

PL/SQL в Oracle

- [Шаг 1 - Установка Oracle \(4\)](#)
- [Шаг 2 - Как выглядит Oracle в системе \(20\)](#)
- [Шаг 3 - Модель клиент-сервер \(24\)](#)
- [Шаг 4 - Настройка с помощью Net8 Easy Config \(26\)](#)
- [Шаг 5 - Командная строка SQL Plus \(32\)](#)
- [Шаг 6 - Создание пользователя и настройка прав доступа \(36\)](#)
- [Шаг 7 - Язык SQL и примитивные типы данных \(39\)](#)
- [Шаг 8 - Пересоздаем пользователя miller \(41\)](#)
- [Шаг 9 - Оператор SELECT \(45\)](#)
- [Шаг 10 - Выборка данных с помощью SELECT \(49\)](#)
- [Шаг 11 - Выборка данных с помощью SELECT, часть 2 \(52\)](#)
- [Шаг 12 - Выборка данных с помощью SELECT, часть 3 \(56\)](#)
- [Шаг 13 - Условие WHERE оператора SELECT \(58\)](#)
- [Шаг 14 - Оператор BETWEEN и IN \(61\)](#)
- [Шаг 15 - Оператор LIKE \(65\)](#)
- [Шаг 16 - Еще раз и подробно про NULL \(67\)](#)
- [Шаг 17 - Составные операторы в условии WHERE \(69\)](#)
- [Шаг 18 - Сортировка записей запроса, предложение ORDER BY \(73\)](#)
- [Шаг 19 - Подводим итог по запросам! \(75\)](#)
- [Шаг 20 - Подходим ближе! Основные компоненты БД Oracle \(81\)](#)
- [Шаг 21 - Основные компоненты БД Oracle - продолжаем! \(85\)](#)
- [Шаг 22 - Основные компоненты БД Oracle - Блоки Oracle \(88\)](#)
- [Шаг 23 - Основные компоненты БД Oracle - Пользовательские объекты БД \(90\)](#)
- [Шаг 24 - Основные компоненты БД Oracle - Сегменты БД - Таблицы \(94\)](#)
- [Шаг 25 - Основные компоненты БД Oracle - Сегменты БД, продолжаем \(98\)](#)
- [Шаг 26 - Снова SELECT - многотабличные запросы \(101\)](#)
- [Шаг 27 - Снова SELECT - Таблиц становится больше! \(106\)](#)
- [Шаг 28 - Снова SELECT - Объединения таблиц \(109\)](#)
- [Шаг 29 - Снова SELECT - имена, псевдонимы... \(111\)](#)
- [Шаг 30 - Снова SELECT - самообъединения, правила и т.д. \(115\)](#)
- [Шаг 31 - Снова SELECT - Применение агрегатных функций \(121\)](#)
- [Шаг 32 - Снова SELECT - Агрегатные функции далее... \(126\)](#)
- [Шаг 33 - SELECT - Агрегаты и старый знакомый NULL! \(130\)](#)
- [Шаг 34 - SELECT - Агрегаты и группировка данных \(133\)](#)
- [Шаг 35 - SELECT - "Группировка" данных далее \(138\)](#)
- [Шаг 36 - SELECT - "Группировка" и отбор HAVING \(143\)](#)
- [Шаг 37 - Вопрос ответ и идем дальше ! \(147\)](#)
- [Шаг 38 - PL/SQL - вводный курс \(150\)](#)
- [Шаг 39 - PL/SQL - топаем дальше .. топ .. топ \(154\)](#)
- [Шаг 40 - PL/SQL - дальше и дальше в пути \(156\)](#)
- [Шаг 41 - PL/SQL - блоки, литералы, переменные \(159\)](#)
- [Шаг 42 - PL/SQL - переменные, инициализация, правила \(165\)](#)
- [Шаг 43 - PL/SQL - оператор IF - THEN - ELSE \(168\)](#)
- [Шаг 44 - PL/SQL - GOTO и его определения, типы данных... \(174\)](#)
- [Шаг 45 - PL/SQL - DDL, DML и еще кое что \(178\)](#)
- [Шаг 46 - PL/SQL - DML, оператор INSERT \(183\)](#)
- [Шаг 47 - PL/SQL - DML, оператор DELETE \(187\)](#)
- [Шаг 48 - PL/SQL - DML, оператор UPDATE \(191\)](#)
- [Шаг 49 - PL/SQL - виды и типы циклов \(196\)](#)
- [Шаг 50 - PL/SQL - переменные их типы и кое что еще! \(202\)](#)

[Шаг 51 - PL/SQL - КУРСОР, теория](#) (205)
[Шаг 52 - PL/SQL - КУРСОР, теория - продолжаем](#) (207)
[Шаг 53 - PL/SQL - КУРСОР - идем дальше](#) (209)
[Шаг 54 - PL/SQL - КУРСОР - извлекаем данные, оператор FETCH](#) (212)
[Шаг 55 - PL/SQL - КУРСОР и его атрибуты](#) (218)
[Шаг 56 - PL/SQL - составные типы](#) (224)
[Шаг 57 - PL/SQL - составные типы данных](#) (232)
[Шаг 58 - PL/SQL - PL/SQL таблицы и их атрибуты](#) (237)
[Шаг 59 - PL/SQL - PL/SQL атрибуты PL/SQL таблиц, примеры](#) (244)
[Шаг 60 - PL/SQL - КУРСОРЫ - Неявные Курсоры](#) (249)
[Шаг 61 - PL/SQL - КУРСОРЫ - циклы и выборка данных](#) (254)
[Шаг 62 - PL/SQL - КУРСОРЫ - цикл выборки FOR](#) (259)
[Шаг 63 - PL/SQL - КУРСОРЫ - курсоры с параметрами](#) (261)
[Шаг 64 - PL/SQL - КУРСОРЫ - курсоры с обновлением](#) (266)
[Шаг 65 - PL/SQL - встроенные функции, часть 1](#) (272)
[Шаг 66 - PL/SQL - встроенные функции, часть 2](#) (275)
[Шаг 67 - PL/SQL - встроенные функции, часть 3](#) (278)
[Шаг 68 - PL/SQL - встроенные функции, часть 4](#) (281)
[Шаг 69 - PL/SQL - встроенные функции, дата и время](#) (284)
[Шаг 70 - PL/SQL - встроенные функции, даты и снова даты](#) (287)
[Шаг 71 - PL/SQL - функции, преобразования TO_CHAR\(...\) даты](#) (292)
[Шаг 72 - PL/SQL - функции, преобразования TO_CHAR\(...\) числа](#) (297)
[Шаг 73 - PL/SQL - функции PL/SQL осталось немного...](#) (301)
[Шаг 74 - Архитектура БД Oracle](#) (307)
[Шаг 75 - Архитектура БД Oracle ЧАСТЬ II](#) (311)
[Шаг 76 - Архитектура БД Oracle ЧАСТЬ III](#) (314)
[Шаг 77 - Архитектура БД Oracle ЧАСТЬ IV](#) (316)
[Шаг 78 - Архитектура БД Oracle ЧАСТЬ V](#) (317)
[Шаг 79 - Архитектура БД Oracle ЧАСТЬ VI](#) (319)
[Шаг 80 - Архитектура БД Oracle ЧАСТЬ VII](#) (321)
[Шаг 81 - Архитектура. Серверы, пользователи и т. д.](#) (323)
[Шаг 82 - Простейший мониторинг экземпляра БД](#) (325)
[Шаг 83 - Представления V\\$Process и V\\$Bgprocess](#) (327)
[Шаг 84 - Представление V\\$session](#) (330)
[Шаг 85 - Другие представления для мониторинга](#) (335)
[Шаг 86 - PL/SQL - Язык программирования Oracle](#) (337)
[Шаг 87 - PL/SQL - Именованные блоки процедуры](#) (339)
[Шаг 88 - PL/SQL - Работаем с процедурами](#) (343)
[Шаг 89 - PL/SQL - Процедуры и их параметры](#) (346)
[Шаг 90 - PL/SQL - Процедуры и их параметры II](#) (350)
[Шаг 91 - PL/SQL - Процедуры и параметры III](#) (353)
[Шаг 92 - PL/SQL - Процедуры и параметры еще немного](#) (357)
[Шаг 93 - PL/SQL - Функции](#) (362)
[Шаг 94 - PL/SQL - Функции и процедуры - размещение](#) (368)
[Шаг 95 - PL/SQL - Хранимые процедуры, объявления, зависимости](#) (373)
[Шаг 96 - PL/SQL - Понятие пакета в языке PL/SQL](#) (377)
[Шаг 97 - PL/SQL - Учимся создавать пакеты и работать с ними](#) (379)
[Шаг 98 - PL/SQL - Пакеты - Изменение, удаление](#) (384)
[Шаг 99 - PL/SQL - Пакеты - Переопределение, функций и процедур](#) (389)
[Шаг 100 - PL/SQL - Уровни строгости - Прагма RESTRICT REFERENCES](#) (394)
[Шаг 101 - Oracle - Создание схемы БД - Назначение прав](#) (398)
[Шаг 102 - Oracle - Пользователь \(схема\) и ее модификация](#) (403)
[Шаг 103 - Oracle - Оператор GRANT](#) (407)

[Шаг 104 - Oracle - Таблица - Ключевой блок БД](#) (412)
[Шаг 105 - Oracle - Реляционная модель, Правила КОДА \(E.F. CODD\)](#) (415)
[Шаг 106 - Oracle - Модели взаимосвязей объектов и проектирование](#) (417)
[Шаг 107 - Oracle - Реляционная модель - НОРМАЛИЗАЦИЯ](#) (421)
[Шаг 108 - Oracle - ТАБЛИЦЫ БД определение отношений](#) (427)
[Шаг 109 - Oracle - ТАБЛИЦЫ БД основные операторы определения](#) (430)
[Шаг 110 - PL/SQL - Триггеры БД, теория](#) (433)
[Шаг 111 - PL/SQL - Триггеры таблиц БД, операторный триггер](#) (435)
[Шаг 112 - PL/SQL - Триггеры - строковые и операторные](#) (442)
[Шаг 113 - PL/SQL - Триггеры - псевдозаписи - ЧАСТЬ I](#) (451)
[Шаг 114 - PL/SQL - Триггеры - псевдозаписи - ЧАСТЬ II \(бизнес правила\)](#) (459)
[Шаг 115 - PL/SQL - Триггеры - ЧАСТЬ III - условие WHERE](#) (466)
[Шаг 116 - PL/SQL - Триггеры - ЧАСТЬ IV - предикаты](#) (472)
[Шаг 117 - PL/SQL - Триггеры - ЧАСТЬ V - mutating table](#) (481)
[Шаг 118 - PL/SQL - Триггеры - ЧАСТЬ VI - системные триггеры](#) (488)
[Шаг 119 - Табличные пространства - Системы БД OLTP](#) (493)
[Шаг 120 - Табличные пространства - Системы БД DSS и другие](#) (495)
[Шаг 121 - Табличные пространства - создание, изменение, удаление](#) (496)
[Шаг 122 - Табличные пространства на практике](#) (499)
[Шаг 123 - БД Oracle - Табличные пространства, ввод - вывод](#) (505)
[Шаг 124 - БД Oracle - СИСТЕМА Ввода-Вывода и работа с ней](#) (507)
[Шаг 125 - БД Oracle - Оценка табличных пространств - дефрагментация](#) (509)
[Шаг 126 - БД Oracle - Дефрагментация и борьба с ней!](#) (512)
[Шаг 127 - БД Oracle - Быстрее, выше, сильнее!](#) (516)
[Шаг 128 - БД Oracle - Работа с табличными пространствами в целом](#) (518)
[Шаг 129 - БД Oracle - Импорт, экспорт данных в теории](#) (524)
[Шаг 130 - БД Oracle - Экспорт - команды и управление выводом](#) (528)
[Шаг 131 - БД Oracle - Экспорт - на примерах](#) (531)
[Шаг 132 - БД Oracle - Экспорт данных - окончание](#) (534)
[Шаг 133 - БД Oracle - Импорт данных - параметры импорта](#) (536)
[Шаг 134 - БД Oracle - Импорт \(экспорт\) в примерах](#) (539)
[Шаг 135 - БД Oracle - Загрузка данных SQL*Loader](#) (543)
[Шаг 136 - БД Oracle - SQL*Loader структура загрузчика](#) (546)
[Шаг 137 - БД Oracle - SQL*Loader Способы загрузки данных в БД](#) (549)
[Шаг 138 - БД Oracle - SQL*Loader Способы загрузки - ЧАСТЬ II](#) (553)
[Шаг 139 - БД Oracle - SQL*Loader Способы загрузки - ЧАСТЬ III](#) (559)
[Шаг 140 - БД Oracle - SQL*Loader Загрузка данных - ЧАСТЬ IV](#) (564)
[Шаг 141 - БД Oracle - SQL*Loader Теория и полезные примеры - ЧАСТЬ V](#) (570)
[Шаг 142 - БД Oracle - SQL*Loader Загрузка LOB объектов! ЧАСТЬ VI](#) (573)

Шаг 1 - Инсталляция Oracle

Для начала небольшое лирическое отступление. Компания **Oracle** была создана в 1977 году, в данное время директором компании является Лари Элисон. Компания насчитывает более 36 000 сотрудников (не считая нас пользователей), в более чем 90 странах мира. Годовой оборот компании составляет более 5,2 миллиарда долларов в год! Вот так работают ребята! Для примера сравните с бюджетом России скажем за прошлый год! :) Истории пока хватит.

Переходим к аппаратным требованиям. В принципе **Oracle Personal Edition**, ставиться на Пентиум 330 и 128 мег озу! Хотя возможно поэкспериментировать и с меньшими объемами, но я настоятельно не советую, так как заработаете "мозговую травму", пытаюсь например понять почему все так медленно работает? Например, я на своей службе не имею машин ниже Celeron 1300 и 256 озу. Но это кому как повезет! Значит будем считать, что **NT4.0** либо **Advanced 2000** у вас установлен и в радостном ожидании встречи с **Oracle Server** уже все работает.

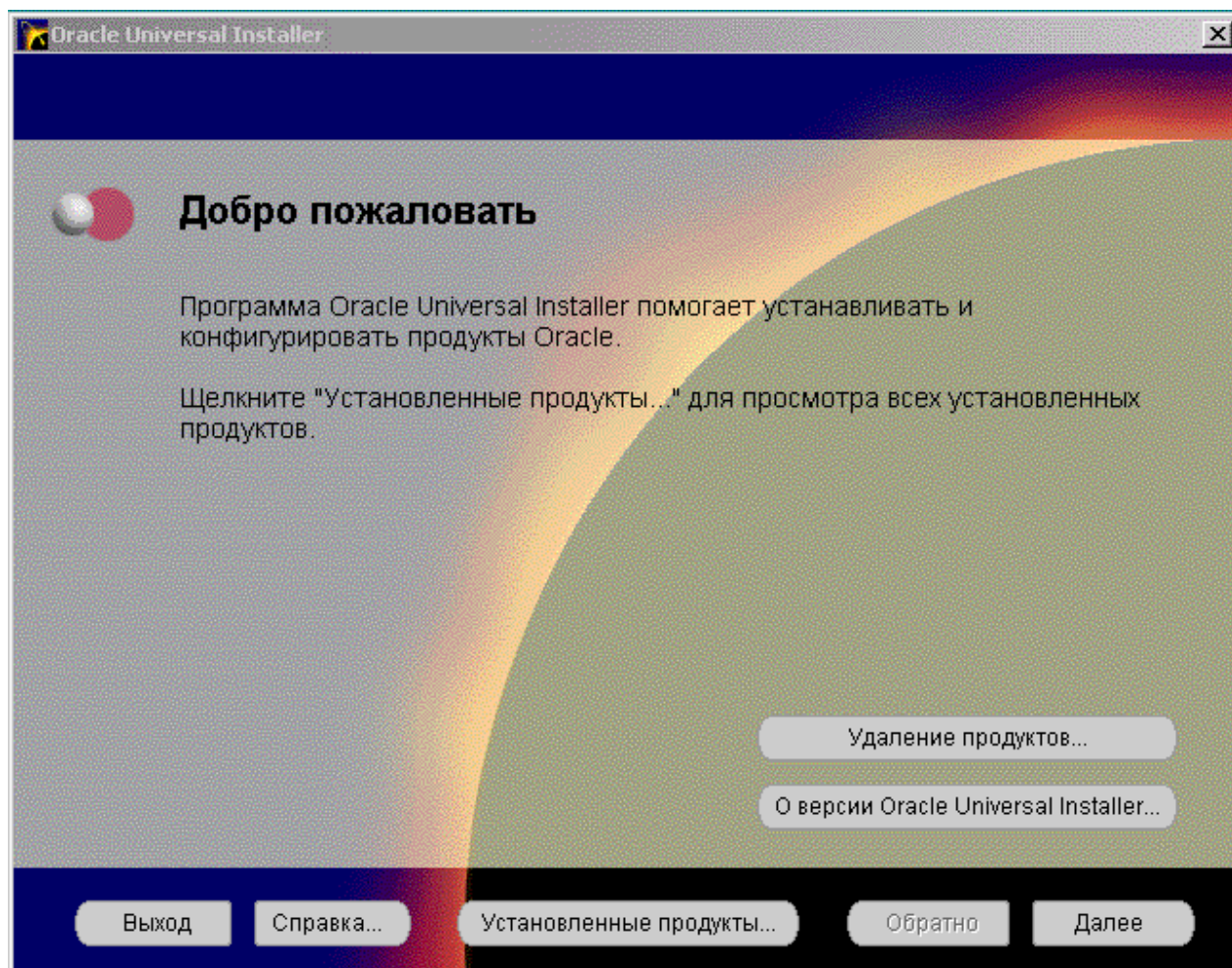
Сразу оговорюсь, все будем пробовать на платформу **NT**. Почему? Во-первых, засорять вам мозги операционкой типа **AIX, Solaris, Linux**, я не буду, тем более, что когда поумнеете и без меня сами разберетесь, а работать с **Oracle**, что на **UNIX**, что на **NT**, с точки зрения клиента особой разницы нет. Просто я хочу без лишних трений попытаться показать Вам всю мощь этой зверюги под именем **ORACLE**, а там флаг вам в руки, держайте изучайте операционные системы и юзайте его по полной!

Еще один щепетильный момент - МелкоМягкие очень не любят все, что производят в конторе Лари Элисона, по этому предупреждаю сразу не стоит экспериментировать и пытаться скрещивать на одном серваке **MS SQL** и **Oracle** !!! :(Результаты будут самые плачевные!!! Лучше отдельная машина и веселитесь в свое удовольствие!!! А по поводу какой сервер лучше **MS SQL** или **Oracle**, я полемику разводить уж тем более не буду!!! Да и вам не советую !!!

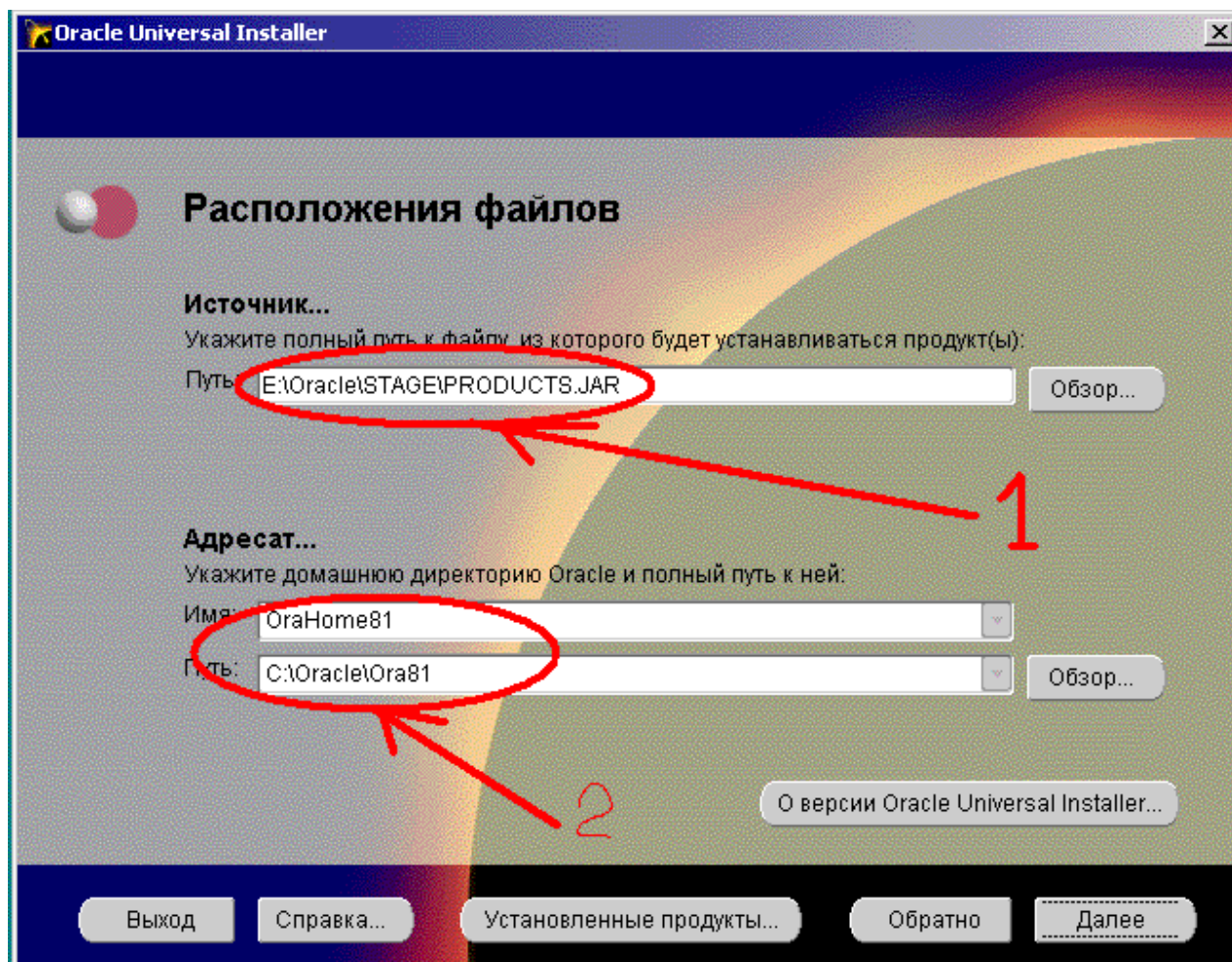
Переходим непосредственно к инсталляции, так как без это начинать работать с **Oracle**, не имеет особого смысла, а знать этот процесс, на будущее вам пригодиться! Я все поведу от **Oracle 8.1.5.0** и вот почему! У меня на руках версия этого сервера наиболее правдоподобная и проверенная, 8.1.7.0 у меня тоже есть, но она имеет мало отличий от первой. Есть у меня и 9i, но для нее у меня еще нет путевого сервака, а посему делайте вывод. Можете отступать от моих инструкций благо я не страдаю манией величия и вообще всем свойственно ошибаться. Все замечания и дополнения будут приниматься дабы увеличить, долю истины в сим мероприятии познания **Oracle Server**!

Итак, кто не имеет дистрибутива 8.1.5.0 и иже с ними желательно приобрести, либо поспрашивать у друзей и знакомых!!! Кто-нибудь обязательно поможет и я в это искренне верю, время у вас еще есть!!!

Итак, устанавливаем диск в компашик и чутко прислушиваемся к процессу **AutoRun**! Ух ты, и что мы видим!?

