      UNIQUE (email) );

CREATE UNIQUE INDEX emp_emp_id_pk ON employees(employee_id);

ALTER TABLE employees
   ADD (CONSTRAINT emp_emp_id_pk
           PRIMARY KEY (employee_id)
      , CONSTRAINT emp_dept_fk
           FOREIGN KEY (department_id)
           REFERENCES departments
      , CONSTRAINT emp_job_fk
           FOREIGN KEY (job_id)
           REFERENCES jobs (job_id)
      , CONSTRAINT emp_manager_fk
           FOREIGN KEY (manager_id)
           REFERENCES employees);

ALTER TABLE departments
   ADD CONSTRAINT dept_mgr_fk
       FOREIGN KEY (manager_id)
       REFERENCES employees (employee_id);
```

## JOB_HISTORY

Таблица JOB_HISTORY содержит сведения об истории занятия должностей сотрудниками компании.

```
Name                          Null?    Type
----------------------------- -------- ------------
EMPLOYEE_ID                   NOT NULL NUMBER(6)
START_DATE                    NOT NULL DATE
END_DATE                      NOT NULL DATE
JOB_ID                        NOT NULL VARCHAR2(10)
DEPARTMENT_ID                 NUMBER(4)
```

EMPLOYEE_ID – идентификатор сотрудника. Часть составного первичного ключа. Внешний ключ, ссылается на EMPLOYEES.employee_id.

START_DATE - дата начала работы сотрудника в должности. Часть составного первичного ключа.

END_DATE - дата окончания работы сотрудника в должности.

JOB_ID – идентификатор должности. Внешний ключ, ссылается на JOBS.job_id.

DEPARTMENT_ID - идентификатор подразделения, к которому был приписан сотрудник. Внешний ключ, ссылается на DEPARTMENTS.department_id.

Таблица JOB_HISTORY имеет составной первичный ключ, состоящий из атрибутов EMPLOYEE_ID и START_DATE.

```
CREATE TABLE job_history(
   employee_id   NUMBER(6) CONSTRAINT jhist_employee_nn  NOT NULL
 , start_date    DATE      CONSTRAINT jhist_start_date_nn  NOT NULL
 , end_date      DATE      CONSTRAINT jhist_end_date_nn  NOT NULL
 , job_id        VARCHAR2(10) CONSTRAINT jhist_job_nn  NOT NULL
 , department_id NUMBER(4)
 , CONSTRAINT jhist_date_interval
      CHECK (end_date > start_date));

ALTER TABLE job_history
   ADD (CONSTRAINT jhist_emp_id_st_date_pk
            PRIMARY KEY (employee_id, start_date)
      , CONSTRAINT jhist_job_fk
            FOREIGN KEY (job_id)
            REFERENCES jobs
      , CONSTRAINT jhist_emp_fk
            FOREIGN KEY (employee_id)
            REFERENCES employees
      , CONSTRAINT jhist_dept_fk
            FOREIGN KEY (department_id)
            REFERENCES departments);

CREATE UNIQUE INDEX jhist_emp_id_st_date_pk
   ON job_history (employee_id, start_date);
```

**CUSTOMERS**

Таблица CUSTOMERS содержит сведения о покупателях, с которыми работает компания.

```
Name              Null?    Type
----------------- -------- ---------------
CUSTOMER_ID       NOT NULL NUMBER(6)
CUST_FIRST_NAME   NOT NULL VARCHAR2(20)
CUST_LAST_NAME    NOT NULL VARCHAR2(20)
CUST_ADDRESS               CUST_ADDRESS_TYP
PHONE_NUMBERS              PHONE_LIST_TYP
NLS_LANGUAGE               VARCHAR2(3)
NLS_TERRITORY              VARCHAR2(30)
CREDIT_LIMIT               NUMBER(9,2)
CUST_EMAIL                 VARCHAR2(30)
ACCOUNT_MGR_ID             NUMBER(6)
DATE_OF_BIRTH              DATE
MARITAL_STATUS             VARCHAR2(20)
GENDER                     VARCHAR2(1)
INCOME_LEVEL               VARCHAR2(20)
```

CUSTOMER_ID – идентификатор покупателя (первичный ключ).

CUST_FIRST_NAME – имя покупателя.

CUST_LAST_NAME – фамилия покупателя.

CUST_ADDRESS – адрес покупателя (пользовательский тип данных CUST_ADDRESS_TYP):