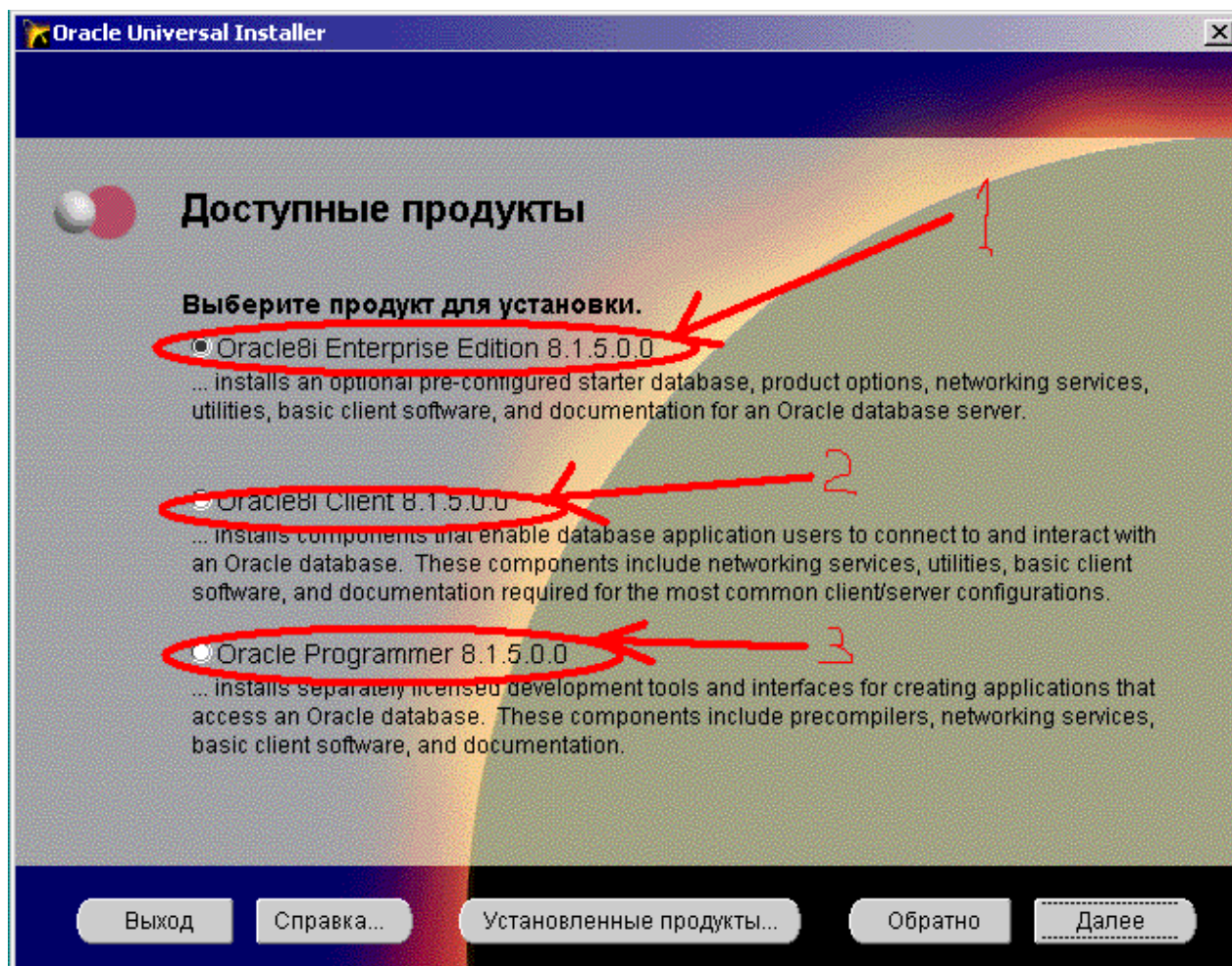


Oracle Universal Installer - сия прога поможет нам установить и сконфигурировать продукты **Oracle** на вашем сервере или рабочей станции. Предложение щелкнуть по кнопке "Установленные продукты" советую пропустить и не никуда пока не "щелкать", а просто кликнуть кнопочку "далее" насладившись приятным баннером со словами "Добро пожаловать"!!!



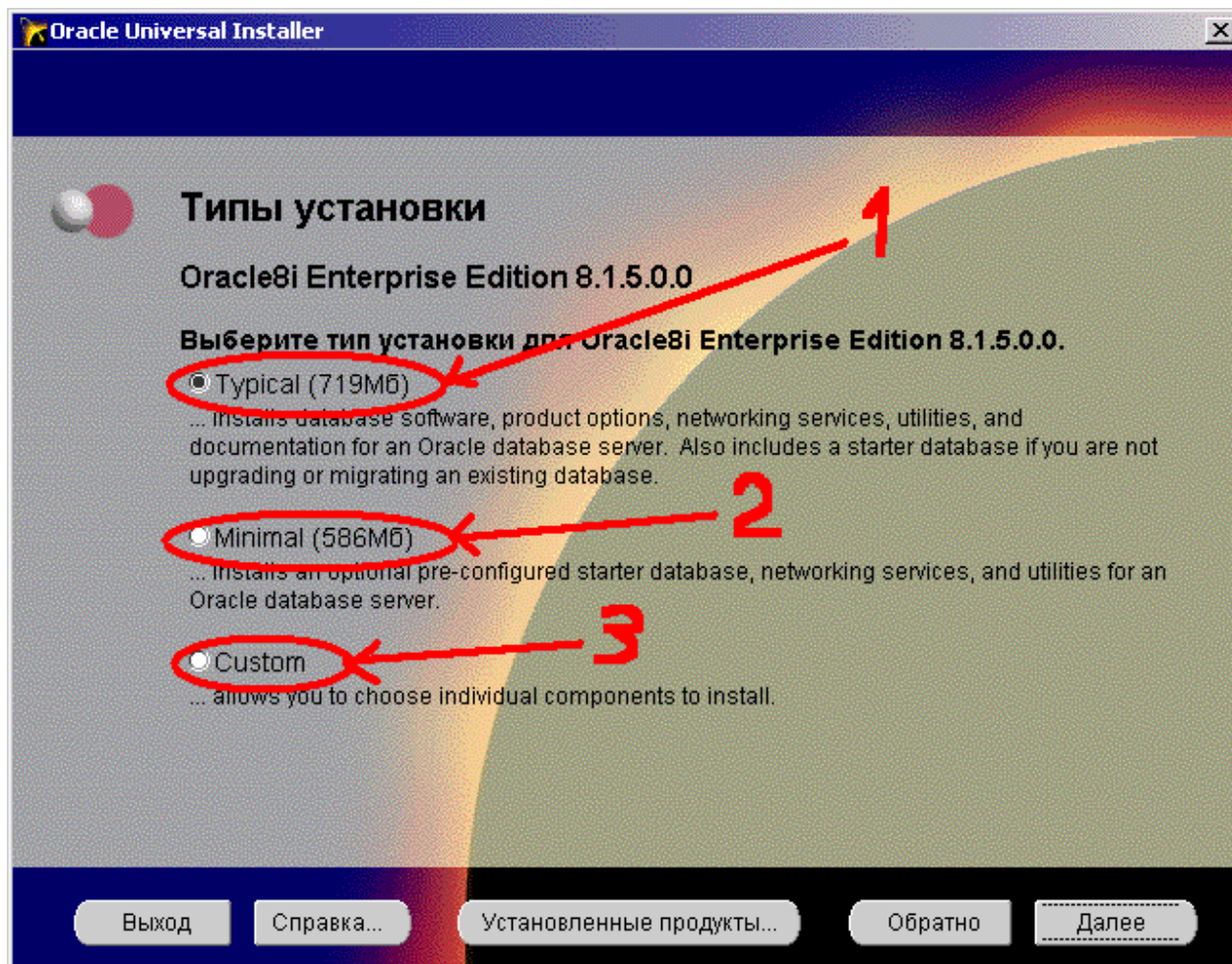
Далее попадаем на экран "Расположение файлов" - здесь думается нужно немного остановиться подробнее: позиция 1 - это то где расположены ваши файлы инсталляции, то есть компакт привод вашей машины, либо что-то еще. Позиция 2 - это системная переменная окружения **Oracle**, по которым ищутся компоненты сервера или клиента и соответственно путь к этим компонентам. Менять здесь, что либо до поры до времени не советую, но если очень чешутся ручки, можете попробовать, но за последствия не отвечаю!!! :))))

Особо усидчивые кликайте кнопку "далее" и смотрите следующий экран!

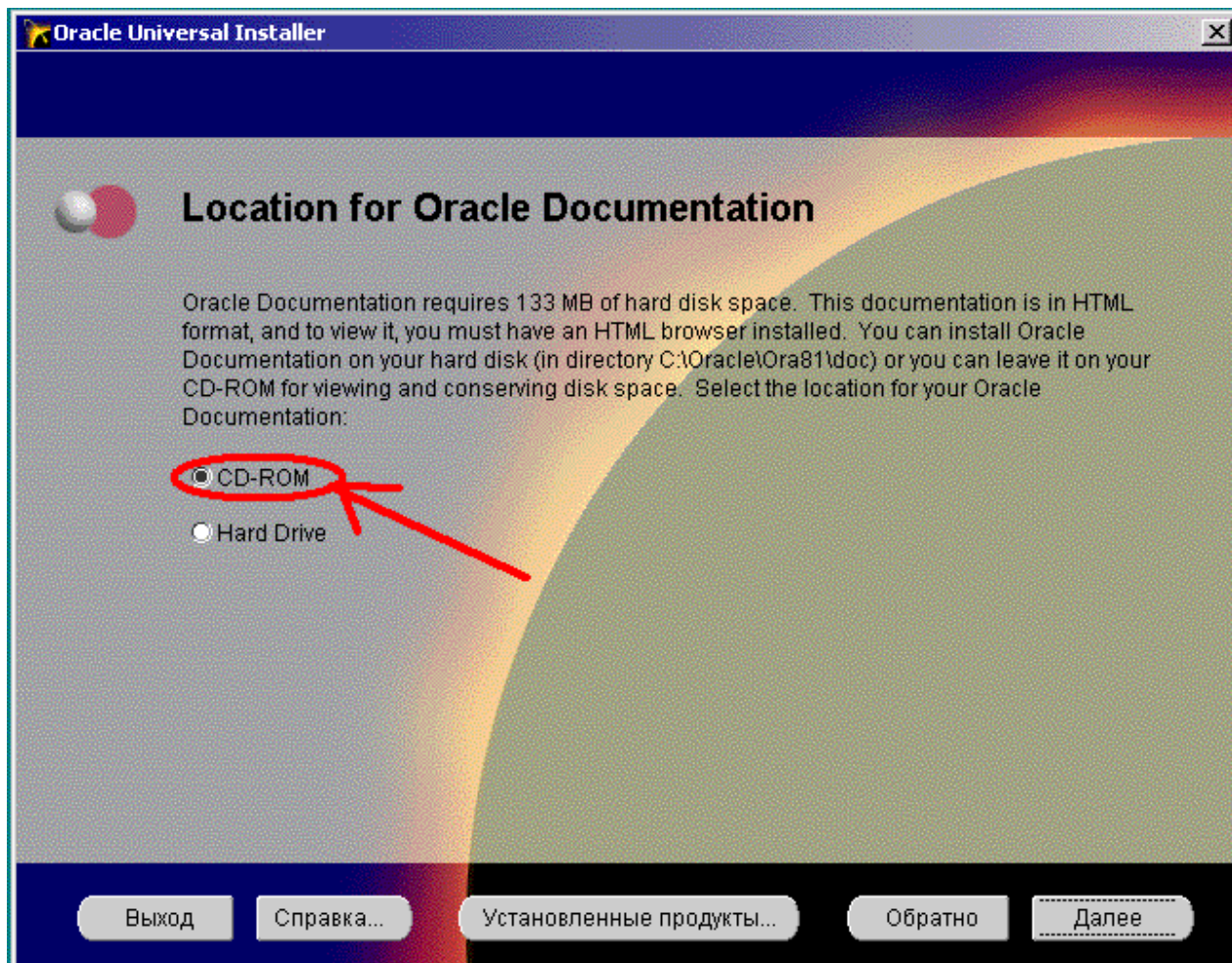


Итак, кажется началось! Вот оно собственно сам сервер собственной персоной (хорошо, что не нон-грато :). Теперь давайте немного притормозим и разберемся. Позиция 1 указывает на то, что мы выбираем для инсталляции собственно сервер, но это не совсем так, потому что эта же позиция добавит на сервер NT и клиентскую часть, чтобы можно было общаться с сервером **Oracle** непосредственно с сервера! Во, туфтология началась! Вторая позиция это установка чистого клиента, его еще называют "толстым"! Но об этом чуть позже. И на конец третья позиция, это как не трудно догадаться для программера, дабы он своял нам что-то удобоваримое для работы с сервером!

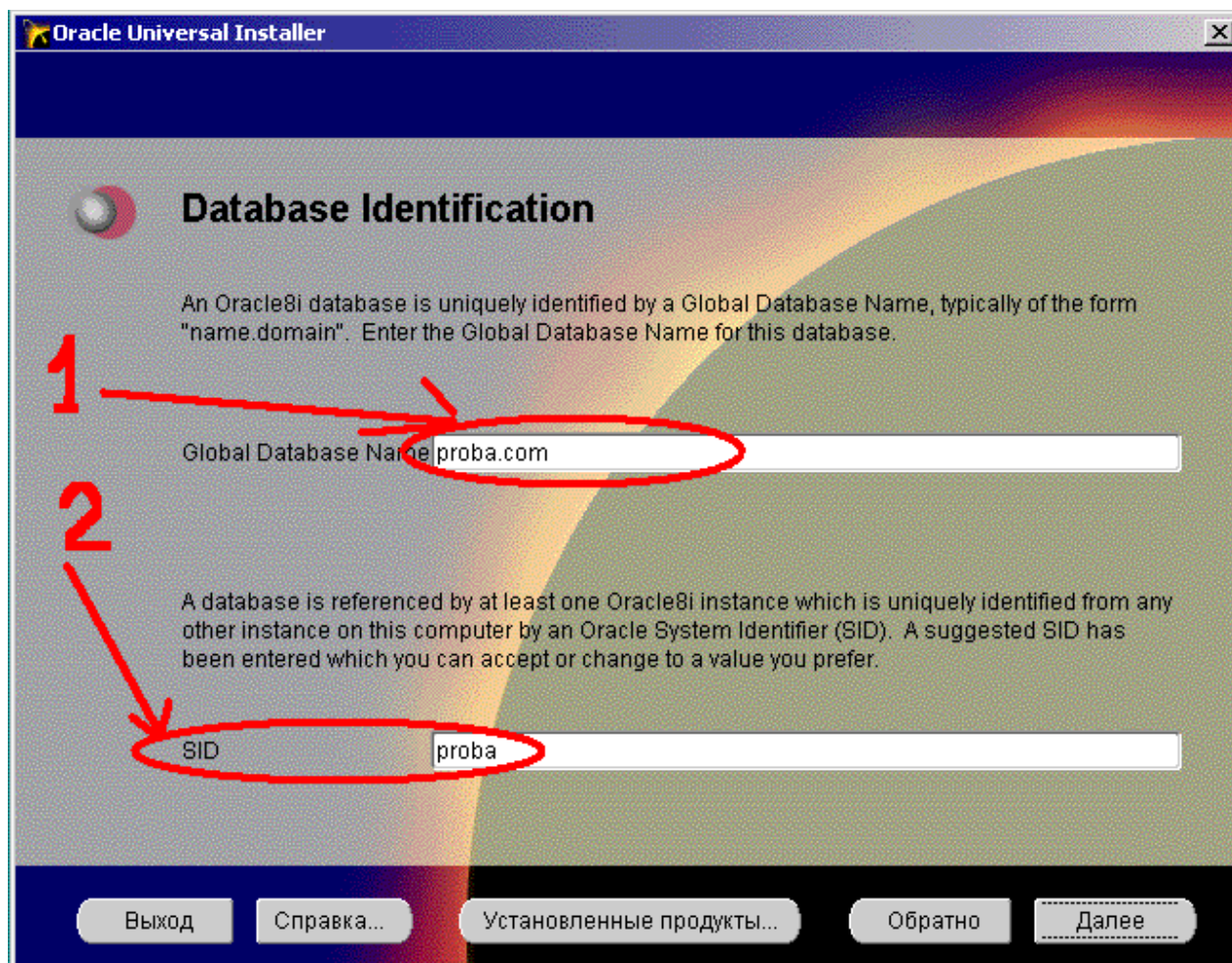
Усидчивые и те кому я еще не надоел кликайте кнопочку далее и смотрите следующий экран!



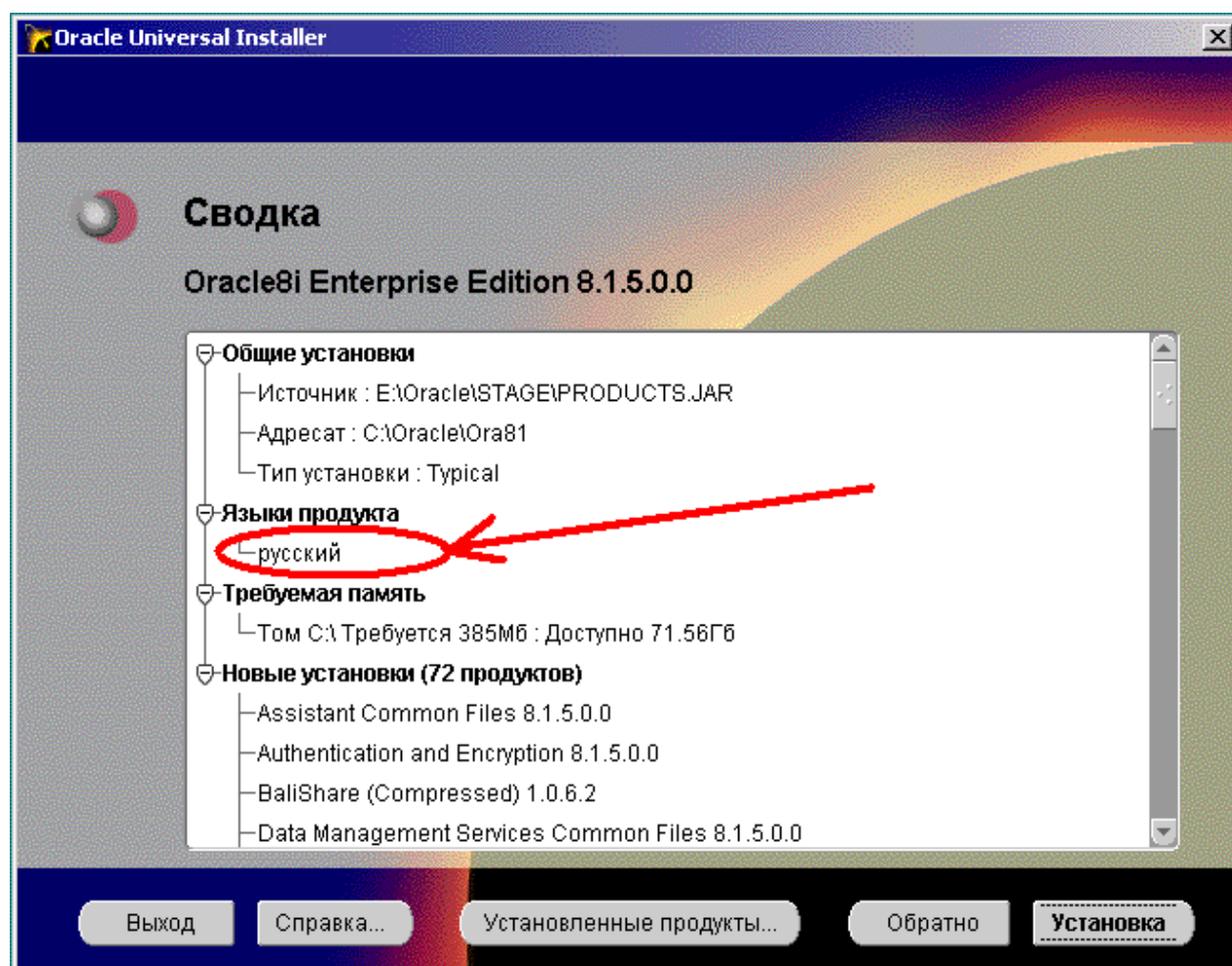
А!, вот тоже кое что интересное! Сейчас нам предлагают, выбрать тип установки, пока выбирайте "типовая", она самая простая и задает меньше всего вопросов, а самое главное делает почти все до конца сама и не надоедает всякими глупыми предложениями! Выбирайте ее и не мудрите. Хотя, если кто выберет минимальную, тоже ничего, но вопросы по ходу будете решать сами! А вот пользовательскую пока не советую трогать вообще, так как у вас еще знаний маловато. Жмем далее и смотрим следующий экран!



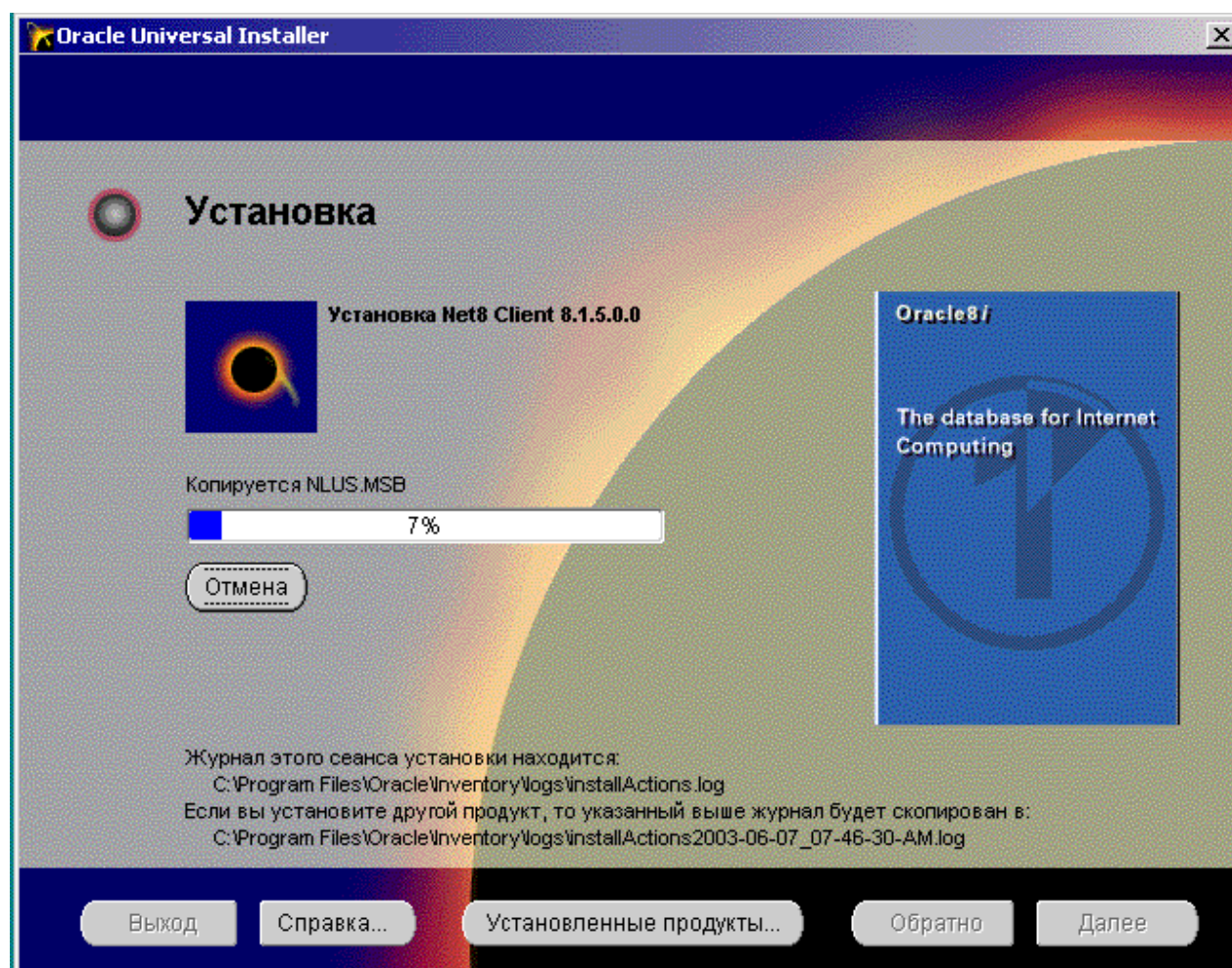
Здесь советую оставить как есть опция **CD-ROM**, иначе она вам зафигачит к тем семистам еще 133 метра, на ваш диск! Так что как всегда Далее!!!



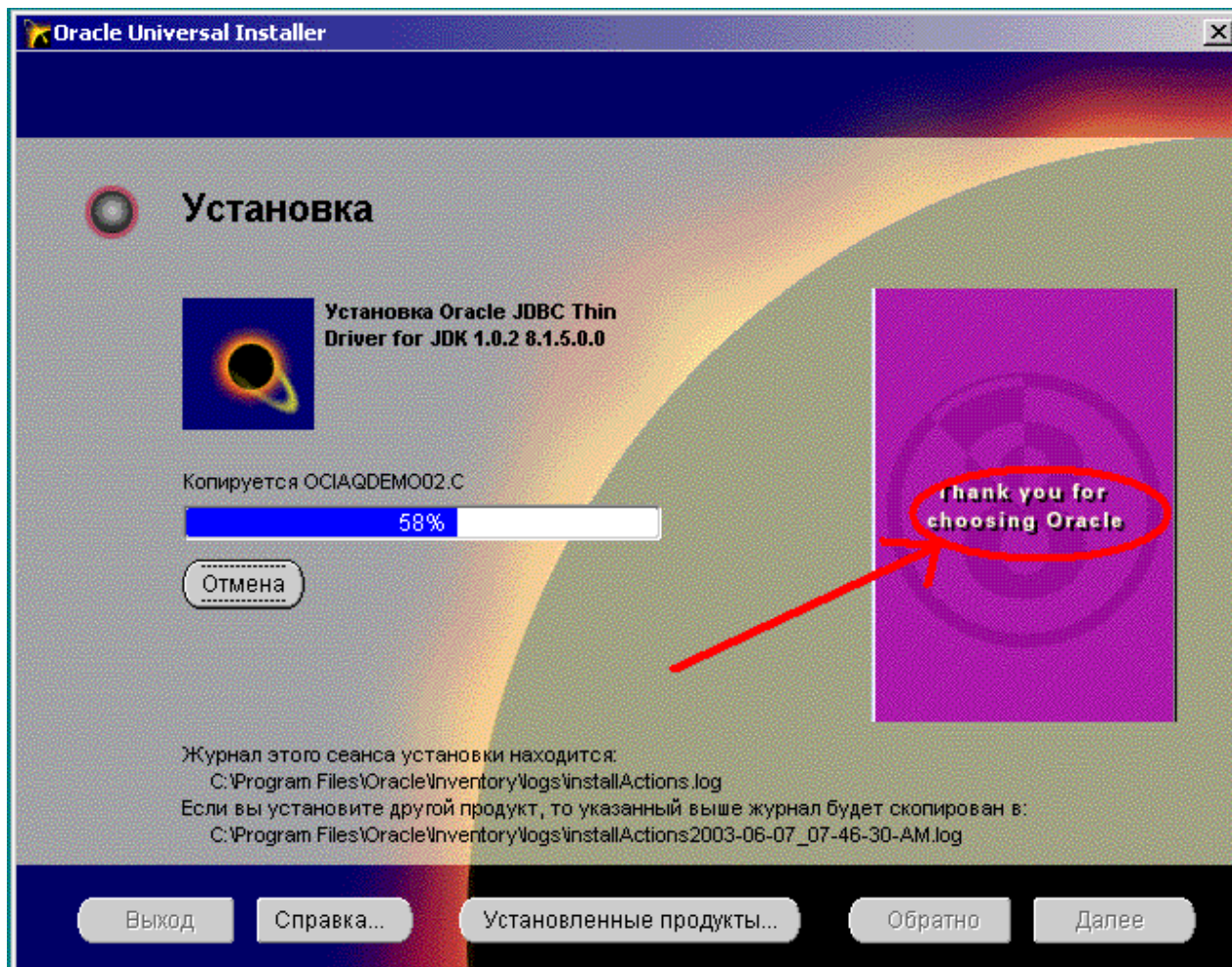
Так - ПРИЕХАЛИ!!! Не знаю даже с чего начать! Начнем с первой позиции **Global DataBase Name** - так как **Oracle** начиная с версии с буквой **-i**, означает что доменные имена БД имеют те же правила, что и в Интернете, следовательно получай фашист гранату! Необходимо писать Имя->Дот->Домен, то есть, например **vasiarupkin.ru**, либо что-то еще что придет в ваше окрыленное моментом сознание! Я написал просто **proba.com**, так как "он уважать себя заставил и лучше выдумать не мог"! Вторая позиция **SID** это все кроме **.com**, то есть имя экземпляра базы глобальное, с ним будет связан еще один момент, но о нем чуть позже! Так что, если здесь не соблюсти схему Имя->Дот->Домен, будете делать все заново!!! А ваш клиент так и не найдет экземпляр вашей базы!



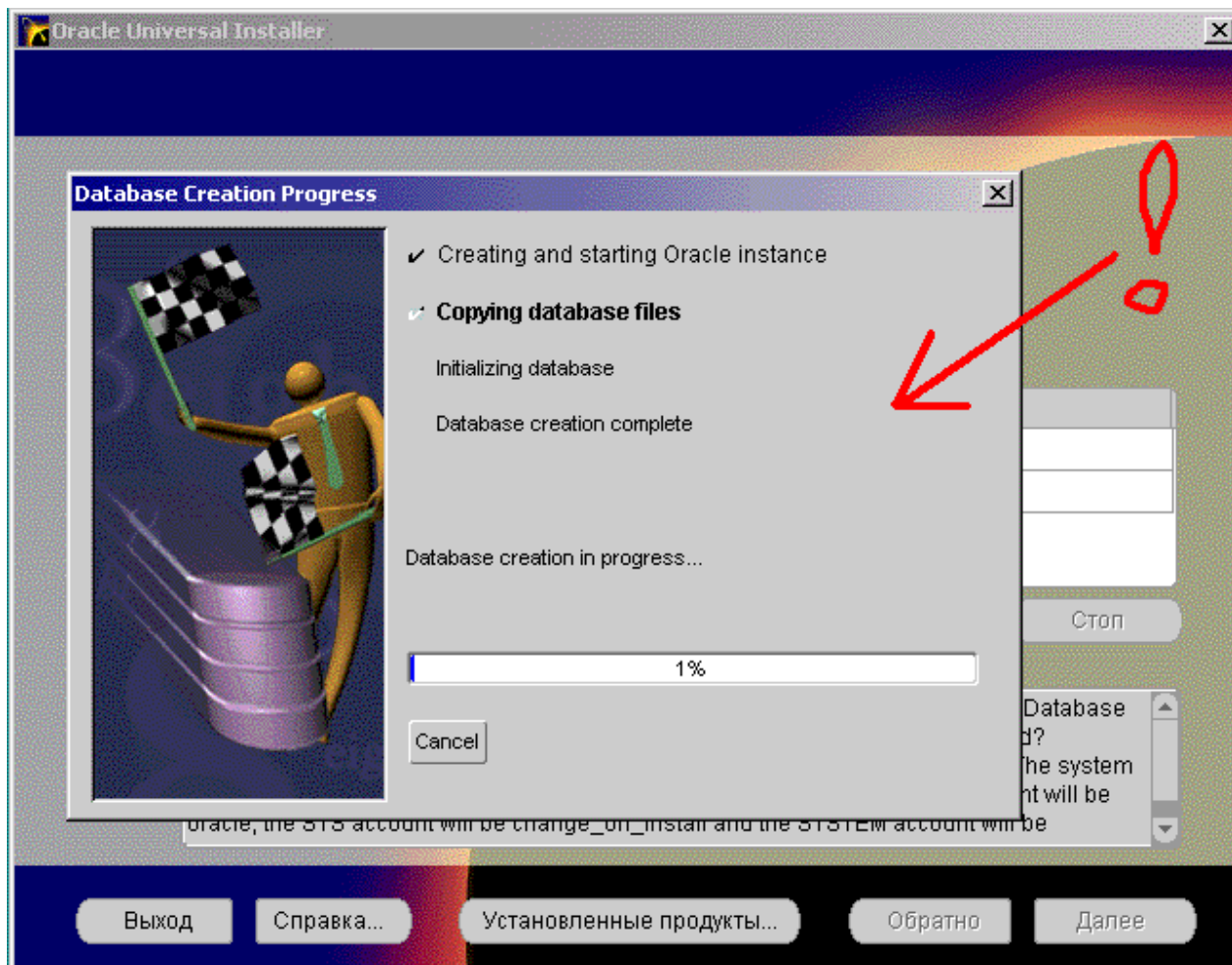
Вот и открылась сводочка, здесь проследите только чтобы язык установки был русский! А он и так русский, вот и славненько!!! Кликать далее завязываем, так как вот он и настал долгожданный миг и со всей силы Жмем "Установка"!!!



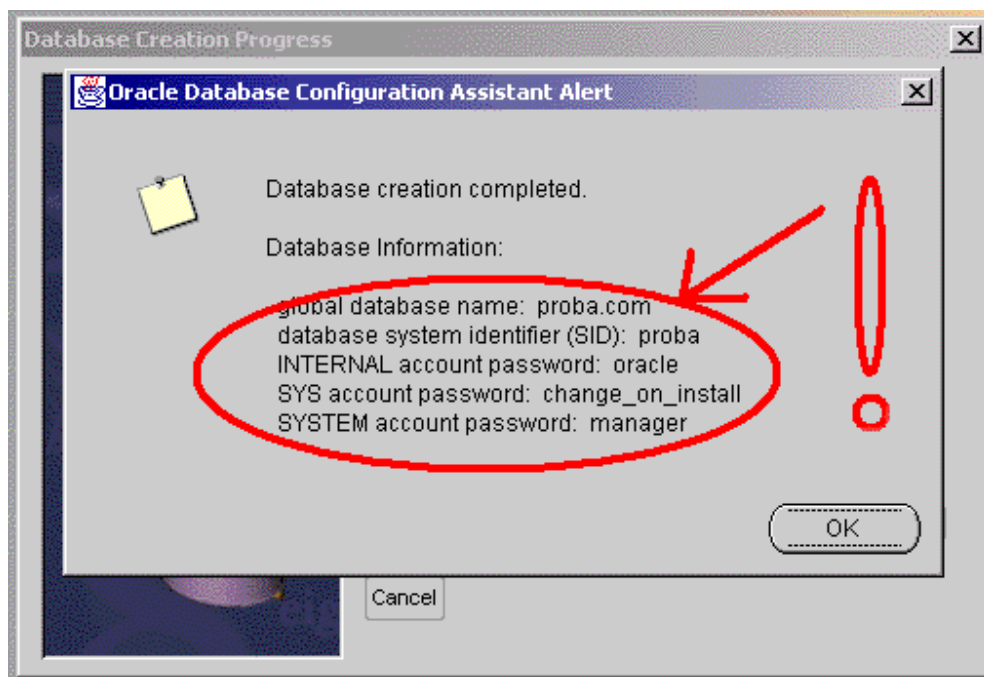
Вокруг черной дыры начнет крутиться планета, а прогрессбар начнет свое вечное синее движение вправо или голубое, а в прочем какая разница!!! Смотрим и наслаждаемся этим прекрасным зрелищем!!!



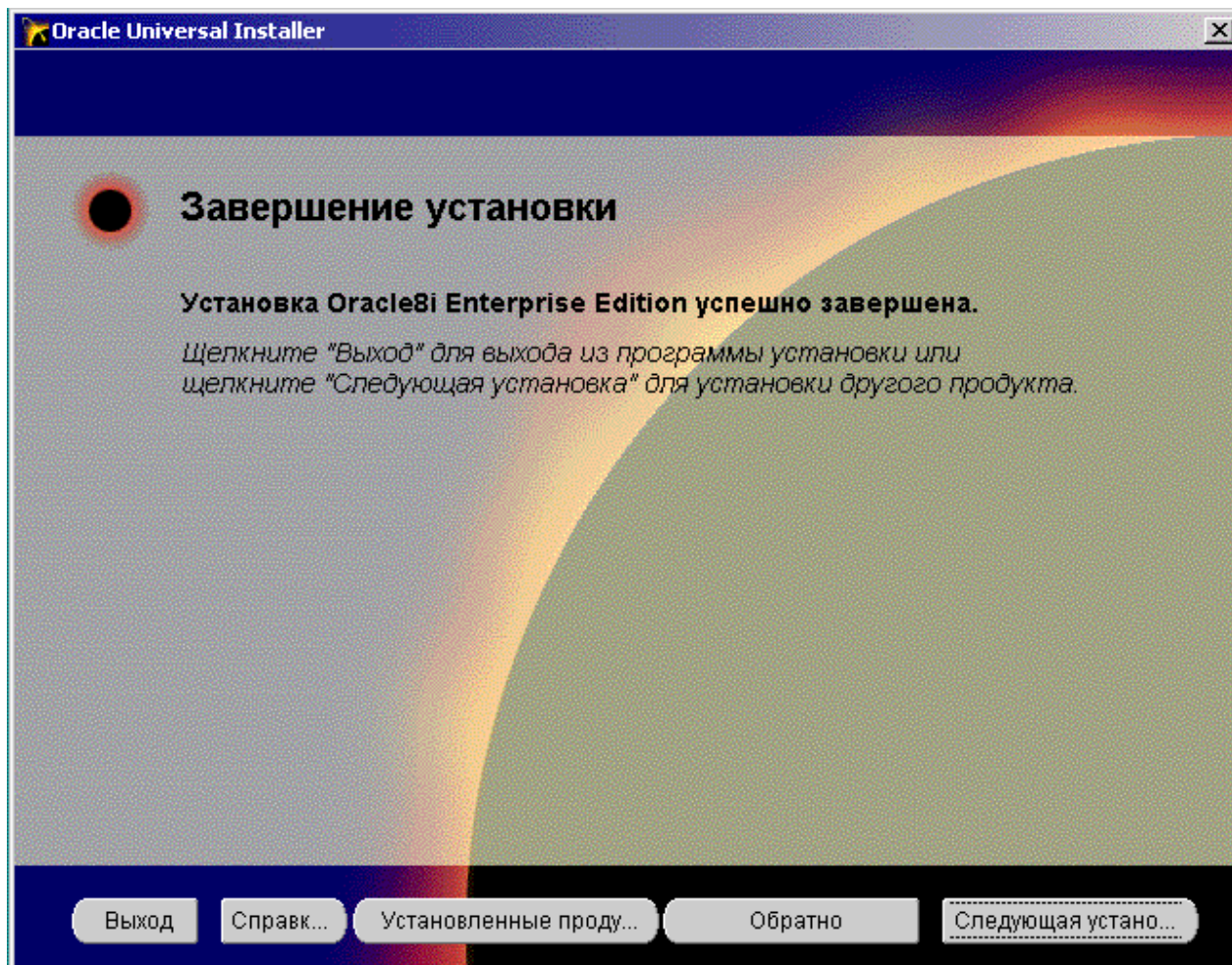
Полет планеты продолжается, но нас с вами уже благодарят за выбор сервера (показано стрелочкой) скрывая гордость продолжаем установку сервера!!!



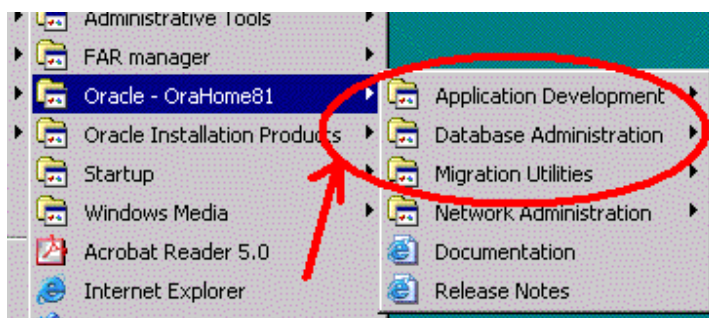
Так, а вот и ответственный момент все вроде бы само собой, но вот именно сейчас рождается экземпляр БД!!! Очень ответственный участок, да пока "накорми собак и ничего не трогай!!!" :))



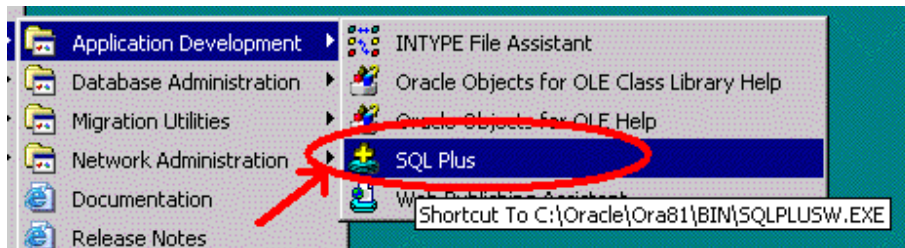
Ура!!! БД создана экземпляр запущен и готов к приему коннектов! Но вот здесь и начинается самое интересное. Вот это окошко рассказывает о трех танкистах, нет не о тех, что выпили по триста!!! А тех, которые отвечают за Администрирование только что новоиспеченного экземпляра БД!!! НЕ ТОРОПИСЬ НАЖИМАТЬ кнопку ОК!!! Выполни принтскрин и скопируй из буфера в паинтбраш, так я кстати и делал эти скриншоты!!! И вот почему. У этих трех, твои ключи от БД как администратора. Первого зовут **sys** он старший из братьев, его пароль по умолчанию **change_on_install**, второго зовут **system**, пароль по умолчанию **manager** ! Средний был и так и сяк, а вот младший совсем не дурак!!! Его зовут **INTERNAL**, его пароль (как его задавать это тема для отдельного шага) - правильно **ORACLE**! Кто они такие и с чем их едят, я еще расскажу, пока это все на данный момент, жми ОК!



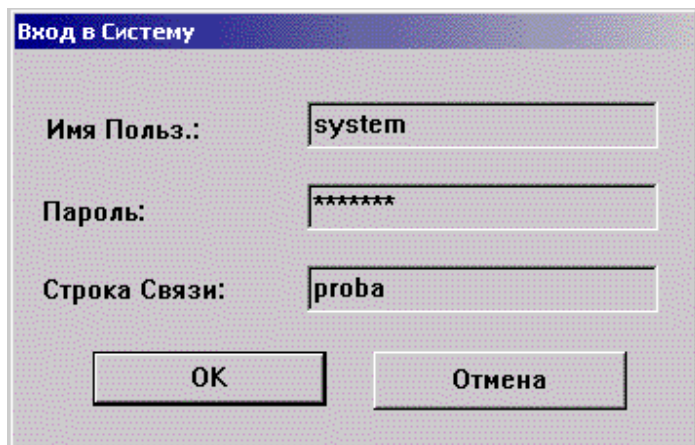
Вот собственно и все! Не в том смысле, что все вообще, а все в частности!!! Жмите "Выход", инсталляция сервера закончена!!! Но это только начало. Один маленький шаг для человека и огромный прыжок ... ладно! Об этом потом! :)



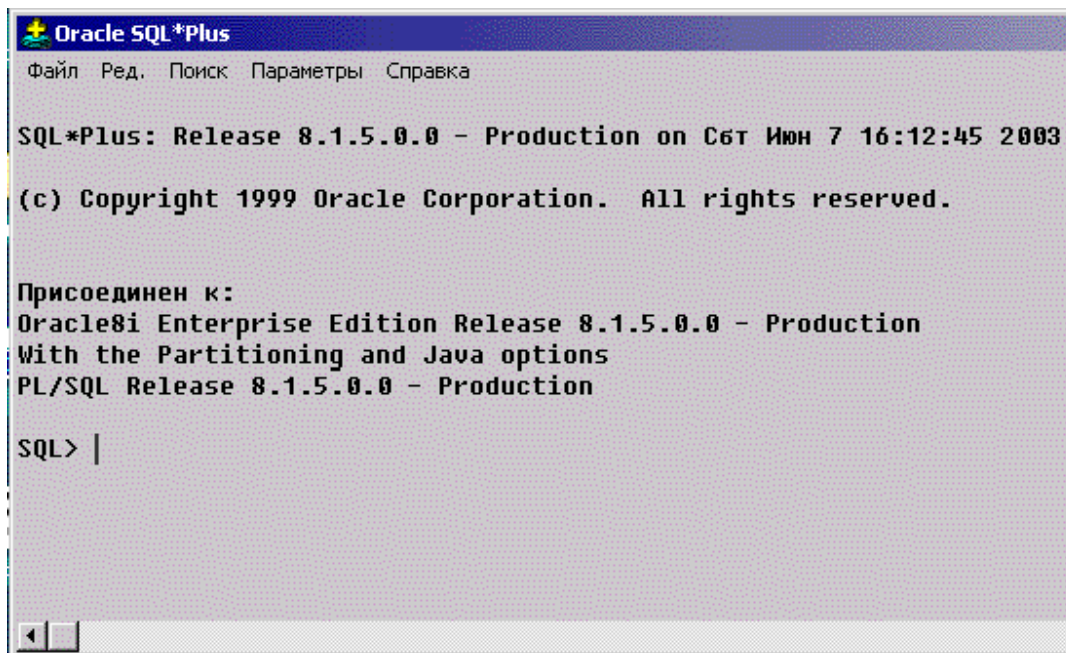
Нажимаем кнопку **Start** и убеждаемся, что все на месте !



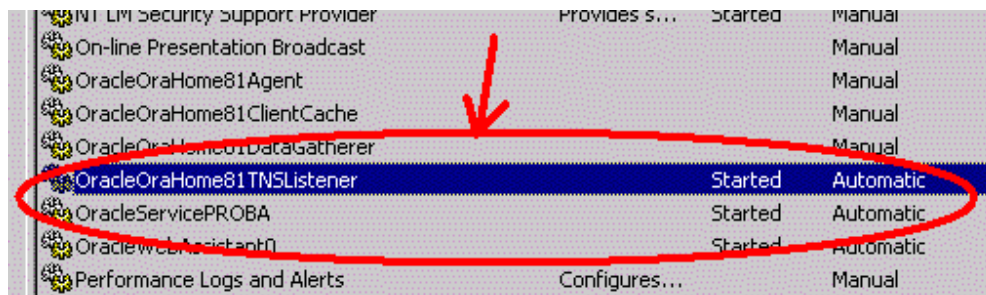
Теперь выбираем **Application Development** и запускаем **SQL Plus**!



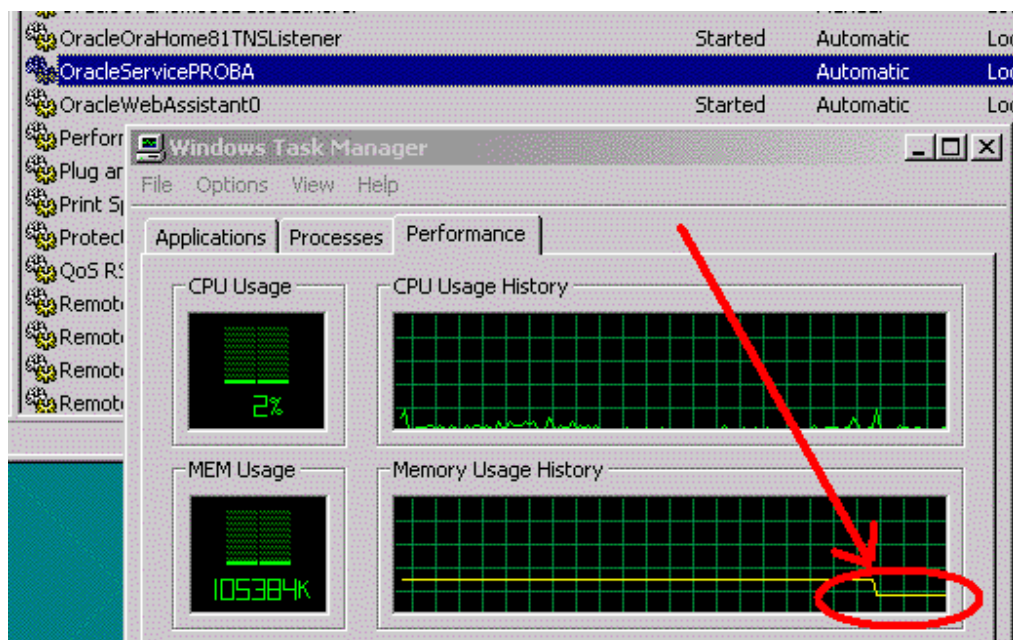
Надо убедиться, что сервер жив и с ним все в порядке. Вводим имя пользователя **System** и пароль **manager**, в строке связи **proba** или то, что было указано при установке в качестве **SID**'а базы, как ни странно они совпадают, почему я скажу позже. Нажимаем Ok.