```
Name              Null?    Type
----------------- -------- -----------------
STREET_ADDRESS             VARCHAR2(40 CHAR)
POSTAL_CODE                VARCHAR2(10 CHAR)
CITY                       VARCHAR2(30 CHAR)
STATE_PROVINCE             VARCHAR2(10 CHAR)
COUNTRY_ID                 CHAR(2)
```

PHONE_NUMBERS - номера телефонов покупателя (пользовательский тип данных PHONE_LIST_TYP):

```
PHONE_LIST_TYP VARRAY(5) OF VARCHAR2(25)
```

NLS_LANGUAGE – код родного языка покупателя.

NLS_TERRITORY – название территории, где живет покупатель.

CREDIT_LIMIT – размер кредитного лимита покупателя.

CUST_EMAIL – адрес электронной почты покупателя.

ACCOUNT_MGR_ID - идентификатор персонального менеджера покупателя. Внешний ключ, ссылается на EMPLOYEES.employee_id.

DATE_OF_BIRTH – дата рождения покупателя.

MARITAL_STATUS – семейное положение покупателя:

- 'married' – женат/замужем,
- 'single' – не женат/не замужем.

GENDER – пол покупателя:

- 'F' - женский,

- 'M' – мужской.

INCOME_LEVEL – уровень доходов покупателя.

```
CREATE OR REPLACE TYPE  cust_address_typ AS OBJECT(
   street_address     VARCHAR2(40 CHAR)
 , postal_code        VARCHAR2(10 CHAR)
 , city               VARCHAR2(30 CHAR)
 , state_province     VARCHAR2(10 CHAR)
 , country_id         CHAR(2)
   )
/

CREATE OR REPLACE TYPE phone_list_typ
AS
VARRAY(5) OF VARCHAR2(25)
/

CREATE TABLE customers(
   customer_id      NUMBER(6)
 , cust_first_name  VARCHAR2(20 CHAR) CONSTRAINT cust_fname_nn NOT NULL
 , cust_last_name   VARCHAR2(20 CHAR) CONSTRAINT cust_lname_nn NOT NULL
 , cust_address     CUST_ADDRESS_TYP
 , phone_numbers    PHONE_LIST_TYP
 , nls_language     VARCHAR2(3 CHAR)
 , nls_territory    VARCHAR2(30 CHAR)
 , credit_limit     NUMBER(9,2)
 , cust_email       VARCHAR2(30 CHAR)
 , account_mgr_id   NUMBER(6)
 , date_of_birth    DATE
 , marital_status   VARCHAR2(20 CHAR)
 , gender           VARCHAR2(1 CHAR)
 , income_level     VARCHAR2(20 CHAR)
 , CONSTRAINT customers_pk
      PRIMARY KEY(customer_id)
 , CONSTRAINT customers_account_manager_fk
      FOREIGN KEY (account_mgr_id)
      REFERENCES employees (employee_id)
      ON DELETE SET NULL
 , CONSTRAINT customer_credit_limit_max
      CHECK (credit_limit <= 5000)
 , CONSTRAINT customer_id_min
      CHECK (customer_id > 0) );

CREATE INDEX cust_account_manager_ix ON customers(account_mgr_id);

CREATE INDEX cust_email_ix ON customers(cust_email);

CREATE INDEX cust_lname_ix ON customers(cust_last_name);

CREATE INDEX cust_upper_name_ix
   ON customers(UPPER(cust_last_name), UPPER(cust_first_name));
```

## WAREHOUSES

Таблица WAREHOUSES содержит сведения о товарных складах компании.