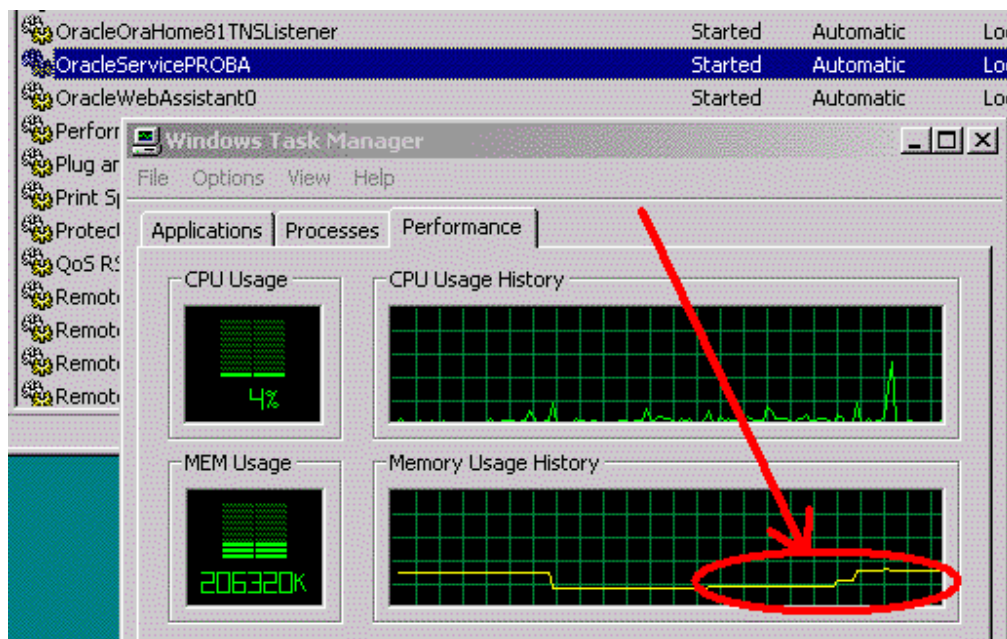
Вот теперь, если сервер установился и с ним все нормально должно появиться именно это!



А вот еще один способ, проверить как себя чувствует сервер. Надо открыть "Панель управления - Администрирование - Службы" и найти два сервиса, один в имени заканчивается так же как **SID** базы, а другой заканчивается **TNSListener**! Так вот они должны оба быть запущены и оба стоять **Automatic**!!! Кто из них, для чего пока разбирать не будем.



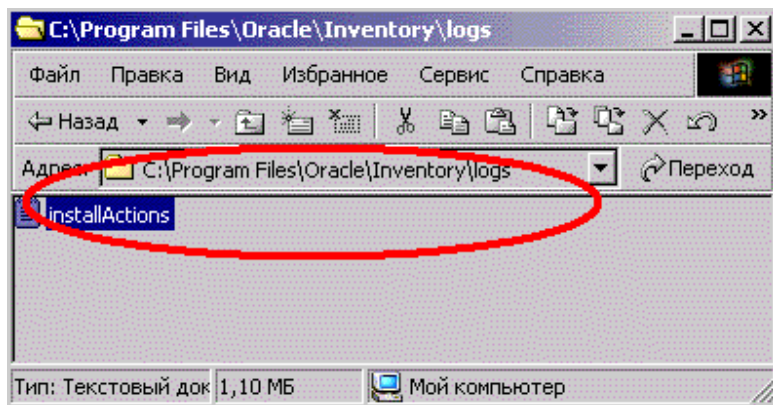
А сделаем вот что. Нажмем на первом (...PROBA) правой кнопкой мыши и выберем пункт меню **Stop**. Надо не забыть запустить Диспетчер задач! Если после того как процесс остановится желтая линия расхода памяти резко опуститься вниз, значит сервер УПАЛ! Нет, вернее просто экземпляр базы остановлен, но в этом нет ничего страшного :)



Теперь там же правой кнопкой мыши, только на этот раз **Start**, УРА!!! Память скакнула вверх! Экземпляр снова в работе. База доступна и снова ждет коннектов. Второй сервис пока не отключайте и остальные тоже пока не трогайте, с ними разберемся позже! Вот отсюда напрашивается вывод! Весь экземпляр БД живет в ОЗУ!!! И это одна из особенностей сервера **Oracle**! А, особенностей у него уйма в чем мы и убедимся в дальнейшем! На этом с установкой пока все!

Шаг 2 - Как выглядит Oracle в системе

Давайте отвлечемся от нашего повествования и немного поразмышляем. Если прошлый шаг прошел успешно, то в вашем **NT** сервере, все равно 4.0, 2000 или еще какой, появилась некая сущность, под именем **Oracle Server**. Он уже живет и работоспособен! Следовательно, если посмотреть на систему повнимательнее, то можно обнаружить, где и как он устроился и что делает в данный момент!



Начнем по порядку, откройте на вашем сервере, каталог **C:\Program Files\Oracle\Inventory\Logs** там, лежит файл **InstallActions.log**. Что видно из рисунка. Имя диска может быть другим, но положение именно такое.


```
Файл  Правка  Формат  Справка
installActions2003-06-01_09-16-32-AM.log
Переменные среды:
    ORACLE_HOME =
    PATH = C:\PROGRA~1\Borland\CBUILD~1\Bin;C:\PROGRA~1\Bor
    CLASSPATH =
Имя пользователя: sergl
*** Добро пожаловать Страница***установка значения FROM_LOCATION
*** Расположения файлов Страница***FromLocation = FROM_LOCATION
Версия установщика 1.6.0.9.0

Установка значения ORACLE_HOME в C:\Oracle\ora81
Установка значения ORACLE_HOME_KEY в Software\ORACLE\HOME0
Установка значения ORACLE_HOME_FOLDER в Oracle - oraHome81
Установка значения ORACLE_HOME_SERVICE в oraHome81
Выполняется установка данных установки
Установка значения TopLevelComp в oracle.server, 8.1.5.0.0, >0.0
Установка значения SELECTED_LANGUAGES в [ru]

*** доступные продукты Страница***
TopLevelComp = TopLevelComp = oracle.server, 8.1.5.0.0, >0.0, [
LangsSel = SELECTED_LANGUAGES = [ru]
Установка значения TopLevelComp в oracle.server, 8.1.5.0.0, >0.0
Установка значения DepMode в 0
Установка значения TLDepModes в [Ljava.lang.Integer;@19767f

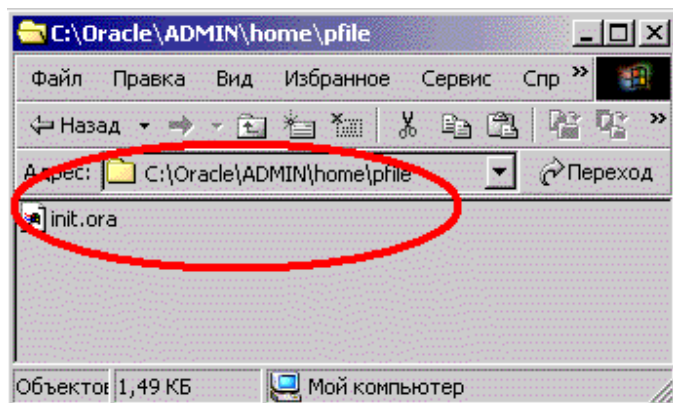
*** типы установки Страница***
DepMode = DepMode = 0
TLDepModes = TLDepModes = [Ljava.lang.Integer;@19767f
Установка значения DepMode в 0
Вызов запрос generalQueries1.6.0.8.1  isCurrentPlatformInGroup

platGroup = WINDOWS
Возвращенный запрос : true
Вызов запрос fileQueries1.6.0.4.0  exists

filename = C:\Oracle\ora81\ORAINST\nt.rgs
Возвращенный запрос : false
Вызов запрос fileQueries1.6.0.4.0  exists

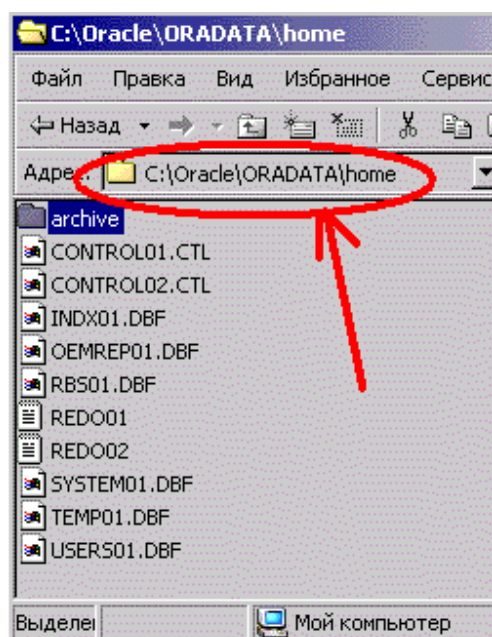
filename = C:\Oracle\ora81\ORAINST\win95.rgs
Возвращенный запрос : false
Вызов запрос fileQueries1.6.0.4.0  exists
```

Теперь посмотрим на фрагмент этого файла, я взял его со своей домашней машины, по этому, когда мы делали, в предыдущем шаге экземпляр, базы она называлась **proba.com**, теперь название сменится на **home.gov**, но в этом ничего страшного нет, можете использовать то, что написали вы, тем более фантазия у нас богатая! Что же мы видим в данном случае. Да просто то, как ставился наш экземпляр и когда и что при этом происходило! Вот собственно и все, но иногда информация, из этого файла может быть полезной!



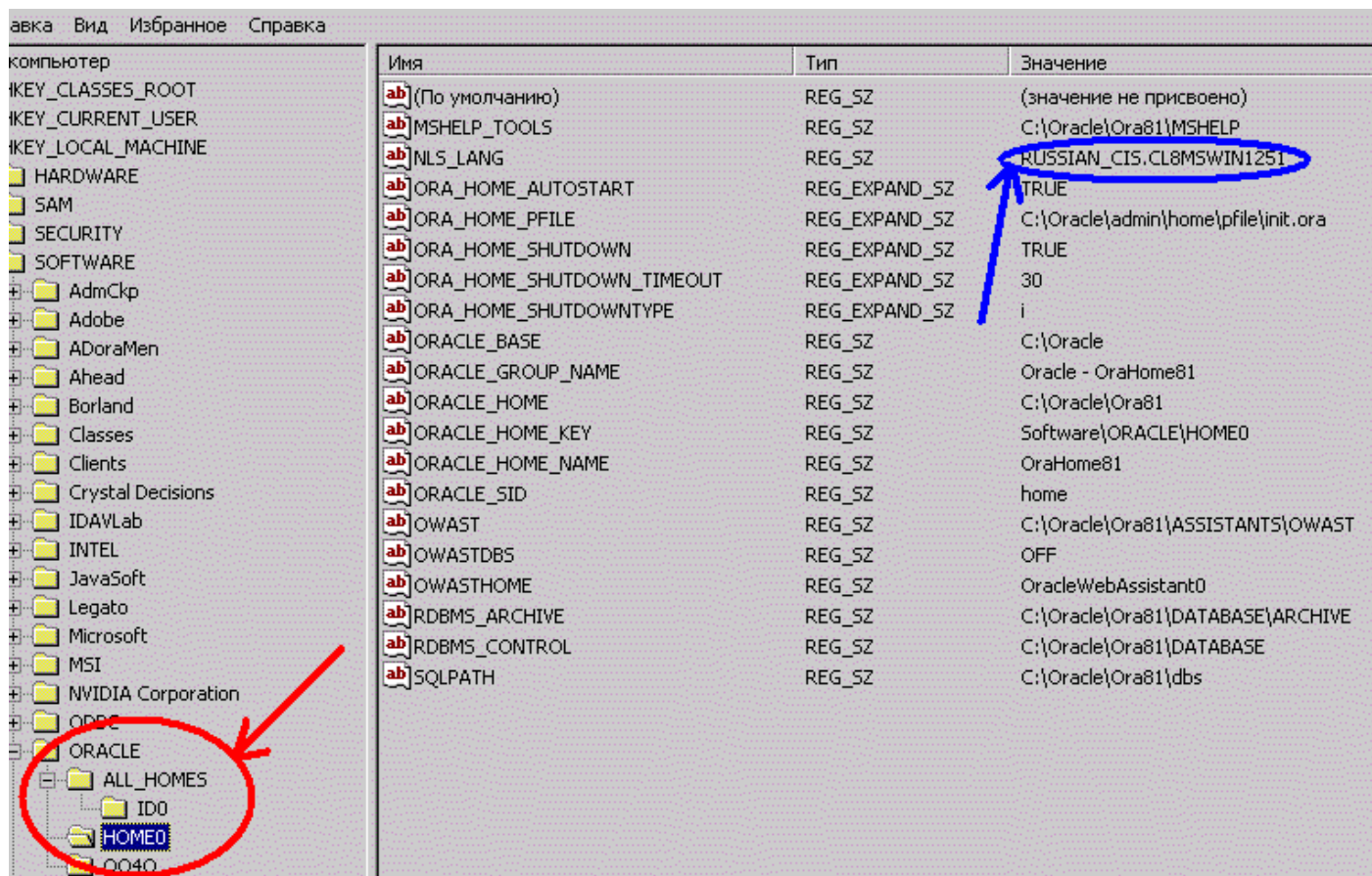
Теперь пойдем дальше, найдите каталог **C:\Oracle\ADMIN\home(proba)\pfile**. Вот мы и сделали шаг к самому ответственному файлу нашего экземпляра БД! Этот файл **init.ora**, как можно видеть из рисунка, и есть, так скажем, небольшой по своему объему файл инициализации вашего экземпляра! С его помощью, можно проделывать, довольно полезные вещи с БД, а именно настроить ее, так чтобы все "летало"! Вот уже слышу скепсис по поводу хранения настроек в файле! Ну что ж, компания производитель, тем не менее, пошла по этому пути и я думаю, правильно!

ПРЕДУПРЕЖДЕНИЕ!!!! Пока не пытайтесь, что-то менять в этом файле!!! Последствия будут самые печальные!!!



Теперь заглянем в каталог **C:\Oracle\ORADATA\home(proba)**. А вот здесь, живут файлы экземпляра вашей БД, давайте немного подробнее, посмотрим на то, что здесь находится! Так как в дальнейшем вам это будет полезно! Два файла с расширением **CTL**, являются "управляющими файлами", в них содержится вся схема БД. Файлы с расширением **DBF** (ничего не напоминает? только не вздумайте открыть их с помощью FoxPro! не советую) являются так называемыми файлами табличных пространств хотя такая формулировка несколько не верна как мне кажется, так как в них содержатся не только как можно подумать - таблицы, а и хранимые процедуры,

триггеры и еще много чего! Как правило, их имена, передают сущность организации экземпляра, а именно наличие пространств (а соответственно и файлов) как минимум - **SYSTEM, TEMP, USERS, INDEX**. Есть еще так называемые сегменты "отката", которые могут иметь собственный файл, а могут находиться и внутри других табличных пространств! Пока вы окончательно не запутались, давайте пока с этим закончим. :)



И вот напоследок очень аккуратно, откроем редактор реестра, и посмотрим, что же помещает **Server Oracle**, именно туда. Смотрим ключ **HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME0** синим, обведен ключ **NLS_LANG** как видно из картинки наш родной РУССКИЙ ЯЗЫК! Менять что-то здесь, я тоже пока не советую, хотя дальше мы кое-что сделаем! Но чуть позже. Вот собственно именно так и обустроился, ваш **Server Oracle** на вашем **NT** сервере. Теперь вы кое что, уже можете с ним вытворять, но в разумных, пределах!

Шаг 3 - Модель клиент-сервер

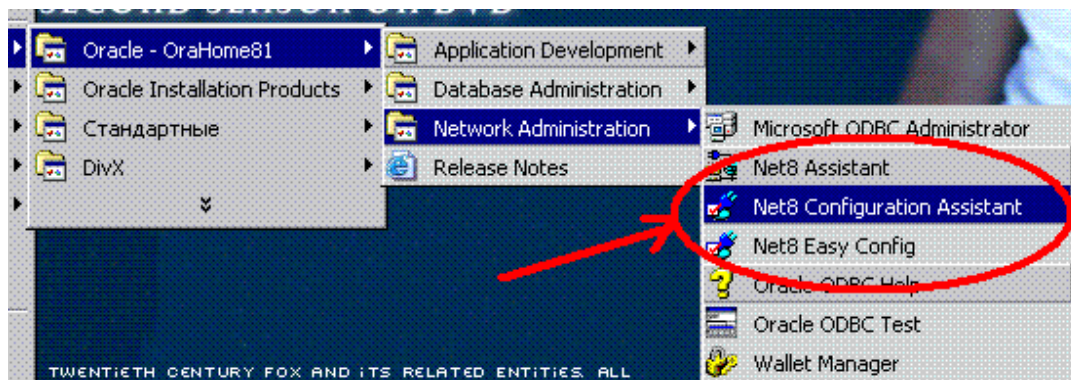
Вот теперь пришло время разобраться, что же такое модель КЛИЕНТ-СЕРВЕР? И понять, как вообще работают серверы баз данных основанных на этой модели.

Первые БД, на основе так называемых "файл серверных БД" базировались на концепции ФАЙЛ СЕРВЕР с расположенным на нем, файлом БД (dBASE, Paradox и.т.д) КЛИЕНТ, то есть машина пользователя имела, определенный софт, который подключался, например, по имени устройства сервера и выполнял определенную работу. Самое интересное было то, что в большинстве случаев, всю нагрузку по выборке, чтению, обновлению, поиску в такой БД, ложилась на процессор клиентской машины! А сервер предоставив в пользование файл с таблицами, просто прохладился в это время! Достичь, достаточной эффективности работы и целостности данных при такой схеме, построения БД, было не просто, если скажем терялся сетевой коннект, а строилась сеть на коаксиальном кабеле (как правило), то данные могли бесследно испариться. Чему я был неоднократно свидетелем, так как просидел на **dBASE** и **FoxPro** достаточно долгое время! Эх золотое было времечко! Погулять по этажам с "терминатором"! Нет не Арнольдом, а маленькой такой штучкой... Ладно не будем отвлекаться! И вот свершилось - появились клиент-серверные БД.

Характерная особенность таких серверов, это наличие "прослушивающего процесса"!

Само по себе это понятие довольно несложное, где-то в сети, там где работает сервер БД, запущен этот процесс, то есть ухо, которое слушает сетку, а вернее пакеты, которые в ней бегают и как только обнаружит запрос на соединение отвечает на него и предоставляет канал связи между сервером и клиентом, так вот один из сервисов, который я показывал, в предыдущем случае именуемый как **TNSListener(SID базы)**, как раз и является этим "чутким ушком", обеспечивающим соединения, сервера с клиентом!

Слышу резонный вопрос, - Ну и в чем разница? А вот в чем, существенная и основная! Концепт клиент-сервер, позволяет заставить, трудиться над поисками, обновлениями, удалением и черт знает еще чем, именно, СЕРВЕР БД! Клиент формирует запрос: - Найти всех участников пивного фестиваля, за 2000г. Сформировал, отправил и все, жди! Сервер выполнил, запрос, вернул результат клиенту! Трафик сети меньше, бесполезных шуршаний приводом тем более, да и быстрее на порядок, чем у файл-серверных БД! Хотя здесь, еще много можно поспорить, но пока принимайте на веру как есть!



Вот эти три пункта меню (если у вас все установлено правильно), как раз запускают три различных приложения, которые в вместе определяют всю работу клиента с сервером, а так же и на оборот! Сразу оговорюсь, если вы будете ставить, сервер на какой либо машине, то в принципе после

установки вы получите и клиента, а чего мелочиться, все ставится сразу. По этому можно общаться с сервером на машине где он установлен через клиента, да вы собственно этого и не заметите! Либо с другой машины в сети, предварительно установив на нее клиента! Пока надеюсь все понятно, вот и славно! Так вот остановимся пока на приложении **Net8 Easy Config**, это приложение, формирует файл находящийся в папке: **C:\Oracle\Ora81\NETWORK\ADMIN** и называется он **tnsnames.ora**. Давайте заглянем в него.

```
# C:\ORACLE\ORA81\NETWORK\ADMIN\TNSNAMES.ORA Configuration
#File:C:\Oracle\Ora81\NETWORK\ADMIN\tnsnames.ora
# Generated by Oracle Net8 Assistant

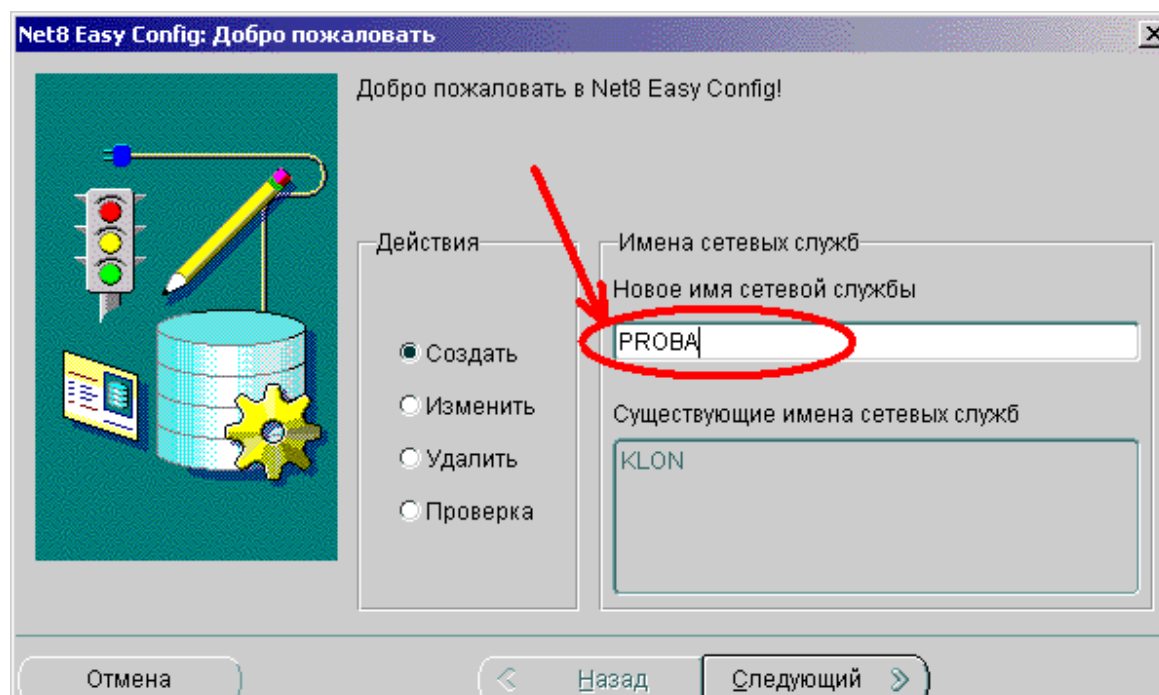
HOME =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = 127.0.0.1)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = proba.gov)
    )
  )
```

Пока наверное нифига не понятно, но это пока! Думаю примерно некоторые строки, что то напоминают, наверняка. А вот что они значат, разберем далее, предметно рассмотрев сам **Net8 Easy Config**! Ну, а пока осмыслите все, что я тут наболтал или попейте кофе и отдохните! :)

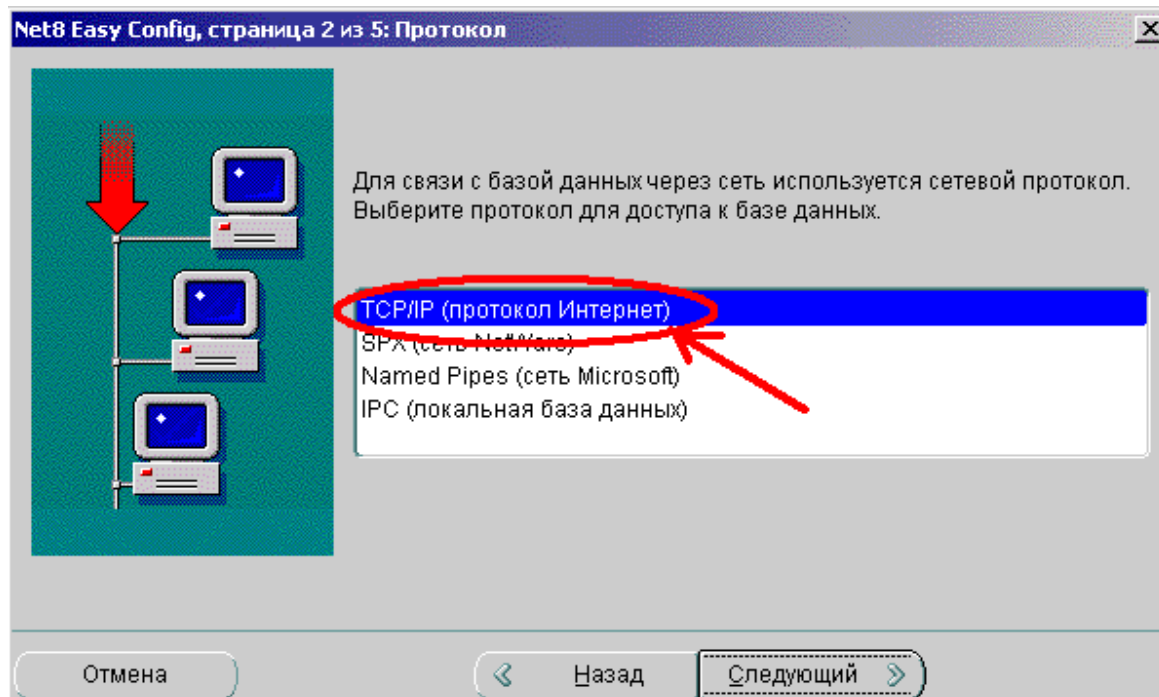
Шаг 4 - Настройка с помощью Net8 Easy Config

Давайте подробнее рассмотрим организацию взаимодействия сервера и клиента с использованием службы **Net8i Oracle Server**. И что означают строки внутри файла **tnsnames.ora**, который я приводил в [прошлый раз](#). Сам файл можно формировать и в ручную, если вы не знаете, как это делать, то тогда лучше воспользоваться утилитой **Net8 Easy Config**, которую мы сейчас и разберем. К стати в **9i** ее уже нет, но разобраться будет полезно так как, служба **Net8i** никуда не делась!