```
Name            Null?    Type
--------------- -------- -----------------

WAREHOUSE_ID    NOT NULL NUMBER(3)
WAREHOUSE_SPEC           SYS.XMLTYPE
WAREHOUSE_NAME           VARCHAR2(35 CHAR)
LOCATION_ID              NUMBER(4)
```

WAREHOUSE_ID – идентификатор товарного склада (первичный ключ).

WAREHOUSE_SPEC – описание товарного склада.

WAREHOUSE_NAME – название товарного склада.

LOCATION_ID – идентификатор места расположения товарного склада. Внешний ключ, ссылается на LOCATIONS.location_id.

```
CREATE TABLE warehouses(
   warehouse_id    NUMBER(3)
 , warehouse_spec  SYS.XMLTYPE
 , warehouse_name  VARCHAR2(35 CHAR)
 , location_id     NUMBER(4)
 , CONSTRAINT warehouses_pk
      PRIMARY KEY(warehouse_id)
 , CONSTRAINT warehouses_location_fk
      FOREIGN KEY(location_id)
      REFERENCES locations (location_id)
      ON DELETE SET NULL )
  XMLTYPE COLUMN warehouse_spec STORE AS SECUREFILE BINARY XML;
CREATE INDEX whs_location_ix ON warehouses(location_id);
```

## PRODUCT_INFORMATION

Таблица PRODUCT_INFORMATION содержит сведения о товарах, продаваемых компанией.

```
Name                 Null?    Type
-------------------- -------- -------------------------
PRODUCT_ID           NOT NULL NUMBER(6)
PRODUCT_NAME                  VARCHAR2(50 CHAR)
PRODUCT_DESCRIPTION           VARCHAR2(2000 CHAR)
CATEGORY_ID                   NUMBER(2)
WEIGHT_CLASS                  NUMBER(1)
WARRANTY_PERIOD               INTERVAL YEAR(2) TO MONTH
SUPPLIER_ID                   NUMBER(6)
PRODUCT_STATUS                VARCHAR2(20 CHAR)
LIST_PRICE                    NUMBER(8,2)
MIN_PRICE                     NUMBER(8,2)
CATALOG_URL                   VARCHAR2(50 CHAR)
```

PRODUCT_ID – идентификатор товара (первичный ключ).

PRODUCT_NAME – название товара.

PRODUCT_DESCRIPTION – описание товара.

CATEGORY_ID – идентификатор категории, к которой относится товар.

WEIGHT_CLASS – весовая группа (требуется для организации доставки товаров).

WARRANTY_PERIOD – гарантийный период (если имеется).

SUPPLIER_ID – идентификатор поставщика данного товара.

PRODUCT_STATUS – статус доступности товара.

LIST_PRICE – цена товара по каталогу.

MIN_PRICE – минимальная цена, по которой может быть продан товар.

CATALOG_URL – URL-ссылка на информацию производителя товара.

```
CREATE TABLE product_information(
   product_id           NUMBER(6)
 , product_name         VARCHAR2(50 CHAR)
 , product_description  VARCHAR2(2000 CHAR)
 , category_id          NUMBER(2)
 , weight_class         NUMBER(1)
 , warranty_period      INTERVAL YEAR(2) TO MONTH
 , supplier_id          NUMBER(6)
 , product_status       VARCHAR2(20 CHAR)
 , list_price           NUMBER(8,2)
 , min_price            NUMBER(8,2)
 , catalog_url          VARCHAR2(50 CHAR)
 , CONSTRAINT product_information_pk PRIMARY KEY(product_id)
 , CONSTRAINT product_status_lov
   CHECK (product_status IN ('orderable' ,'planned' ,'under development'
,'obsolete')) );

CREATE INDEX prod_supplier_ix ON product_information(supplier_id);
```

**PRODUCT_ DESCRIPTIONS**

Таблица PRODUCT_DESCRIPTIONS содержит названия и описания товаров на разных национальных языках.