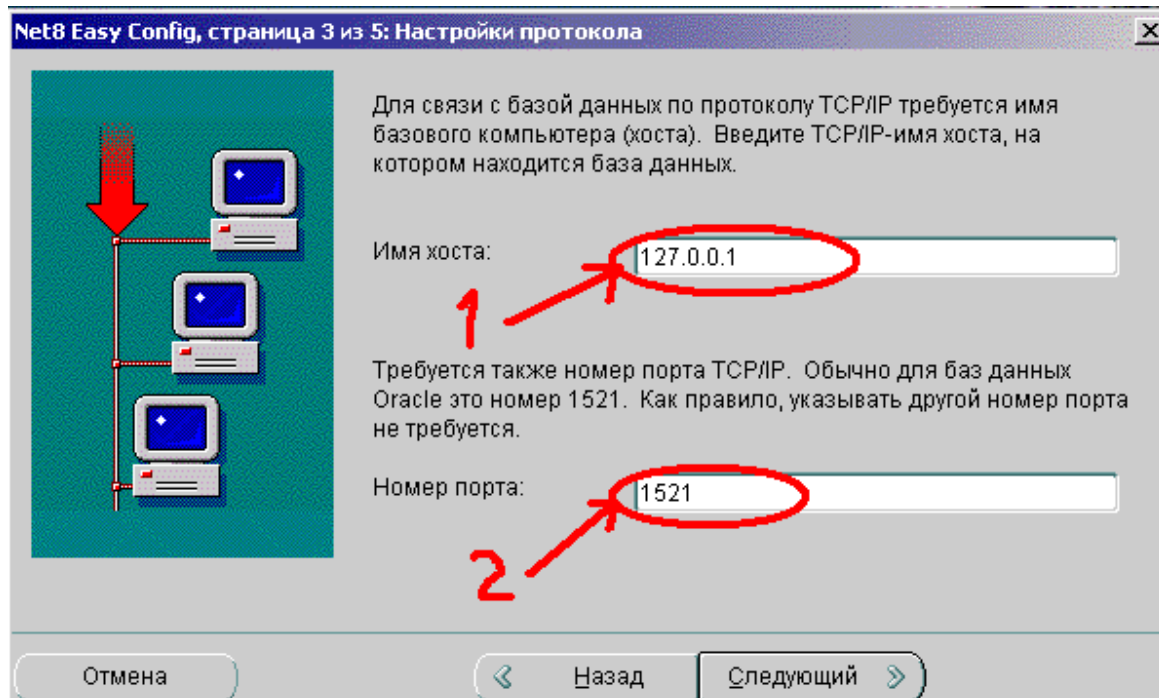
Итак, приступим:



В поле "Новое имя сетевой службы" необходимо ввести имя, которое в дальнейшем будет использоваться для подключения к **Oracle Server**. Само имя в конечном итоге роли не играет, здесь можете полагаться на свою фантазию. Я написал **proba**, но если вы установили **Oracle** на свой **NT** сервер и действовали, так как я вам предлагал, то есть использовали глобальное имя БД **proba.com**, то у вас в поле "Существующие имена сетевых служб", уже будет прописано такое имя и именно по тому, что когда мы используем типовую установку **Oracle Server** при создании экземпляра БД, автоматически, создается, сетевая служба **Net8i** для подключения к **Oracle Server**, именно с именем, совпадающим с **SID** вашего экземпляра БД. (Напомню, что **SID** глобальное имя экземпляра, это всегда первая часть, глобального имени сервера **proba(SID).com(full name)** учитывая тот момент, что один сервер **Oracle**, может содержать несколько экземпляров БД!) Пока постарайтесь понять, что я тут наболтал так как мы к этому еще вернемся! Итак, придумали имя? Вписывайте и смотрим следующий рисунок!

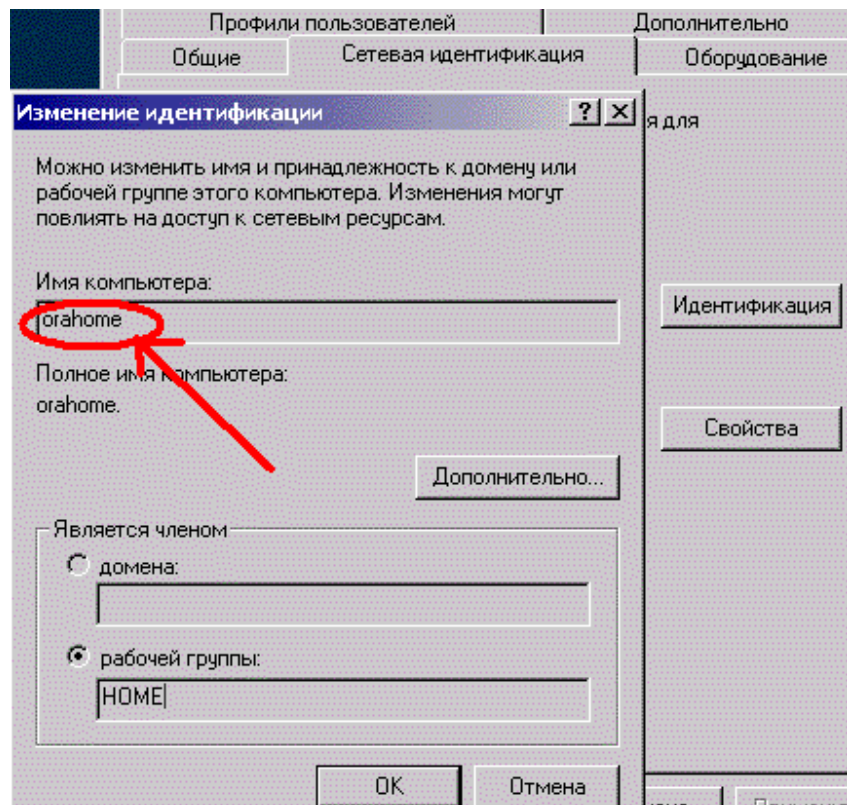


Естественно выбираем протокол **TCP/IP**, я думаю, что смысл выбора очевиден и в комментариях не нуждается! :)

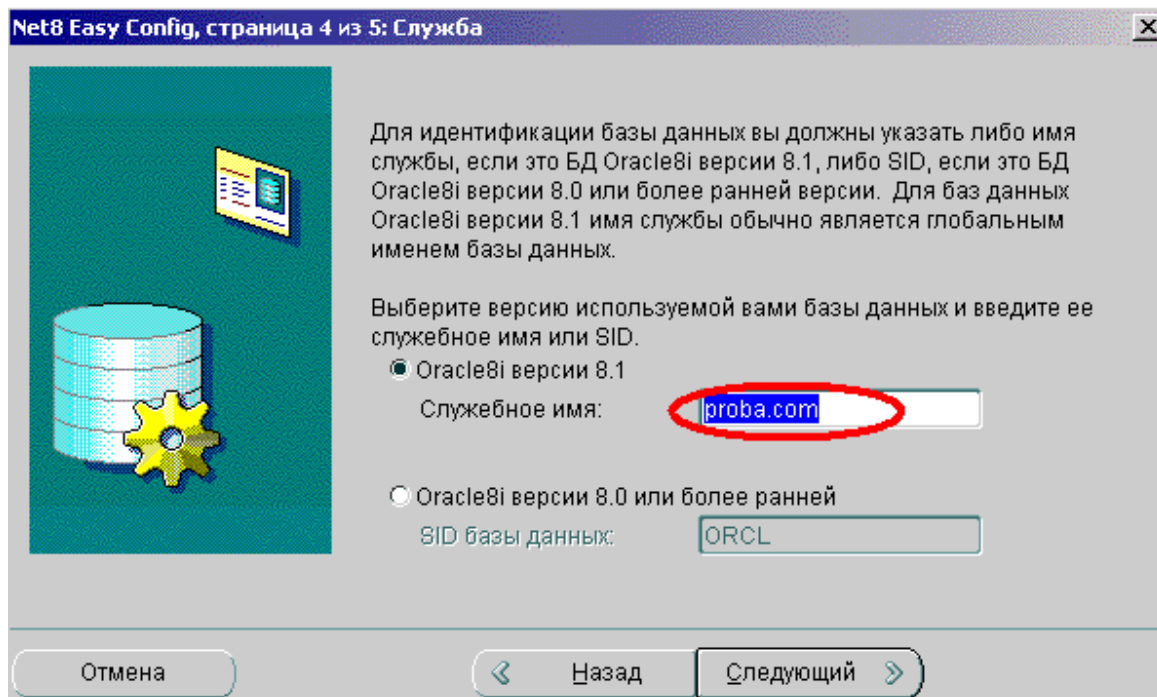


А вот здесь, немного прокомментируем. Первое "Имя хоста", я написал **127.0.0.1** так как предполагаю, что ваш **Oracle Server**, установлен на той же машине на которой мы и будем работать дальше, если вы установили **Oracle Server**, где то в сети или он у вас уже установлен

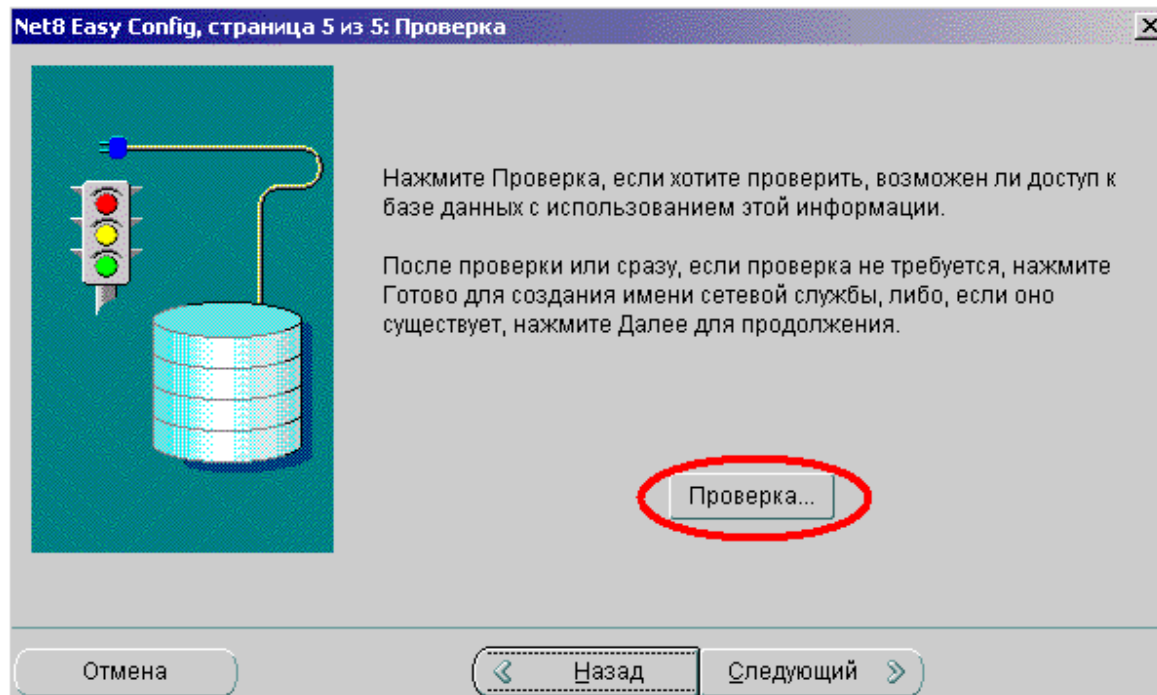
может быть и не вами, но вы все-таки читаете мои "записки сумасшедшего", в надежде получить какие-то знания и научиться работать с **Oracle**, то на вашей рабочей станции должен стоять как минимум клиент **Oracle**, который устанавливается почти так же как и сервер, но несколько быстрее и он тоже содержит все необходимые утилиты **Net8i**. Так вот, если это так, то впишите в этом поле **IP** вашего сервера, либо его глобальное сетевое имя, которое легко найти открыв свойства вашего **NT** сервера как на рисунке ниже.



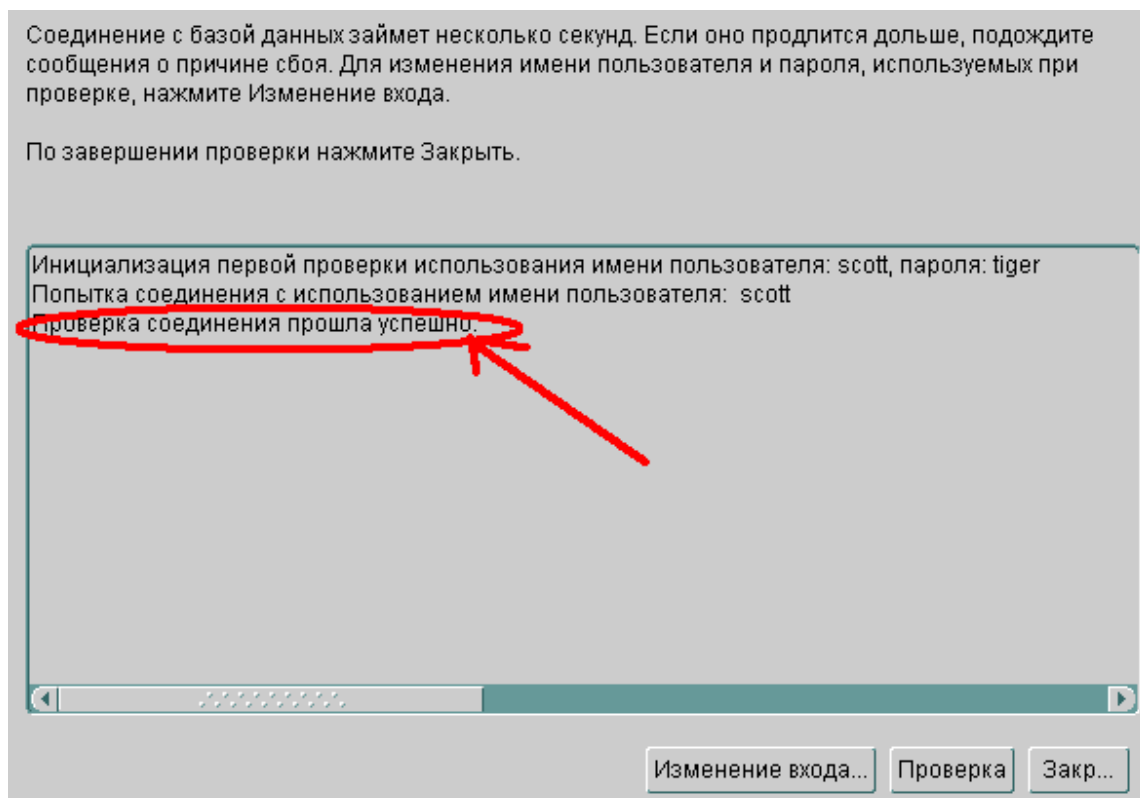
А вот поле "Номер порта", разумнее оставить как есть, а именно 1521, это именно тот порт который использует, "прослушивающий процесс", его можно и изменить, но пока в этом нет особой необходимости и мы не будем это делать!



Вот здесь, выбрана опция **Oracle8i** версии 8.1 "Служебное имя". Здесь необходимо вписать полное служебное имя вашего **Oracle Server**, которое вы выбрали при инсталляции, если не изменяет память **proba.com**, что собственно и видно на рисунке! Если вы написали, что то другое, то скорее вспоминайте, что и вписывайте!!!

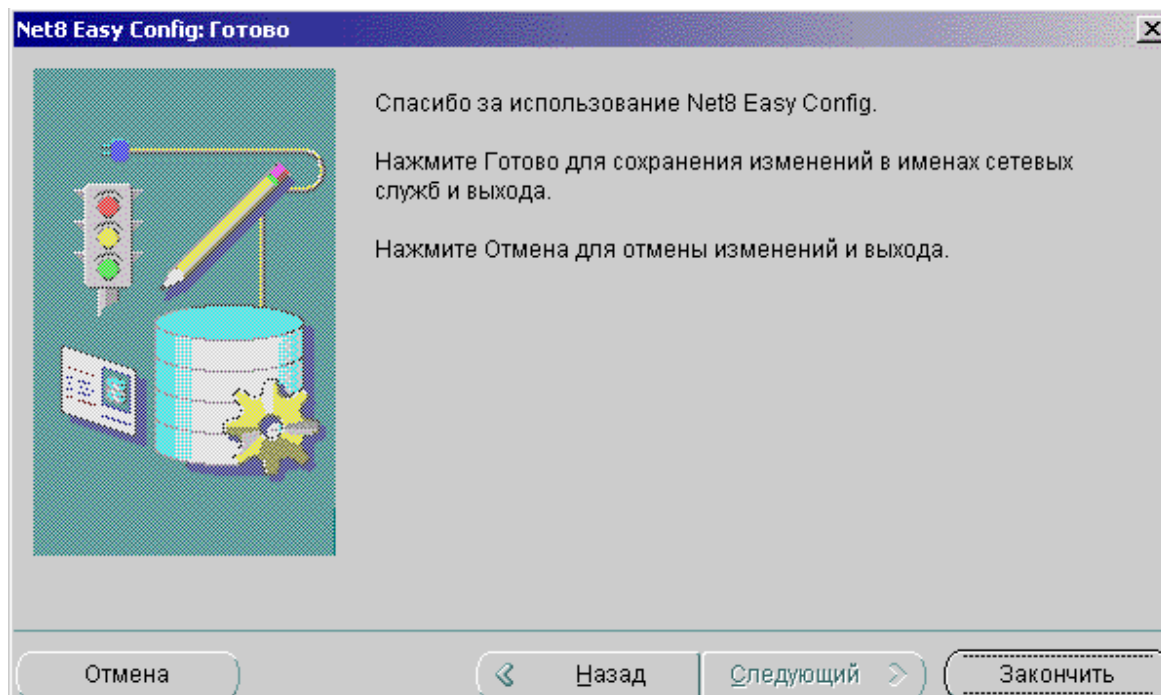


Ура! Почти все! Жмем кнопку "Проверка", если видим то что на рисунке ниже, то можно вздохнуть спокойно!



Проверка прошла успешно! Вот и замечательно!

Если при проверке была ошибка, то можно вернуться на начало и все проверить, если и при этом ошибка, то это еще не фатально! И вот почему, если вы отступали от моих инструкций при инсталляции **Oracle Server** или как я уже говорил, пытаетесь работать с уже установленным кем-то сервером, то вероятно, что схемы **Scott** на вашем экземпляре нет! По этому, жмите изменение входа, например с именем **system** и паролем **manager**! Но не забывайте, что можно получить по шее, от вашего админа БД, за такие эксперименты!!! :) Ладно, будем считать, что все прошло успешно!



Жмите "Закончить"!

Файл **tnsnames.ora** сформирован.

```
HOME =  
  (DESCRIPTION =  
    (ADDRESS_LIST =  
      (ADDRESS = (PROTOCOL = TCP)(HOST = 127.0.0.1)(PORT = 1521))  
    )  
    (CONNECT_DATA =  
      (SERVICE_NAME = proba.com)  
    )  
  )
```

Надеюсь теперь, стало понятнее назначение его секций! В 4-й строке протокол конечно же, **TCP/IP**, хост, имя вашего **NT** сервера, порт, номер порта, прослушивающего процесса, **Oracle Server**! **SERVICE_NAME** полное имя сервера БД **Oracle**. Вот собственно и все, я умолчал пока о том, что с помощью, **Net8i** можно настроить, брандмауэр, так же некое подобие роутера и еще кое что! Но если вам не наскучит, то что я вам повествую, я расскажу и об этом! Пока, вы должны четко представлять, как формируется коннект, для **Oracle** и уметь его настраивать, так как в дальнейшем, вам это не раз понадобится! Есть еще два файла, которые мы разберем позднее, так же относящихся к службе **Net8i**, вследствие того, что на нее замыкается достаточно много приложений в **Oracle Server**! А пока можете поэкспериментировать с **Net8 Easy Config**, создавая и удаляя службы! Но в разумных пределах!

Шаг 5 - Командная строка SQL Plus

Что, пошли дальше? База созданная типовой установкой, работает!? Замечательно! Теперь давайте попробуем сделать что-то полезное! Тем более, что уже руки наверное чешутся! Думаю пришло самое время обратить внимание на приложение с именем **SQL Plus**!

Запускайте! В [первом шаге](#) мы уже запускали **SQL Plus**! Но только для того, чтобы убедиться, что все нормально. Сейчас с помощью "Плюса", мы попробуем научиться самым азам работы с сервером БД и немного поизображать админа БД!!! Итак, запустили?

В поле "Имя польз.:", пишем **system**, в поле "пароль:" пишем **manager**, а вот в поле "Строка связи:" пишем то самое имя сетевой службы, созданное в [прошлый раз](#)!!! Если мне не изменяет память, то звали ее **PROBA**! Если все прошло хорошо, то мы увидим примерно следующее:

```
-----  
SQL*Plus: Release 8.1.5.0.0 - Production on Пнд Июн 23 22:13:55 2003
```

```
(c) Copyright 1999 Oracle Corporation. All rights reserved.
```

```
Присоединен к:  
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production  
With the Partitioning and Java options  
PL/SQL Release 8.1.5.0.0 - Production
```

```
SQL> |(мигает!)
```

Это значит, что все прекрасно, и вы как администратор допущены в самое сердце вашего сервера **Oracle**! Объясню сразу и без вариантов, здесь будет рассматриваться работа и примеры в командной строке **SQL Plus**! Никаких красивых, визардов и волшебников не БУДЕТ!!! И вот почему, если вы действительно хотите усвоить довольно мощную БД **oracle**, а так же ее встроенный язык программирования **PL/SQL**, то начинать будем, с КОМАНДНОЙ СТРОКИ! Я в свое время навидался, крутых, навороченных, молодых людей, которые понятия не имели, не то что о возможности, а даже не подозревали, что каталог создается, в командной строке **DOS**! :) К стати вопрос на засыпку, а как создать каталог на диске в строке **DOS**?

Так что, тот, кто привык ставить галочки в визардах, советую, немного напрячь воображение или... ну это уже решать вам! :) Так же могу сказать, что **SQL Plus**, действительно имеет ряд недостатков и конечно же в дальнейшем, мы перейдем на что-то более передовое, Но!!! Как инструмент для истинного знатока администрирования БД **Oracle**, это то что, как раз и достаточно! Вот так!!! :) Ладно, хватит болтать, пора действовать!

В ответ на это приглашение

```
SQL>
```

Введите примерно следующее:

```
SELECT COUNT(*) FROM USER_OBJECTS
```


Нажмите **Enter** и увидите:

```
SQL> SELECT COUNT(*) FROM USER_OBJECTS
2
```

Это интерпретатор команд предлагает ввести следующую строку для вашего выражения если написать, что-то еще, то появится:

```
3
4
и т.д.
```

То есть среда предлагает вам ввести выражение построчно столько сколько нужно! Например, если вы намереваетесь, записать какой либо сложный запрос!

Да, но нам нужно выполнить только один короткий оператор и увидеть результат!

Да запросто! Там где было 2, введите символ "/" и снова нажмите **Enter** это означает, что вы закончили ввод и собираетесь, получить результат!

Я получил примерно следующее:

```
COUNT(*)
-----
64
```

Что значит, что, пользователь **SYSTEM**, имеет в своей схеме 64 объекта БД! Если фраза не совсем понятна, пока не напрягайтесь все будет далее!!!! :)

Сейчас в ответ на приглашение **SQL>** введите еще раз "/" и **Enter**!

И опять:

```
COUNT(*)
-----
64
```

Так можно делать хоть целый день! Но боюсь, что вас примут за ..., а не важно!!! :)

Это значит, что наше выражение где-то внутри и поле ввода "/" может выполняться хоть сто раз! Еще выражение "/" можно заменить, написав **RUN** вот так:

```
SQL> RUN
1* SELECT COUNT(*) FROM USER_OBJECTS

COUNT(*)
-----
64
```

Видите, теперь мы снова увидели выражение, которое содержится в строковом буфере и поле нажатия **Enter** не нужно больше ничего!

Если же вы просто хотите посмотреть весь строковый буфер **SQL Plus**, то введите примерно следующее:

```
SQL> LIST
```

Получаем:

```
SQL> LIST
1* SELECT COUNT(*) FROM USER_OBJECTS
```

А вот и сам строковый буфер!

Еще она полезная команда **SQL Plus** это **EDIT**, на ней остановимся подробнее.

Вводим:

```
SQL> EDIT
```

После нажатия **Enter** появится Блокнот, мое любимое орудие при написании скриптов **PL/SQL!!!**
В окне блокнота, будет строка:

```
SELECT COUNT(*) FROM USER_OBJECTS
/
```

Если ничего не менять, то она же и останется, если вместо нее написать, например:

```
SELECT COUNT(*) FROM USER_TABLES
/
```

То при закрытии будет задан вопрос: Сохранить изменения в файле **afiedt.buf!**

Вот он то и содержит, то, что хранит буфер, а лежит он обычно в каталоге **C:\Oracle\Bin!**
Говорим да сохранить, и видим следующее:

```
1* SELECT COUNT(*) FROM USER_TABLES
SQL>
```

Что-то не то, а! Вводим **"/"**! Упсс, получилось!

```
COUNT(*)
-----
17
```

У пользователя **SYSTEM** 17 таблиц в схеме! Просто замечательно! Еще можно делать следующее, создайте каталог на вашем диске, скажем **Temp** и поместите в него файл содержащий следующее:

```
C:\Temp\proba.txt
Hello world!!!
```

Введите:

```
SQL> EDIT C:\temp\proba.txt
```

Откроется Блокнот и покажет содержимое файла, вроде бы бесполезно, но не совсем!!! дальше пригодится! Да, напоследок еще пара полезных команд, если экран **SQL Plus** сильно захламлен введите **CLEAR SCREEN** и будет чисто! Если надоели команды в буфере напиши **CLEAR BUFFER** и привет!!! Все очистится! Пока можете потренироваться с вышеизложенным, но советую сменить пользователя, так как работа в этом может при неосторожных действиях завалить сервер!!! Как менять пользователя на ходу и кое что еще в [следующем шаге](#)!

Шаг 6 - Создание пользователя и настройка прав доступа

Продолжаем работать с **SQL Plus!** Попробуем сделать следующее, войти пользователем **SYSTEM** с паролем **manager**, а затем не закрывая плюс сменим действующего пользователя.

Запускаем плюс, вводим пользователя, пароль и название сетевой службы (**proba!** или что-то еще!), получилось? Замечательно! Теперь проделаем примерно следующее:

```
SQL> CONNECT SCOTT/TIGER@PROBA
Соединено.
SQL>
```

Команда **CONNECT** производит подключение к серверу требуемой схемы(пользователя) в соответствии с заданной строкой соединения! Вот так:

```
USER/PASSWORD@NETWORKSERVICE
```

- **USER** - это пользователь(схема в экземпляре БД).
- **PASSWORD** - пароль для входа в схему.
- **NETWORKSERVICE** - имя службы сформированное программой **Net8i**.

В предыдущем случае мы зашли в схему(пользователя) **SCOTT** с паролем доступа **TIGER** с помощью сетевой службы **PROBA**. Берите сразу на заметку или на память: в **Oracle Server** в паролях доступа не допускаются цифровые символы! Так же созданный пользователь, создает схему в экземпляре БД и понятие схема и пользователь в **Oracle** практически тождественны! Сама строка подключения, вами еще не однократно, будет использоваться в дальнейшем! Теперь, я думаю пришло время, создать собственную схему, тем более она нам понадобится, в дальнейшем, для того, что бы научиться использовать **PL/SQL**! Первое и самое простое, действие, для создания нашей схемы, ввести следующее:

```
SQL> CONNECT SYSTEM/MANAGER@PROBA
Соединено.
```

Для начала заходим на сервер, как администратор!!!

Вводим ниже приведенную строку, которая создает пользователя **MILLER** с паролем в системе **KOLOBOK** (можете написать свое!), который будет жить в табличном пространстве **USER** владея им целиком и захватив в свое распоряжение еще кусочек табличного пространства **TEMP**, так на всякий случай, пригодится!!!

```
SQL> CREATE USER MILLER IDENTIFIED BY KOLOBOK DEFAULT TABLESPACE USERS
2 QUOTA UNLIMITED ON USERS QUOTA 2M ON TEMP
3 /
Пользователь создан.
```

После нажатия **Enter** на последней строке видим, что все прошло удачно!

Но, это только полдела, теперь этому пользователю, нужно, дать ряд прав и первостепенное, это создавать сессию с сервером! Теперь введем нижеследующее: Можно по очереди или целиком!

Главное, чтобы сработал последний оператор **COMMIT!!!** Иначе наши старания пройдут бесследно!

```
GRANT CREATE SESSION TO MILLER
/
GRANT CREATE TABLE TO MILLER
/
GRANT CREATE PROCEDURE TO MILLER
/
GRANT CREATE TRIGGER TO MILLER
/
GRANT CREATE VIEW TO MILLER
/
GRANT CREATE SEQUENCE TO MILLER
/
GRANT CREATE VIEW TO MILLER
/

GRANT DELETE ANY TABLE TO MILLER
/
GRANT DROP ANY TABLE TO MILLER
/
GRANT DROP ANY PROCEDURE TO MILLER
/
GRANT DROP ANY TRIGGER TO MILLER
/
GRANT DROP ANY VIEW TO MILLER
/

GRANT ALTER ANY TABLE TO MILLER
/
GRANT ALTER ANY TABLE TO MILLER
/
GRANT ALTER ANY PROCEDURE TO MILLER
/
GRANT ALTER ANY TRIGGER TO MILLER
/

COMMIT
/
```

В результате получим кучу сообщений типа: "Привилегии предоставлены."

И последнее: "Фиксация обновлений завершена."

Операторы **GRANT** предоставляют пользователю, определенные привилегии. В типах привилегий пока, предметно разбираться не будем, скажу только, что данное мероприятие можно проделать еще проще, если собрать все строки, которые мы вводили в файл, затем используя команду **START** или операцию "@"! Можете проделать это сами, предварительно введя, находясь в схеме **SYSTEM**:

```
SQL> DROP USER MILLER CASCADE
```

2/
Пользователь удален.

Затем соберите все строки, в файл, скажем **CrMiller.sql**, поместите его в каталог, например, **Temp**, и введите следующее:

```
SQL> @C:\TEMP\CRMILLER.SQL
```

Выскочит множество надписей, последняя из которых должна быть: "Фиксация обновлений завершена." Значит, все прошло нормально и пользователь создан! Далее в схеме **MILLER**, мы развернем, ряд объектов БД и посмотрим как это будет происходить!

Так же на заметку в заключении **SQL Plus**, есть еще много внутренних команд, например, очень полезной может оказаться **SET TIME ON** приглашение примет вот такой вид:

```
14:09:52 SQL>
```

Например, можно оценивать время на запрос из таблицы!

Если ввести **SET TIME OFF**, то все станет по прежнему. Например, если написать **SHOW USER** (мне напоминает Cisco IOS!), то увидим примерно следующее:

```
SQL> SHOW USER  
USER имеет значение "MILLER"
```

По ходу дела, мы познакомимся с большинством из них!

Шаг 7 - Язык SQL и примитивные типы данных

Думаю теперь настало время немного пофилософствовать, на тему **Structured Query Language (SQL)**! Что такое структурированный язык запросов баз данных и откуда он вообще появился?

Сам язык **SQL** родился в недрах компании **IBM**, при создании СУБД **DB2** и наиболее, широко применялся в **UNIX** системах на машинах с **RISC** процессорами, а так же на так называемых мэйнфреймах, больших вычислительных комплексах на базе суперкомпьютеров. По некоторым утверждениям, которые я встречал, не однократно, звучит примерно, следующее: "-язык SQL является языком программирования", что ж можно конечно, поспорить, но пока примем на веру! Я бы сказал, что скорее, это именно структурированный язык, для выполнения операций над БД!

В то же время не являясь самостоятельным, обычно, он инкапсулируется во внутренние языки программирования, такие как, например **PL/SQL**, или **Transact-SQL**! Нельзя найти **SQL**, сам по себе! В 1986 году, **ANSI (American National Standard Institute)** опубликовал, официальный стандарт, языка **SQL**, а в 1992 году, он был расширен. Откуда и появились стандарты **SQL91**, **SQL92**. Вот в принципе кратенько как все это начиналось! Для того что бы, в дальнейшем нам разобраться более подробно, с основными операторами **SQL**, лучше сделать, это не привязываясь к конкретной платформе БД, а пройдя это отдельным курсом. А, в дальнейшем, после того как все разберем, плавно перейдем к **PL/SQL** и я вас уверяю, вот здесь и будет самое интересное!

Начинаем по порядку! Весь язык **SQL**, имеет около 30 операторов, в разных реализациях, может быть чуть больше или чуть меньше. Так же любая БД, имеет ряд различных объектов, такие как таблицы, процедуры, функции, представления, последовательности и т.д. Описывать, сразу все операторы я думаю, не имеет смысла, получается слишком сухо и мало понятно. По этому сделаем немного иначе. Так же весь синтаксис и применение, будет на базе **Oracle**, по этому сразу привыкайте!

Все объекты БД имеют уникальные имена, а БД **Oracle**, так же имеет такое понятие как атомарная матрица (матрица!!!), однозначно идентифицирующая, каждую запись в БД! Идем дальше. Если вы обращаетесь к столбцу **SALES** таблицы **SALESREPS**, при организации запроса из одной таблицы, то все правильно, если же запрос производится из двух таблиц, где столбцы имеют одинаковые имена, то применяется запись такого вида: **SALESREPS.SALES**, **PROBA.SALES**!

В этом случае ошибки не будет так как применяется, точечная нотация при выборе столбцов, для запроса. При этом таблицы как объекты имеют разные имена, хотя и содержат одинаковые столбцы! Так же разные схемы БД, то есть пользователи, так же могут содержать таблицы с одинаковыми именами, но в разных схемах! Например: **SCOTT.SALESREPS.SALES** и **MILLER.SALESREPS.SALES**! Думаю, это не понятно. А теперь давайте рассмотрим некоторые типы данных, так же применительно к БД **Oracle**.

- **CHAR(n), VARCHAR2(n)** - Строки переменной длины.
- **INTEGER** - Масштабируемое целое.
- **NUMBER(n)** - Масштабируемое целое с плавающей точкой.
- **DATE** - Дата/Время
- **ROWID, ROW** - Идентификаторы записей в БД.
- **BLOB** - Большие двоичные объекты.
- **CLOB** - Большие строковые объекты.

- **BFILE** - Указатели на большие внешние объекты.

Так же в таблицах БД применяется такое понятие как "отсутствующее значение" или **NULL**. Которое по своей сути определено 3м, правилом Кодда, а именно "Правило поддержки недействительных значений" ! Вокруг этого понятия до сих пор происходит, множество дискуссий и даже предложений исключить его как таковое. Но, тем не менее, до сих пор, оно существует и мне кажется это верное решение! Вследствие применения модели реального мира. Далее мы еще вернемся к этому вопросу. Пока вот собственно все, что касается типов данных в БД **Oracle**. В следующем шаге, мы создадим учебные таблицы и двинемся дальше по пути познания **SQL**!

Шаг 8 - Пересоздаем пользователя miller

Продолжаем! В шаге ["Шаг 6 - Создание пользователя и настройка прав доступа"](#) мы создали пользователя **MILLER**, но я допустил ряд небольших недочетов, пересмотрев изложенный материал, по этому поступим так, удалим **MILLER**'а благо это не трудно, создадим его заново, а затем, наконец, зальем в схему **MILLER** учебные таблички и начнем, наконец, разбирать **SQL** по полочкам! Итак, поехали! Запускаем **SQL Plus** пользователем **SYSTEM**, (пароль **MANAGER**, строка связи **PROBA**) Видим:

```
Присоединен к:
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production
```

```
SQL>
```

Прекрасно, пишем:

```
SQL> DROP USER MILLER
2/
Пользователь удален.
```

Отлично, теперь делаем следующее, создайте на своем диске **C:** каталог **Tmp** он нам пригодиться еще не раз для того, чтобы не писать слишком длинные пути к файлам и затем введите в ответ на приглашение:

```
SQL> SPOOL C:\TMP\SPL.TXT
```

Теперь все, что мы делаем "спулится" в файл **SPL.TXT** в каталоге **C:\TMP**, если нужно можно, в него заглянуть после закрытия **SQL Plus** и что-то самое интересное сохранить на память. Да, файл переписывается с каждым новым запуском **SQL Plus**! Теперь заберите файлы из проекта.

После того как забрали файлы, которые я вам приготовил, поместите файл **CREATEMILLER.SQL** в каталог **C:\TMP** и введите следующее:

```
SQL> @C:\TMP\CREATEMILLER.SQL
Пользователь создан.
Привилегии предоставлены.
```

```
.
.
.
```

```
Привилегии предоставлены.
```

```
Фиксация обновлений завершена.
```

```
SQL>
```


Замечательно! Теперь **MILLER** создан и без ошибок!!! Можно заливать таблички в схему! Теперь, подключитесь к БД, пользователем **MILLER** написав:

```
SQL> CONNECT MILLER/KOLOBOK@PROBA
Соединено.
```

Получилось! Спросите систему кто я?

```
SQL> SHOW USER
USER имеет значение "MILLER"
```

MILLER, не врет! Посмотрите, есть ли у пользователя **MILLER** таблички запросив представление **USER_TABLES** следующим образом:

```
SQL> SELECT * FROM USER_TABLES
2 /
строки не выбраны
```

Да, действительно ничего нет! Но это не проблема, сейчас создадим семь табличек, которые понадобятся нам в дальнейшем для разбора командного языка **SQL**! Пока можете закрыть **SQL Plus** введя **EXIT**. Далее сделаем вот, что. Возьмите файлы **miller.bat** и **miller.dat** (почти каламбур) и поместите их в каталог **C:\Oracle\Ora81\BIN**, запустите файл **miller.bat** (все действия совету производить из утилиты типа **FAR**, так как удобнее наблюдать за происходящим и не пугайтесь, работе с командной строкой!!!) Если все прошло успешно, то увидите следующее:

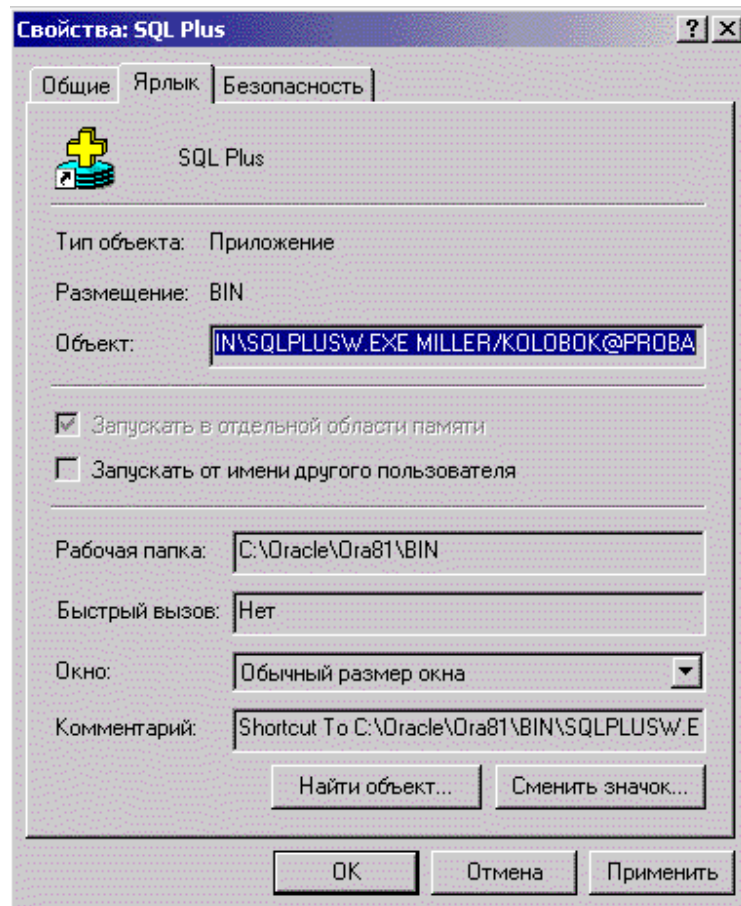
```
C:\Oracle\Ora81\BIN>Miller.bat
C:\Oracle\Ora81\BIN>set nls_lang=russian_cis.ru8pc866
C:\Oracle\Ora81\BIN>imp.EXE USERID=MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
Import: Release 8.1.5.0.0 - Production on Пнд Июн 30 11:49:55 2003
(c) Copyright 1999 Oracle Corporation. All rights reserved.

Соединен c: Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning and Java options
PL/SQL Release 8.1.5.0.0 - Production

Экспорт-файл создан EXPORT:V08.01.05 через обычный маршрут
импорт выполнен в кодировке RU8PC866 и CL8MSWIN1251 кодировке NCHAR
импортирующий сервер использует кодировку CL8MSWIN1251 (возможно перекодирование)
. . импортируемая таблица          "BOYS" импорт-но      6 строк
. . импортируемая таблица          "CUSTOMERS" импорт-но  22 строк
. . импортируемая таблица          "GIRLS" импорт-но     6 строк
. . импортируемая таблица          "OFFICES" импорт-но   5 строк
. . импортируемая таблица          "ORDERS" импорт-но    30 строк
. . импортируемая таблица          "PRODUCTS" импорт-но   25 строк
. . импортируемая таблица          "SALESREPS" импорт-но   11 строк
Импорт завершился успешно без предупреждений.
```

Вот теперь таблички залиты в схему **MILLER**. Все это было произведено с помощью утилиты экспорта в схему БД, как она работает, это тема для отдельного разговора, по этому пока не будем углубляться в содержимое этих файлов, я все это расскажу в дальнейшем и подробно! Теперь предлагаю сделать еще одну вещь, для того чтобы не вводить каждый раз при запуске **SQL Plus**, пользователя и пароли, тем более, что мы будем работать в следующих шагах, в основном с пользователем **MILLER**, выберите в стартовом меню иконку **SQL Plus** и нажмите правую клавишу

мыши и выберите пункт меню "Свойства" в строке "Объект" закладки "Ярлык" после строки "C:\Oracle\Ora81\BIN\SQLPLUSW.EXE" введите строку "MILLER/KOLOBOK@PROBA" и теперь ваш **SQL Plus** при старте будет заходить в схему **MILLER**!



Вот и все! Быстро и время тратить не нужно!

Теперь после старта с пользователем **MILLER** введите:

```
SQL> SELECT TABLE_NAME, TABLESPACE_NAME FROM USER_TABLES
2 /
```

TABLE_NAME	TABLESPACE_NAME
BOYS	USERS
CUSTOMERS	USERS
GIRLS	USERS
OFFICES	USERS
ORDERS	USERS
PRODUCTS	USERS
SALESREPS	USERS

7 строк выбрано.

Ура! Имеем семь табличек для работы и изучения командного языка **SQL**! Пока все, отдохните и осмыслите эту тему!

Шаг 9 - Оператор SELECT

Что ж, наконец, пришло время взяться за **SQL** вплотную. Если говорить прямо, то практически весь **SQL** - базируется на единственном краеугольном камне, а именно операторе **SELECT**. Давайте попробуем разобраться, что же это такое? Сам оператор **SELECT** есть не что иное, как выражение для получения результирующего набора из таблиц БД. Мы формируем запрос при помощи оператора **SELECT**, а он, в свою очередь, выполнив свою работу, возвращает результирующий набор данных. Возвращаются эти данные в виде таблицы, которая в свою очередь, может быть обработана следующим оператором **SELECT** и т.д. Пока понятно?

Давайте поближе рассмотрим сам оператор и его синтаксис. Согласно стандарту **SQL92**, все это выглядит примерно так:

```
SELECT -- ALL ----- схема , столбец ----  
      -- DISTINCT -- ---- * -----  
  
FROM   -- схема , таблица .. -----  
  
WHERE  -- условие поиска -----  
  
GROUP BY -- схема , столбец -----  
  
HAVING -- условие поиска -----  
  
ORDER BY -- спецификатор сортировки -----
```

Первое правило, само выражение **SELECT** обязательно включает, выражение **FROM**. Остальные выражения используются по мере необходимости.

Выражение **SELECT** включает в себя список столбцов возвращаемых запросом.

Выражение **FROM** включает в себя список таблиц для выполнения запроса.

Выражение **WHERE** устанавливает условие поиска, если необходимо вернуть не все строки, а только ту часть, которая описана условием, поиска.

Выражение **GROUP BY** позволяет создать итоговой запрос, разбитый на группы.

Выражение **HAVING** определяет условие возврата групп и используется только совместно с **GROUP BY**.

Выражение **ORDER BY** определяет порядок сортировки результирующего набора данных.

Вот вкратце, как устроен оператор **SELECT**. Пока еще не все понятно, но в процессе разберемся!

Для начала давайте, выполним простой запрос, который возвращает, все записи (это обуславливает оператор *****) из таблицы **OFFICES**. Наш запрос будет выглядеть так:

```
Выбрать все записи из таблицы OFFICES.  
SELECT *(все!) FROM OFFICES
```

```
SQL> SELECT * FROM OFFICES
2 /
```

OFFICE	CITY	REGION	MGR	TARGET
22	Запиндрищинск	Запад	108	300
186,042				
11	Красный Мотоцикл	Восток	106	575
692,637				
12	Чугуевск	Восток	104	800
735,044				
13	Бубурино	Восток	105	350
367,911				
21	Котрогайка	Запад	108	725
835,915				

Результат несколько не нагляден вследствие того, что столбец **CITY** не попадает в формат вывода **SQL Plus**, это легко исправить, введя предикат **COL CITY FORMAT A16**. Таким образом, в начале каждого запроса можно отформатировать каждый столбец. Немного громоздко, но полезно!!! :)

Итак снова выполним:

```
SELECT *(все!) FROM OFFICES
```

```
SQL> COL CITY FORMAT A16
SQL> SELECT * FROM OFFICES
2 /
```

OFFICE	CITY	REGION	MGR	TARGET	SALES
22	Запиндрищинск	Запад	108	300	186,042
11	Красный Мотоцикл	Восток	106	575	692,637
12	Чугуевск	Восток	104	800	735,044
13	Бубурино	Восток	105	350	367,911
21	Котрогайка	Запад	108	725	835,915

Ну вот, теперь все очень хорошо видно. Запрос вернул шесть столбцов и пять строк! Это важно!