```
Name                                      Null?    Type
----------------------- -------- ----------------
PRODUCT_ID              NOT NULL NUMBER(6)
LANGUAGE_ID             NOT NULL VARCHAR2(3 CHAR)
TRANSLATED_NAME         NOT NULL NVARCHAR2(50)
TRANSLATED_DESCRIPTION NOT NULL NVARCHAR2(2000)
```

PRODUCT_ID – идентификатор товара. Часть составного первичного ключа. Внешний ключ, ссылается на PRODUCT_INFORMATION.product_id.

LANGUAGE_ID – идентификатор национального языка. Часть составного первичного ключа.

TRANSLATED_NAME – название товара на национальном языке.

TRANSLATED_DESCRIPTION – описание товара на национальном языке.

Таблица PRODUCT_DESCRIPTIONS имеет составной первичный ключ, состоящий из атрибутов PRODUCT_ID и LANGUAGE_ID.

```
CREATE TABLE product_descriptions(
   product_id              NUMBER(6)
 , language_id             VARCHAR2(3 CHAR)
 , translated_name         NVARCHAR2(50) CONSTRAINT translated_name_nn NOT
NULL
 , translated_description  NVARCHAR2(2000) CONSTRAINT translated_desc_nn NOT
NULL
 , CONSTRAINT product_descriptions_pk PRIMARY KEY(product_id, language_id)
 , CONSTRAINT pd_product_id_fk
      FOREIGN KEY (product_id)
      REFERENCES product_information(product_id) );

CREATE INDEX prod_name_ix ON product_descriptions(translated_name);
```

**INVENTORIES**

Таблица INVENTORIES содержит учетные сведения по товарам.

```
Name              Null?    Type
----------------- -------- ---------
PRODUCT_ID        NOT NULL NUMBER(6)
WAREHOUSE_ID      NOT NULL NUMBER(3)
QUANTITY_ON_HAND  NOT NULL NUMBER(8)
```

PRODUCT_ID – идентификатор товара. Часть составного первичного ключа. Внешний ключ, ссылается на PRODUCT_INFORMATION.product_id.

WAREHOUSE_ID – идентификатор склада, где товар имеется в наличии. Часть составного первичного ключа. Внешний ключ, ссылается на WAREHOUSES.warehouse_id.

QUANTITY_ON_HAND – имеющееся в наличии количество товара.

Таблица PRODUCT_DESCRIPTIONS имеет составной первичный ключ, состоящий из атрибутов PRODUCT_ID и WAREHOUSE_ID.

```
CREATE TABLE inventories(
   product_id        NUMBER(6)
 , warehouse_id      NUMBER(3) CONSTRAINT inventory_warehouse_id_NN NOT NULL
 , quantity_on_hand  NUMBER(8) CONSTRAINT inventory_qoh_nn NOT NULL
 , CONSTRAINT inventory_pk PRIMARY KEY(product_id, warehouse_id)
 , CONSTRAINT inventories_product_id_fk
      FOREIGN KEY(product_id)
      REFERENCES product_information(product_id)
 , CONSTRAINT inventories_warehouses_fk
      FOREIGN KEY(warehouse_id)
      REFERENCES warehouses(warehouse_id) );
```

**ORDERS**

Таблица ORDERS содержит сведения о заказах, размещенных покупателями.

```
Name          Null?    Type
------------- -------- ---------------------------
ORDER_ID      NOT NULL NUMBER(12)
ORDER_DATE    NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE             VARCHAR2(8 CHAR)
CUSTOMER_ID   NOT NULL NUMBER(6)
ORDER_STATUS           NUMBER(2)
ORDER_TOTAL            NUMBER(8,2)
SALES_REP_ID           NUMBER(6)
PROMOTION_ID           NUMBER(6)
```

ORDER_ID – идентификатор заказа (первичный ключ).

ORDER_DATE- дата размещения заказа.

ORDER_MODE – способ размещения заказа.

CUSTOMER_ID – идентификатор покупателя, разместившего заказ. Внешний ключ, ссылается на CUSTOMERS.customer_id.

ORDER_STATUS – текущий статус заказа:

0: Not fully entered,

1: Entered,

2: Canceled - bad credit,

3: Canceled - by customer,

4: Shipped - whole order,

5: Shipped - replacement items,

6: Shipped - backlog on items,

7: Shipped - special delivery,

8: Shipped - billed,

9: Shipped - payment plan,

10: Shipped – paid.

ORDER_TOTAL – сумма заказа.

SALES_REP_ID – идентификатор торгового представителя, содействовавшего размещению заказа. Внешний ключ, ссылается на EMPLOYEES.employee_id.

PROMOTION_ID – идентификатор акции по продвижению товара.