Да, давайте еще раз посмотрим, сколько у нас таблиц в нашей схеме? Введите:

```
SELECT TABLE_NAME(выбираем одно поле!) FROM USER_TABLES(из таблицы USER_TABLES!)
```

```
SQL> SELECT TABLE_NAME FROM USER_TABLES
2 /
```

```
TABLE_NAME
```

```
-----
BOYS
CUSTOMERS
GIRLS
OFFICES
ORDERS
PRODUCTS
SALESREPS
```

7 строк выбрано.

Семь таблиц, запрос вернул, один столбец и семь строк!

Теперь попробуем выбрать еще несколько запросов пока, что не задумываясь, над их формулировками.

```
-----
SQL> SELECT CITY, TARGET, SALES
2 FROM OFFICES
3 /
```

```
CITY                TARGET    SALES
-----
Запндрищинск        300  186,042
Красный Мотоцикл    575  692,637
Чугуевск            800  735,044
Бубурино            350  367,911
Котрогайка          725  835,915
```

Вернули список офисов с их плановыми и фактическими, объемами продаж.

```
-----
SQL> SELECT CITY, TARGET, SALES
2 FROM OFFICES
3 WHERE REGION = 'Запад'
4 /
```

```
CITY                TARGET    SALES
-----
Запндрищинск        300  186,042
Котрогайка          725  835,915
```

Вернули список офисов расположенных в западном регионе с их плановыми и фактическими, объемами продаж.

```
-----
SQL> SELECT CITY, TARGET, SALES
2 FROM OFFICES
```

```
3 WHERE REGION = 'Запад'
4 AND SALES > TARGET
5 ORDER BY CITY
6 /
```

CITY	TARGET	SALES
Котрогайка	725	835,915

Вернули список офисов расположенных в западном регионе где фактические объемы продаж, превысили плановые и отсортировать их в алфавитном порядке, по названию городов. Вернулась только одна запись, а вернее три столбца и одна строка, так что можно было и не сортировать!

```
-----
SQL> SELECT AVG(TARGET), AVG(SALES)
2 FROM OFFICES
3 WHERE REGION = 'Запад'
4 /
```

AVG(TARGET)	AVG(SALES)
512,5	510,9785

Ну, на последок выберем из наших офисов, средние значения плановых и фактических объемов продаж по всем офисам расположенным в западном регионе. Вот примерно, так на первый взгляд просто строить, запросы при помощи операторов **SELECT**! Но так ли, это на самом деле, посмотрим дальше!

Шаг 10 - Выборка данных с помощью SELECT

Продолжаем путь по тернистой дорожке **SQL**! Давайте немного углубимся в оператор **SELECT**. Итак данный оператор как вы уже надеюсь поняли, является ключевым при выполнении запросов к таблицам БД. Оператор **SELECT** обязательно должен содержать, "список возвращаемых столбцов", то есть:

```
SELECT FILED1, FIELD2, FIELD3 ... FROM ...
```

Слова **FILED1**, **FIELD2**, **FIELD3** и есть не что иное как "список возвращаемых столбцов"! И набор данных, которые вернет ваш запрос **SELECT**, будет, следовать именно по порядку "списка возвращаемых столбцов"! Это необходимо, четко представлять! Так же "список возвращаемых столбцов", может содержать вычисляемые столбцы и константы.

```
SELECT FILED1, (FIELD2 - FIELD3) "CONST" ... FROM ...
```

В дальнейшем вы все это увидите!

Предложение **FROM** содержит "список спецификаторов таблиц", а именно имена таблиц, из которых производится запрос! Такие таблицы называются "исходными таблицами запроса". Так как результаты запроса берутся именно из них! Теперь давайте попробуем практически проверить все сказанное. Запускайте **SQL Plus** и введите запрос: Вывести список имен, офисов и дат приема на работу всех служащих.

```
SQL> SELECT NAME, REP_OFFICE, HIRE_DATE FROM SALESREPS
2 /
```

NAME	REP_OFFICE	HIRE_DATE
Вася Пупкин	13	12.02.88
Маша Распутина	11	12.10.89
Филип Киркоров	21	10.12.86
Света Разина	11	14.06.88
Наташа Королева	12	19.05.87
Игорь Николаев	12	20.10.86
Крис Кельми	13	13.05.90
Игорь Петров	21	12.10.89
Дима Маликов	12	01.03.87
Маша Сидорова	22	14.11.88
Максим Галкин	12	12.10.89

11 строк выбрано.

Запрос вернул три столбца и одиннадцать строк, замечательно! Я использовал таблицы из учебной БД, она несколько суховата, я только заменил в ней англоязычные слова на более "родные", названия, просто, чтобы было веселее и быстрее усваивалось! Ваша задача правильно понимать, постановку задачи запроса и его, результат! Что собственно и является конечной целью, нашего мероприятия! :-) Теперь давайте выполним еще один запрос, с такой формулировкой: Как зовут, а так же каковы плановый и фактический объем продаж, служащего с идентификатором 107?

```
SQL> SELECT NAME, QUOTA, SALES FROM SALESREPS
```

```
2 WHERE EMPL_NUM = 107
3 /
```

NAME	QUOTA	SALES
Маша Сидорова	300	186,042

Получаем таблицу, из трех столбцов и одной строки, хотя здесь и одна строка это тем не менее таблица результата запроса! Некоторые запросы вообще возвращают единственное значение, например: Каково среднее значение фактических объемов продаж по всем служащим компании?

```
SQL> SELECT AVG(SALES) FROM SALESREPS
2 WHERE EMPL_NUM = 107
3 /
```

```
AVG(SALES)
-----
186,042
```

Получаем одно значение (!), но и это тоже таблица, состоящая из одной строки и одного столбца! И, наконец, любой запрос может вернуть ноль строк!

Например: Вывести список имен и дат приема на работу, всех служащих фактический объем продаж которых превышает 50 000?

```
SQL> SELECT NAME, HIRE_DATE FROM SALESREPS
2 WHERE SALES > 50000
3 /
```

строки не выбраны

```
NAME, HIRE_DATE
-----
```

Даже в этом случае результат является таблицей, как это не кажется парадоксальным. Он содержит два (!) столбца и ноль (!!!) строк! Неожиданно не так ли? Но тем не менее утверждение считается верным!!! Идем дальше, в таблицах как вы помните есть значения **NULL**, это значит, что значение пока не определено, не ДА ни НЕТ, а так сяк! :) Давайте сделаем запрос такого типа: Вывести список служащих, с их плановыми объемами продаж и идентификаторами менеджеров.

```
SQL> SELECT NAME, MANAGER, QUOTA FROM SALESREPS
2 /
```

NAME	MANAGER	QUOTA
Вася Пупкин	104	350
Маша Распутина	106	300
Филип Киркоров	108	350
Света Разина	NULL	275
Наташа Королева	106	200
Игорь Николаев	104	300

Крис Кельми	101	NULL
Игорь Петров	106	350
Дима Маликов	104	275
Маша Сидорова	108	300
Максим Галкин	108	400

11 строк выбрано.

Что же получилось у Светы Разиной, нет менеджера! (Да и зачем он ей собственно!!!) А у Криса Кельми нет планового объема продаж (Все отпелся голубок!!!) Поля содержат значения **NULL**, но в запрос, тем не менее, они попадают! Что и требовалось доказать! Вот пока все, но с оператором **SELECT**, еще нужно разбираться! :)

Шаг 11 - Выборка данных с помощью SELECT, часть 2

Идем дальше! Давайте, выполним вот такой запрос:

Вывести для каждого из офисов список городов, регионов и объемов продаж.

```
SQL> SELECT CITY, REGION, SALES
2 FROM OFFICES
3 /
```

CITY	REGION	SALES
Запіндрищінск	Запад	186,042
Красный Мотоцикл	Восток	692,637
Чугуевск	Восток	735,044
Бубурино	Восток	367,911
Котрогайка	Запад	835,915

Нечто подобное мы уже проделывали, но я повторяюсь, вот почему. Чтобы вы поняли, что это "простой запрос", возвращающий все записи из трех, столбцов таблицы **OFFICES**. Вот на этом все, надеюсь объяснять больше ничего не нужно! Теперь давайте вернемся к теме "вычисляемые столбцы", как я уже говорил, любой из возвращаемых, столбцов, оператора **SELECT**, может быть "вычисляемым", что это значит. А то, что это не столбец из запрашиваемой таблицы, а некое математическое, выражение представленное в результирующем, наборе запроса как самостоятельный столбец! То есть фактически, столбца с такими значениями и именем в запрашиваемой таблице нет(!), он просто-напросто получается в результате математического, вычисления! Понятно? Пробуем: Выдать для каждого офиса список, городов, регионов и сумм, на которые перевыполнен/недовыполнен план по продажам.

```
SQL> SELECT CITY, REGION, (SALES-TARGET)
2 FROM OFFICES
3 /
```

CITY	REGION	(SALES-TARGET)
Запіндрищінск	Запад	-113,958
Красный Мотоцикл	Восток	117,637
Чугуевск	Восток	-64,956
Бубурино	Восток	17,911
Котрогайка	Запад	110,915

Столбец с именем (**SALES-TARGET**) и есть вычисляемый! А, можно вот так, используя правила и возможности сервера **Oracle**!

```
SQL> COL CITY FORMAT A18
SQL> SELECT CITY, REGION, '$ '||(SALES-TARGET) AS STOIM
2 FROM OFFICES
3 /
```

CITY	REGION	STOIM

Запиндришинск	Запад	\$ -113,958
Красный Мотоцикл	Восток	\$ 117,637
Чугуевск	Восток	\$ -64,956
Бубурино	Восток	\$ 17,911
Котрогайка	Запад	\$ 110,915

Теперь нагляднее, не правда ли? Вместо чего-то мало понятного (**SALES-TARGET**) стоит **STOIM** просто имя столбца! А перед, цифрами появился, наш любимый знак Российской (тьфу ты Американской!) валюты, доллара! отсюда можно сделать четкий вывод, что Запиндришинск и Чугуевск, пролетают с премиями! :) А, теперь немного забегаю вперед, поясню, что это за выражение '\$ '||(SALES-TARGET) AS STOIM. '\$ ' - строковый литерал, включающий строку, из двух символов \$ и пробел, ||- знак конкатенации строковых литералов (а, в языке С и С++ выражение ИЛИ! Привет программистам! Не путайтесь!), (**SALES-TARGET**) - собственно вычисляемое выражение, ну и предикат **AS STOIM** - буквально, назвать столбец как **STOIM**! Вот собственно небольшой экскурс, в **PL/SQL**! Но им мы будем заниматься, позже и более подробно!!! Таким образом, мы получили небольшой отчет для выявления недобросовестных, работников!

Идем дальше, попробуем вот еще что: Показать общую стоимость по каждому товару.

```
SQL> SELECT MFR_ID, PRODUCT_ID, DESCRIPTION, (QTY_ON_HAND * PRICE)
2 FROM PRODUCTS
3 /
```

MFR PRODU DESCRIPTION	(QTY_ON_HAND*PRICE)
-----------------------	---------------------

REI 2A45C Бочка металлическая	16590
ACI 4100Y Коробка картонная	68,75
QSA XK47 Труба алюминиевая	13490
BIC 41672 Тарелка фарфоровая	0
IMM 779C Профиль специальный	16875
ACI 41003 Рейка деревянная	22149
ACI 41004 Рейка пластмассовая	16263
BIC 41003 Стекломаст рулоны	1956
IMM 887P Рубероид рулоны	6000
QSA XK48 Гвоздь длинный	27202
REI 2A44L Доска профильная	54000
FEA 112 Стол офисный	17020
IMM 887H Тумбочка прикроватная	12042
BIC 41089 Сапоги юфтевые	17550
ACI 41001 Лампа настольная	15235
IMM 775C Осветитель ртутный	7125
ACI 4100Z Монитор LG	70000
QSA XK48A Подушка ватная	6549
ACI 41002 Носки черные	12692
REI 2A44R Телевизор SAMSUNG	54000
IMM 773C Наушники SONY	27300

MFR PRODU DESCRIPTION	(QTY_ON_HAND*PRICE)
-----------------------	---------------------

ACI 4100X Карандаш простой	925
FEA 114 Электродвигатель	3645
IMM 887X Нож специальный	15200
REI 2A44G Бочка пластмассовая	4900

25 строк выбрано.

Или вот так:

```
SQL> COL STOIM FORMAT A12
SQL> SELECT MFR_ID, PRODUCT_ID, DESCRIPTION, '$' || (QTY_ON_HAND * PRICE) AS STOIM
2 FROM PRODUCTS
3 /
```

MFR PRODU DESCRIPTION	STOIM
REI 2A45C Бочка металлическая	\$ 16590
ACI 4100Y Коробка картонная	\$ 68,75
QSA XK47 Труба алюминиевая	\$ 13490
BIC 41672 Тарелка фарфоровая	\$ 0
IMM 779C Профиль специальный	\$ 16875
ACI 41003 Рейка деревянная	\$ 22149
ACI 41004 Рейка пластмассовая	\$ 16263
BIC 41003 Стекломаст рулоны	\$ 1956
IMM 887P Рубероид рулоны	\$ 6000
QSA XK48 Гвоздь длинный	\$ 27202
REI 2A44L Доска профильная	\$ 54000
FEA 112 Стол офисный	\$ 17020
IMM 887H Тумбочка прикроватная	\$ 12042
BIC 41089 Сапоги юфтевые	\$ 17550
ACI 41001 Лампа настольная	\$ 15235
IMM 775C Осветитель ртутный	\$ 7125
ACI 4100Z Монитор LG	\$ 70000
QSA XK48A Подушка ватная	\$ 6549
ACI 41002 Носки черные	\$ 12692
REI 2A44R Телевизор SAMSUNG	\$ 54000
IMM 773C Наушники SONY	\$ 27300
MFR PRODU DESCRIPTION	STOIM
ACI 4100X Карандаш простой	\$ 925
FEA 114 Электродвигатель	\$ 3645
IMM 887X Нож специальный	\$ 15200
REI 2A44G Бочка пластмассовая	\$ 4900

25 строк выбрано.

Нагляднее, верно! Да и знак \$, глаз радует!

А, что если увеличить плановый объем продаж, для каждого служащего, на 3% от его фактического, объема продаж! Да, без проблем! Внимательно посмотрите вычисляемое выражение, так работает **Oracle**!

```
SQL> SELECT NAME, QUOTA, (QUOTA + ((SALES/100)*3))
2 FROM SALESREPS
3 /
```

NAME	QUOTA (QUOTA+((SALES/100)*3))	
Вася Пупкин	350	361,03733
Маша Распутина	300	311,78175
Филип Киркоров	350	364,2215
Света Разина	275	283,99736
Наташа Королева	200	204,27782
Игорь Николаев	300	309,17019
Крис Кельми		
Игорь Петров	350	360,85595
Дима Маликов	275	283,60325
Маша Сидорова	300	305,58126
Максим Галкин	400	411,58126

11 строк выбрано.

Получилось! Главное что бы Максим Галкин не надорвался! А, чего ему сделается, в армию все равно не пойдет! Заработает! Ну, да бог ему судья! :) Остановимся, еще вот на таком аспекте, как столбец, константа. Что, он из себя представляет? Иногда, требуется, что бы один из столбцов запрос возвращал, одно и то же значение не зависимо, от условий поиска!

Например: Вывести список объемов продаж, для каждого города.

```
SQL> SELECT CITY, 'Has sales of', SALES
2 FROM OFFICES
3 /
```

CITY	'HASALESOF'	SALES
Запиндрищинск	Has sales of	186,042
Красный Мотоцикл	Has sales of	692,637
Чугуевск	Has sales of	735,044
Бубурино	Has sales of	367,911
Котрогайка	Has sales of	835,915

'Has sales of' и есть столбец константа. Просто и со вкусом! Пока надо пойти попить кофе и покурить, хотя это вредно!!! И курить тоже!!! :)

Шаг 12 - Выборка данных с помощью SELECT, часть 3

Осмыслили! Идем дальше! Вернемся еще раз к типу выбора столбцов в операторе, **SELECT**. Иногда, когда, например вы сталкиваетесь, с незнакомой БД, вам необходимо быстро рассмотреть таблицы. В таком случае вы примените, что-то в этом духе:

```
SQL> COL CITY FORMAT A18
SQL> SELECT * FROM OFFICES
2/
```

Получаем следующее:

OFFICE CITY	REGION	MGR	TARGET	SALES
22 Запиндрищинск	Запад	108	300	186,042
11 Красный Мотоцикл	Восток	106	575	692,637
12 Чугуевск	Восток	104	800	735,044
13 Бубурино	Восток	105	350	367,911
21 Котрогайка	Запад	108	725	835,915

Здесь '*' значит "выдать все строки"! Запросы такого типа применяются не часто, в основном как, я говорил выше, либо при получении отчетов. В некоторых БД, даже допускается такое:

```
SQL> SELECT *, (SALES-TARGET) FROM OFFICES
2/
```

Но, у меня БД ответила так:

```
SELECT *, (SALES-TARGET) FROM OFFICES
*
```

ошибка в строке 1:

ORA-00923: ключевое слово FROM не найдено там, где оно ожидалось

Чего собственно и следовало ожидать! Да и сам запрос как-то малопонятен. Еще могу добавить, что оператор '*' так же применяется и в других случаях, например, по стандарту, по моему **SQL91**, как шаблон условия **LIKE**, но в **Oracle** такого, я не наблюдал! Так что, функция только одна, выдать все записи из заданной таблицы! Так же как правило, при выборе записей, часто используют уникальный ключ, для того, чтобы каждая запись была уникальна! Что это такое, расскажу, когда коснемся темы минимизации таблиц. Но иногда, возникает такая ситуация, что при выборе данных, появляются повторяющиеся значения.

Например: Вывести список идентификаторов менеджеров офисов.

```
SQL> SELECT MGR FROM OFFICES
2 /
```

```
MGR
-----
108
106
```


104
105
108

Запись с номером 108 повторилась дважды! Так как Игорь Петров, из таблицы **SALESREPS** является менеджером по нашей легенде, сразу двух офисов. По этому запись и повторяется. Чтобы этого не происходило, нужно использовать оператор **DISTINCT**. Например, вот так:

```
SQL> SELECT DISTINCT MGR FROM OFFICES
2/
```

```
      MGR
-----
      104
      105
      106
      108
```

Теперь результат, правильный, все ясно и понятно. Сразу предупреждаю, что **DISTINCT** является не производительным, оператором, по этому злоупотреблять им, особенно на больших, таблицах не рекомендую! Пока с этим все! Дальше пойдет, довольно, объемный для изложения материал, так что соберитесь, с мыслями!

Шаг 13 - Условие WHERE оператора SELECT

А вот теперь, самое интересное! Предложение **WHERE**, наверное, если сказать, честно, это самое "навороченное" предложение, языка **SQL**! Собственно, это некий фундаментальный кирпичик, для работы, с таблицами и выполнением, разнообразных запросов! Мы с вами рассмотрим основные из них, а что я не успею разъяснить, сможете разобраться и сами, с помощью, моих "записок сумасшедшего" :)))) Итак для начала выполним пару тройку запросов, к нашей учебной БД, используя выражение **WHERE**: Перечислить, офисы, в которых фактический объем продаж, превысил плановые.

```
SQL> SELECT CITY, SALES, TARGET FROM OFFICES
2 WHERE SALES > TARGET
3 /
```

CITY	SALES	TARGET
Красный Мотоцикл	692,637	575
Бубурино	367,911	350
Котрогайка	835,915	725

Здесь **WHERE SALES > TARGET**, значит, если **SALES** больше **TARGET**! Получили три столбца и три строки.

Вывести, имя, объем фактических и плановых продаж для служащего с идентификатором 105

```
SQL> SELECT SALES, NAME, QUOTA FROM SALESREPS
2 WHERE EMPL_NUM = 105
3 /
```

SALES	NAME	QUOTA
367,911	Вася Пупкин	350

Здесь **WHERE EMPL_NUM = 105**, означает, **EMPL_NUM** равно **105**! Поучили три столбца и одну строку.

Вывести список всех служащих, менеджером которых является Наташа Королева. (вот тут то и станет ясно, ху из ху) :)

```
SQL> 1 SELECT NAME, SALES FROM SALESREPS
2 WHERE MANAGER = 104
3 /
```

NAME	SALES
Вася Пупкин	367,911
Игорь Николаев	305,673
Дима Маликов	286,775

Здесь **WHERE MANAGER = 104** означает то, что и ранее, но получаем два столбца и три строки. Если говорить с точки зрения логики, то происходит следующее. Если условие поиска ИСТИНА(**TRUE**), то строка включается в результирующий набор, если условие поиска ЛОЖНО(**FALSE**), то строка отбрасывается из результирующего набора, если условие поиска **NULL**, то строка так же, отбрасывается из результирующего набора! Это важно, не забывайте, что **NULL**, срабатывает, в этом случае, так же как **FALSE**! По своей сути **WHERE**, работает как фильтр, отсеивая ненужные записи и выдавая нужные! Вот собственно такой смысл, этого оператора!

Основных условий поиска, иначе называемых "предикаты", пять. Рассмотрим их последовательно:

1. Сравнение, то есть значение одного условия, сравнивается со значением, другого условия. Как, например, в первом запросе.
2. Проверка на принадлежность к диапазону значений. Например, проверяется, попадает ли указанное значение, в определенный диапазон или нет.
3. Проверка на членство во множестве. Например, проверяется, совпадает ли значение выражения, с одним из значений из заданного множества.
4. Проверка на соответствие шаблону. Проверяется, соответствует ли строковое значение, содержащееся в столбце, определенному шаблону.
5. Проверка на равенство значению **NULL**. Комментарии здесь я думаю излишни!

Начнем со сравнений. Применяемые выражения могут быть такими и в такой синтаксисе:

```
        больше
----- ВЫРАЖЕНИЕ1 ----- > ----- ВЫРАЖЕНИЕ2 -----
        меньше
        <
        равно
        =
        не равно
        !=
        <>
        меньше либо равно
        <=
        больше либо равно
        =>
```

Операции сравнения могут содержать простые операции, например сравнение поля и константы: Найти имена всех служащих принятых на работу до 1988 г.

```
SQL> SELECT NAME FROM SALESREPS
2  WHERE HIRE_DATE < TO_DATE('01.06.1988','DD/MM/YYYY')
3  /
```

NAME

```
-----
Вася Пупки
Филип Киркоров
Наташа Королева
Игорь Николаев
Игорь Петров
```

Дима Маликов

Функция **TO_DATE('01.06.1988','DD/MM/YYYY')** - встроенная функция **PL/SQL Oracle** для работы с датами. Вот например два одинаковых запроса, в которых видно, что операторы **<>**, **!=** действуют одинаково и какой выбирать дело вкуса.

Вывести список офисов, менеджером которых не является служащий под номером 108.

```
SQL> SELECT CITY, MGR FROM OFFICES
2 WHERE MGR <> 108
3 /
```

CITY	MGR
Красный Мотоцикл	106
Чугуевск	104
Бубурино	105

```
SQL> SELECT CITY, MGR FROM OFFICES
2 WHERE MGR != 108
3 /
```

CITY	MGR
Красный Мотоцикл	106
Чугуевск	104
Бубурино	105

Как видим результат одинаковый! Либо содержать арифметические выражения: Вывести список офисов, фактические объемы продаж в которых, составили менее 80 процентов от плановых.

```
SQL> SELECT CITY, SALES, TARGET FROM OFFICES
2 WHERE SALES < (0.8 * TARGET)
3 /
```

CITY	SALES	TARGET
Запиндрицинск	186,042	300

Следует заметить что, чаще всего используются запросы сравнения с константой, по первичному ключу поиска, такой например как телефон абонента городской АТС, ведь двух одинаковых номеров не бывает! По этому возвращается одно значение, и если таблица хорошо оптимизирована то, как правило, очень быстро. Вот собственно такие выводы!