```
CREATE TABLE orders(
   order_id      NUMBER(12)
 , order_date    TIMESTAMP(6) WITH LOCAL TIME ZONE CONSTRAINT order_date_nn
NOT NULL
 , order_mode    VARCHAR2(8 CHAR)
 , customer_id   NUMBER(6) CONSTRAINT order_customer_id_nn NOT NULL
 , order_status  NUMBER(2)
 , order_total   NUMBER(8,2)
 , sales_rep_id  NUMBER(6)
 , promotion_id  NUMBER(6)
 , CONSTRAINT order_pk PRIMARY KEY(order_id)
 , CONSTRAINT orders_customer_id_fk
      FOREIGN KEY(customer_id)
      REFERENCES customers(customer_id)
      ON DELETE SET NULL
 , CONSTRAINT orders_sales_rep_fk
      FOREIGN KEY(sales_rep_id)
      REFERENCES employees(employee_id)
      ON DELETE SET NULL
 , CONSTRAINT order_mode_lov CHECK (order_mode IN ('direct','online'))
 , CONSTRAINT order_total_min CHECK (order_total >= 0) );

CREATE INDEX ord_customer_ix ON orders(customer_id);

CREATE INDEX ord_order_date_ix ON orders(order_date);

CREATE INDEX ord_sales_rep_ix ON orders(sales_rep_id);
```

**ORDER_ITEMS**

Таблица ORDER_ITEMS информацию о заказанных товарах (товарных позициях) по заказам, размещенным покупателями.

```
 Name          Null?    Type
 ------------- -------- -----------
 ORDER_ID      NOT NULL NUMBER(12)
 LINE_ITEM_ID  NOT NULL NUMBER(3)
 PRODUCT_ID    NOT NULL NUMBER(6)
 UNIT_PRICE             NUMBER(8,2)
 QUANTITY               NUMBER(8)
```

ORDER_ID – идентификатор заказа. Часть составного первичного ключа. Внешний ключ, ссылается на ORDERS.order_id.

LINE_ITEM_ID – номер товарной позиции в заказе. Часть составного первичного ключа.

PRODUCT_ID – идентификатор товара. Внешний ключ, ссылается на PRODUCT_INFORMATION.product_id.

UNIT_PRICE – цена единицы товара.

QUANTITY – заказанное количество единиц товара.

Таблица ORDER_ITEMS имеет составной первичный ключ, состоящий из атрибутов ORDER_ID и LINE_ITEM_ID.

В одном заказе не может быть двух одинаковых товаров. Это правило поддерживается декларативным ограничением уникальности по двум атрибутам ORDER_ID и PRODUCT_ID.

```
CREATE TABLE order_items(
   order_id      NUMBER(12)
 , line_item_id  NUMBER(3)  NOT NULL
 , product_id    NUMBER(6)  NOT NULL
 , unit_price    NUMBER(8,2)
 , quantity      NUMBER(8)
 , CONSTRAINT order_items_pk PRIMARY KEY(order_id, line_item_id)
 , CONSTRAINT order_items_uk UNIQUE (order_id, product_id)
 , CONSTRAINT order_items_order_id_fk
      FOREIGN KEY(order_id) REFERENCES orders(order_id)
      ON DELETE CASCADE
 , CONSTRAINT order_items_product_id_fk
      FOREIGN KEY (product_id) REFERENCES product_information(product_id) );
CREATE INDEX item_order_ix ON order_items(order_id);
CREATE INDEX item_product_ix ON order_items(product_id);
```