Шаг 14 - Оператор BETWEEN и IN

Итак, идем дальше, рассмотрим выражение **BETWEEN**, по своей сути это выражение проверки на принадлежность к диапазону значений. Синтаксис выражения строится примерно так:

--- проверяемое выражение ----- BETWEEN ----- нижнее выражение AND верхнее выражение
- NOT -

Выражение **NOT** обуславливает обратное инвертирование условия, то есть "не принадлежит".

Давайте попробуем применить, на практике, запускайте **SQL Plus** и вводите: Найти все заказы, сделанные в последнем квартале 1989г.

```
SQL> SELECT ORDER_NUM, ORDER_DATE, MFR, PRODUCT, AMOUNT
2 FROM ORDERS
3 WHERE ORDER_DATE BETWEEN TO_DATE('01.11.1989','DD/MM/YYYY') AND
TO_DATE('31.12.1989','DD/MM/YYYY')
4 /
```

ORDER_NUM	ORDER_DA	MFR	PRODU	AMOUNT
112961	17.12.89	REI	2A44L	31,5
112963	17.12.89	ACI	41004	3,276
112983	27.12.89	ACI	41004	702
112922	04.11.89	ACI	41002	760
112987	31.12.89	ACI	4100Y	27,5

Так работает с датами, выражение **TO_DATE** и есть преобразование строк!

Идем дальше: Найти заказы, стоимость которых попадают в различные диапазоны.

```
SQL> SELECT ORDER_NUM, AMOUNT
2 FROM ORDERS
3 WHERE AMOUNT BETWEEN 20.000 AND 29.999
4 /
```

ORDER_NUM	AMOUNT
113036	22,5
112987	27,5
113042	22,5

```
SQL> SELECT ORDER_NUM, AMOUNT
2 FROM ORDERS
3 WHERE AMOUNT BETWEEN 30.000 AND 39.999
4 /
```

ORDER_NUM	AMOUNT
112961	31,5
113069	31,35

```
SQL> SELECT ORDER_NUM, AMOUNT
2 FROM ORDERS
3 WHERE AMOUNT BETWEEN 1.000 AND 9.999
4 /
```

```
ORDER_NUM  AMOUNT
-----
```

```
113012    3,745
112989    1,458
113051     1,42
112968    3,978
112963    3,276
113058    1,478
113024     7,1
113062     2,43
113027    4,104
113007    2,925
112975     2,1
113048     3,75
112993    1,896
113065     2,13
113003    5,625
```

15 строк выбрано.

Так проверяем простое соответствие числовому диапазону.

Идем дальше. Можно так же с помощью выражения **NOT** проверить на вхождение в данный диапазон значений, например: Вывести список служащих, фактические объемы продаж которых не попадает в диапазон, от 80 до 120 процентов плана.

```
SQL> SELECT NAME, SALES, QUOTA
2 FROM SALESREPS
3 WHERE SALES NOT BETWEEN (0.8 * QUOTA) AND (1.2 * QUOTA)
4 /
```

```
NAME                SALES  QUOTA
-----
Маша Распутина      392,725  300
Филип Киркоров     474,05   350
Наташа Королева    142,594  200
Маша Сидорова      186,042  300
```

Заметили что, напоминает **BETWEEN**, да вот такое выражение:

```
.... (!(SALES >= (0.8 * QUOTA)) AND (SALES <= (1.2 * QUOTA)))
```

Хотя **BETWEEN** конечно нагляднее и понятнее, как мне кажется!

Теперь попробуем разобраться с выражением **IN**. Выражение **IN** это проверка на принадлежность множеству значений или, иначе говоря, членство в множестве, (фу слова какие не приглядные :))) Синтаксис команды таков:

```
--- проверяемое выражение ----- IN ----- (-- const -----)
      - NOT -                -- , -----
```

Лучше всего, конечно, пробовать ручками по этому сделаем несколько примеров с нашими учебными табличками: Вывести список служащих, которые работают в Запиндрищенске, Красном-мотоцикле или Бубурино.

```
SQL> SELECT NAME, QUOTA, SALES
2 FROM SALESREPS
3 WHERE REP_OFFICE IN (11, 13, 22)
4 /
```

NAME	QUOTA	SALES
Вася Пупкин	350	367,911
Маша Распутина	300	392,725
Света Разина	275	299,912
Маша Сидорова	300	186,042

Вот такие данные получаем.

Найти все заказы, сделанные в разные дни июня месяца 1990 года.

```
SQL> SELECT ORDER_NUM, ORDER_DATE, AMOUNT
2 FROM ORDERS
3 WHERE ORDER_DATE IN (TO_DATE('14.06.1990','DD/MM/YYYY'),
4 TO_DATE('08.06.1990','DD/MM/YYYY'),
5 TO_DATE('29.06.1990','DD/MM/YYYY'), TO_DATE('04.06.1990','DD/MM/YYYY'))
5 /
```

ORDER_NUM	ORDER_DA	AMOUNT
113013	14.06.90	652
112997	08.06.90	652
113007	08.06.90	2,925
113034	29.06.90	632

Вот так работает с датами.

Найти все заказы полученные четырьмя конкретными служащими.

```
SQL> SELECT ORDER_NUM, REP, AMOUNT
2 FROM ORDERS
3 WHERE REP IN (107, 109, 101, 103)
4 /
```

ORDER_NUM	REP	AMOUNT
-----------	-----	--------

112968	101	3,978
113058	109	1,478
112997	107	652
113062	107	2,43
113069	107	31,35
112975	103	2,1
113055	101	150
113003	109	5,625
113057	103	600,34
113042	101	22,5

10 строк выбрано.

Вот так работает числовой диапазон. С помощью **NOT IN** можно проверить на "не принадлежность" диапазону. В своей сути **IN** это примерно эквивалентно:**(REP = 107) OR (REP = 109) OR (REP = 101) OR (REP = 103)** хотя, конечно же**REP IN (107, 109, 101, 103)** нагляднее и читается гораздо лучше. Так же следует избегать чего-то вроде **NAME IN ('Филип Киркоров')** вследствие того, что выражение **NAME = 'Филип Киркоров'** гораздо проще и понятнее, по этому не нужно терять чувство реальности! Вот пока все, в следующий раз продолжим!

Шаг 15 - Оператор LIKE

Сейчас пожалуй настало время самого, на мой взгляд, употребляемого выражения в операциях поиска данных в таблицах - **LIKE**! На первый взгляд ничего особо примечательного в нем нет! Но это только на первый взгляд ... :) А вот если присмотреться по внимательнее, то можно заметить кое что интересное. Итак, синтаксис по стандарту **SQL92**:

--- ИМЯ СТОЛБЦА ----- LIKE (шаблон) -----
NOT ESCAPE (имя пропуска)

Вот так казалось бы все просто, однако это не так! Давайте выполним простой запрос: Показать лимит кредита для компании "Апельсин":

```
SQL> 1 SELECT COMPANY, CREDIT_LIMIT
      2 FROM CUSTOMERS
      3 WHERE COMPANY = 'Апельсин'
      4 /
```

COMPANY	CREDIT_LIMIT
Апельсин	50,834

Чего проще! А вот если оператор при вводе спутал третью букву в слове, "Апельсин" тогда при запуске запроса вы ничего не получите! Либо вы просто забыли как там называлась эта компания не то "Злыдень корпорейтед", не то "Студень лимитид", да это собственно и не важно, скажем вы просто помните, что там звучало слово скажем "Клоун"! Теперь следует применить оператор **LIKE** с шаблоном **'%'** и все встанет на свои места! Например вот так:

```
SQL> SELECT COMPANY, CREDIT_LIMIT
      2 FROM CUSTOMERS
      3 WHERE COMPANY LIKE '%н'
      4 /
```

COMPANY	CREDIT_LIMIT
Смешной клоун	35,645
Апельсин	50,834

В данном случае говориться, а именно - **LIKE '%н'** покажи мне все записи, которые оканчиваются на букву **'н'**, то есть, если шаблон **'%'** стоит первый, а его заканчивает некое скажем - **'бесы'**, то при выполнении запроса увидим:

```
SQL> SELECT COMPANY, CREDIT_LIMIT
      2 FROM CUSTOMERS
      3 WHERE COMPANY LIKE '%бесы'
      4 /
```

COMPANY	CREDIT_LIMIT
Безбашенные балбесы	20,765
Просто Балбесы	60,653

А это именно и означает что, начальные буквы выражения нам "по барабану", но оканчиваться оно должно так как мы и просим! Что собственно и требовалось доказать! Если написать 'C%', догадались? Естественно получим все записи начинающиеся на букву 'C', а если написать, 'Cy%', то можно найти все, что касается "КБ Сухого", но это если проломиться в БД самого КБ! Держайте!!! Ни кто не запретит и такое выражение '%шар%', которое гласит начало и окончание нам "по барабану", а вот словосочетание 'шар', нас особенно интересует!

Сразу хочу предупредить, что если ввести, что-то типа **WHERE COMPANY LIKE '%'**, можно нарваться на неприятность, а именно: **WHERE COMPANY LIKE '%'** эквивалентно, если бы вы **WHERE** оператор вообще не писали, а так и нужно было бы поступить, если бы вы были в здравом уме и действительно хотели получить все содержимое таблицы! Да именно, такое выражение вернет вам все записи в таблице! Хорошо если она не большая, а если в ней миллион, другой строк, то придется пристрелить **SQLPlus**, все равно не дождетесь результата, да и зачем это нужно!

Еще иногда вместо шаблона '%' применяется знак '*', например в **MS SQL**, ну и кто может быть знаком с **DOS** и не раз применял что-то вроде **c:\>dir *.exe!** :)))

А что если действительно спутан только один символ? Как я говорил в самом начале, тогда применяем шаблон '_' - это означает, что вы не уверены, например в правильности ввода какого либо слова! Например:

```
SQL> SELECT COMPANY, CREDIT_LIMIT
2 FROM CUSTOMERS
3 WHERE COMPANY LIKE 'Ап_льсин'
4 /
```

COMPANY	CREDIT_LIMIT
Апельсин	50,834

Какой бы не была пропущенная буква вы обязательно что-нибудь найдете!!! Пробуйте!!!

А вот еще одна ситуация, которая может встретиться, не так часто, но все же! Что, если имя компании начинается на '%%%' . Что тогда? А все очень просто - применяем "убегающие символы"! А именно последовательности пропуска, которые состоят из выражения **ESCAPE** и собственно строки или символа пропуска, в качестве которого вы можете применить например '\$', тогда символ после '\$' будет считаться литералом, а далее можно снова использовать подстановочный символ вот так:

```
WHERE PRODUCT LIKE 'A$%BS%' ESCAPE '$'
```

Еще раз, символ идущий за '\$', считается литералом, а второй символ '%', считается подстановочным то есть "по барабану"! Понятно! Но вот в умных, книгах рекомендуется таких выражений и сложностей избегать! А, а вообще это все мне например напоминает, работу с регулярными выражениями, которые горааааздоо сложнее! Но, интересно! Вот такие приключения! :) На последок, поэкспериментируйте с оператором **LIKE** и сами сможете убедиться, на сколько он гибок и удобен!

Шаг 16 - Еще раз и подробно про NULL

Вернемся еще раз к значениям, типа **NULL** (в [шаге 13](#) мы с ним уже сталкивались) и разберемся с ними немного подробнее. Трехзначная логика в таблицах БД, как известно до сих пор вызывает, не мало споров, но тем не менее она принята на вооружение, и никто от нее пока не отказывался! Действительно, ведь по своей сути БД, является, моделью реального мира и, например, если в нашей таблице менеджеров компании, одному из них еще по какой либо причине не определена квота продаж, что необходимо внести в поле таблицы, ну уж, наверное, никак не "0" или еще какое-либо вещественное значение. Вещественное значит определенное, но мы то с вами еще не определились! Так вот, для значений неопределенности, как раз и подходит **NULL**! Еще раз хочу обратить внимание, вокруг реальный мир и не нужно терять чувство реальности! :) Надеюсь, теперь все стало ясно? В дальнейшем, мы еще не однократно, будем возвращаться к значениям типа **NULL** и надеюсь, что вы поймете его суть и не будете допускать, досадных, ошибок при его использовании! Итак, давайте рассмотрим какие правила для оператора **SELECT**, применяются при работе со значением **NULL**. Возьмем конкретный пример:

Найти служащего, который еще не закреплен за офисом:

```
SQL> SELECT NAME FROM SALESREPS
2 WHERE REP_OFFICE = NULL
3 /
```

строки не выбраны

Хммм ... странно, правда? Условие я записал, все вроде бы верно. Но это только на первый взгляд. Когда **SQL**, встречает строку:

```
REP_OFFICE = NULL
```

Он выполняет следующую проверку значения:

```
NULL = NULL
```

Такой тип проверки вернет, как вы думаете, что? Конечно же, снова **NULL!!! :)))** А для оператора проверки значения все, что не является **TRUE**, в результирующий набор не входит!!! Хотя на самом деле строки с таким условием существуют!!! Как же быть в этом случае? Да, просто применить верный оператор для проверки на значение **NULL**! Вот и все! А выглядит он так:

```
----- имя столбца IS ----- NULL -----
NOT
```

Применяем: Найти служащего, который еще не закреплен за офисом.

```
SQL> SELECT NAME FROM SALESREPS
2 WHERE REP_OFFICE IS NULL
3 /
```

```
NAME
-----
```

Крис Кельми

Вот теперь все верно! И не нужно забывать, делать это именно так! Ну, естественно можно применять условие **NOT**: Вывести всех кто закреплен за офисами.

```
SQL> SELECT NAME FROM SALESREPS  
2  WHERE REP_OFFICE IS NOT NULL  
3  /
```

NAME

Вася Пупкин
Маша Распутина
Филип Киркоров
Света Разина
Наташа Королева
Игорь Николаев
Игорь Петров
Дима Маликов
Маша Сидорова
Максим Галкин

10 строк выбрано.

Вот теперь есть все кроме Криса Кельми, что ж не повезло парню, бывает! Вот так работают с **NULL**, значениями!

Надеюсь, теперь все встало на свои места, но это еще далеко не все, что касается **NULL**, мы с ним еще не раз встретимся! Что собственно говоря неизбежно! В следующий раз мы займемся составными условиями поиска!

Шаг 17 - Составные операторы в условии WHERE

Ранее мы рассматривали запросы из таблиц с применением простых условий поиска, но ограничиваться этим мы конечно же не станем, а рассмотрим так называемые "составные условия" поиска. То есть в операторе **WHERE** возможно применение нескольких условий поиска связанных между собой операторами **OR**, **AND**, **NOT**. Синтаксис этих операторов следующий:

```
Синтаксис операторов NOT, OR, AND.
(----- WHERE ----- УСЛОВИЕ -----)
      (--- NOT ---)
      (----- AND -----)
      (----- OR -----)
```

Рассмотрим несколько запросов созданных с помощью этих операторов.

Например: Найти служащих, у которых фактический объем продаж меньше планового или меньше \$300.00.

```
SQL> SELECT NAME, QUOTA, SALES
2 FROM SALESREPS
3 WHERE SALES < QUOTA OR SALES < 300.0
4 /
```

NAME	QUOTA	SALES
Света Разина	275	299,912
Наташа Королева	200	142,594
Крис Кельми	(NULL)	75,985
Дима Маликов	275	286,775
Маша Сидорова	300	186,042
Максим Галкин	400	386,042

6 строк выбрано.

Получаем три столбца и шесть строк, которые возвращает оператор **SELECT**. **OR** дословно звучит как "ИЛИ"! То есть ИЛИ слева ИЛИ справа, ИЛИ и то и другое! Если не понятно, не углубляйтесь, ниже я все разьясню.

Найти служащих, у которых фактический объем продаж меньше планового и меньше \$300.00

```
SQL> SELECT NAME, QUOTA, SALES
2 FROM SALESREPS
3 WHERE SALES < QUOTA AND SALES < 300.0
4 /
```

NAME	QUOTA	SALES
Наташа Королева	200	142,594
Маша Сидорова	300	186,042

2 строки выбрано.

А вот здесь получаем три столбца и две строки которые вернул оператор **SELECT. AND** дословно звучит как "И"!

Найти служащих, у которых фактический объем продаж меньше планового, но больше \$150.00

```
SQL> SELECT NAME, QUOTA, SALES
2 FROM SALESREPS
3 WHERE (SALES < QUOTA) AND (NOT SALES > 150.000)
4 /
```

NAME	QUOTA	SALES
Наташа Королева	200	142,594

1 строка выбрано.

Здесь еще пример, с той лишь разницей что, теперь, используем оператор **NOT**. Дословно звучит как "НЕТ" или "НЕ" или "ТОЖЕ САМОЕ ТОЛЬКО НАОБОРОТ"!

С применением этих операторов можно строить, достаточно сложные условия поиска!

Например, вот такого вида: найти всех служащих которые, 1) Работают в "Запиндришинске", "Красном мотоцикле" и "Чугуевске", или 2) не имеют менеджера и были приняты на работу после мая 1988 года или 3) превысили плановый объем продаж, но не достигли \$60.000.

Кому может понадобиться такое условие поиска, я себе смутно представляю, если только спецслужбам, но это просто пример! Кстати умение четко формулировать, суть искомой информации, поможет понять, как строить условие поиска!

```
SQL> SELECT NAME
2 FROM SALESREPS
3 WHERE (REP_OFFICE IN (22,11,12))
4 OR (MANAGER IS NULL AND HIRE_DATE >= TO_DATE('01.05.88','DD/MM/YYYY'))
5 OR (SALES > QUOTA AND NOT SALES > 60.0)
6 /
```

NAME
Маша Распутина
Света Разина
Наташа Королева
Игорь Николаев
Дима Маликов
Маша Сидорова
Максим Галкин

7 строк выбрано.

Получаем один столбец и семь строк.

А вот теперь я думаю стало ясно, что такое **AND**, **OR**, **NOT**! По сути - это стандартные логические операции, которые подчиняются всем законам Булевой алгебры. Надеюсь, с ней вы знакомы и ничего нового для вас нет. Далее посмотрите на таблички, где так же фигурирует оператор **NULL** (я же предупреждал без него никуда!) и показано, что будет возвращать выражение, если слева и справа будут стоять, **TRUE**, **FALSE**, **NULL**.

Алгебра выражения **AND**.

Значения	Результат
-----	-----
TRUE AND TRUE	-> TRUE
FALSE AND TRUE	-> FALSE
TRUE AND FALSE	-> FALSE
FALSE AND FALSE	-> FALSE
NULL AND TRUE	-> NULL
TRUE AND NULL	-> NULL
FALSE AND NULL	-> FALSE
NULL AND FALSE	-> FALSE
NULL AND NULL	-> NULL

Алгебра выражения **OR**.

Значения	Результат
-----	-----
TRUE OR TRUE	-> TRUE
FALSE OR TRUE	-> TRUE
TRUE OR FALSE	-> TRUE
FALSE OR FALSE	-> FALSE
NULL OR TRUE	-> TRUE
TRUE OR NULL	-> TRUE
FALSE OR NULL	-> NULL
NULL OR FALSE	-> NULL
NULL OR NULL	-> NULL

Алгебра выражения **NOT**.

Значения	Результат
-----	-----
NOT TRUE	-> FALSE
NOT FALSE	-> TRUE
NOT NULL	-> NULL

Вот теперь, я надеюсь все неясности с составными операторами, должны остаться в прошлом. Да, чуть не забыл, все операторы составного поиска имеют каждый свой приоритет. Наивысший приоритет у **NOT**, за ним идет **AND** и замыкает **OR**! Вот именно для этого в запросе "для спецслужб" стоят "(" ")", с их помощью, можно строить выражение, так как вам нужно!

И еще кое что, сам SQL92 работает с знакомым нам оператором **IS**, с его помощью, так же можно проверить, будет ли выражение истинным, или ложным или не определенным. Его синтаксис, следующий:

Синтаксис оператора IS

```
----- Сравнение ----- IS (----- TRUE -----)
                          (----- FALSE -----)
--- Логическое выражение --- (----- UNKNOWN -----)
```

Например можно записать, следующее: **((SALES - QUOTA) > 100.000) IS UNKNOWN** такое условие предполагает поиск строк, которые могут, быть не выбраны в следствии того, что столбец **SALES** или **QUOTA**, могут иметь значение **NULL**!

Такое условие еще как-то осмыслено, но вот например выражение: **((SALES - QUOTA) > 100.000) IS FALSE**, вообще бессмысленно! Проще написать: **NOT ((SALES - QUOTA) > 100.000)** по этому в литературе, для обеспечения максимальной переносимости, не рекомендуется применять оператор **IS** в таком контексте, я например пробовал это все на **Oracle**, чего либо внятного к сожалению не получилось! :))) Вот такие события!

Шаг 18 - Сортировка записей запроса, предложение ORDER BY

Ранее в запросах, которые мы с вами приводили, результирующие выборки, получались, в произвольном порядке. Что если нужно вывести скажем, список, учеников школы в алфавитном порядке или стоимость товаров по убыванию? Для этого в операторе **SELECT** предусмотрено предложение **ORDER BY**. Вот его синтаксис:

```
----- ORDER BY -- имя столбца ----- ,
-- порядковый номер столбца --- ----- ASC -----
--                               ----- DESC -----
----- , -----
```

Для начала, давайте сделаем следующий пример: Показать фактические объемы продаж для каждого офиса, отсортированные в алфавитном порядке по названиям, регионов и в каждом регионе - по названию городов.

```
SQL> SELECT CITY, REGION, SALES
2 FROM OFFICES
3 ORDER BY REGION, CITY
4 /
```

CITY	REGION	SALES
Бубурино	Восток	367,911
Красный Мотоцикл	Восток	692,637
Чугуевск	Восток	735,044
Запиндрищинск	Запад	186,042
Котрогайка	Запад	835,915

Столбец идущий сразу за предложением **ORDER BY** является ГЛАВНЫМ ключом, столбцы следующие за ним, являются ВТОРОСТЕПЕННЫМИ ключами. Сортировать записи можно как по возрастанию, так и по убыванию.

Например, в следующем выражении: Вывести список офисов, отсортированный по фактическим объемам продаж в порядке убывания.

```
SQL> SELECT CITY, REGION, SALES
2 FROM OFFICES
3 ORDER BY SALES DESC
4 /
```

CITY	REGION	SALES
Котрогайка	Запад	835,915
Чугуевск	Восток	735,044
Красный Мотоцикл	Восток	692,637
Бубурино	Восток	367,911
Запиндрищинск	Запад	186,042

Получаем отсортированные объемы продаж по убыванию, с применением предиката **DESC**, для сортировки по возрастанию, применяется **ASC**, вследствие того, что данный тип сортировки применяется по умолчанию, его можно не указывать. Так же, если столбец сортировки вычисляемый и не имеет имени, в выражении **ORDER BY** можно просто указать его порядковый номер!

Например, вот так: Вывести список офисов, отсортированный по разности между фактическим и плановым объемам продаж в порядке убывания.

```
SQL> SELECT CITY, REGION, (SALES - TARGET)
2 FROM OFFICES
3 ORDER BY 3 DESC
4 /
```

CITY	REGION	(SALES-TARGET)
Красный Мотоцикл	Восток	117,637
Котрогайка	Запад	110,915
Бубурино	Восток	17,911
Чугуевск	Восток	-64,956
Запиндрищинск	Запад	-113,958

Так же применяя в выражении **ORDER BY** имена столбцов, номера столбцов, а так же выражения **DESC**, **ASC**, возможно строить достаточно сложные условия сортировки.

Например: Вывести список офисов, отсортированный в алфавитном порядке по названиям регионов, а в каждом регионе по - разности между фактическим и плановым объемам продаж в порядке убывания.

```
SQL> SELECT CITY, REGION, (SALES - TARGET)
2 FROM OFFICES
3 ORDER BY REGION ASC, 3 DESC
4 /
```

CITY	REGION	(SALES-TARGET)
Красный Мотоцикл	Восток	117,637
Бубурино	Восток	17,911
Чугуевск	Восток	-64,956
Котрогайка	Запад	110,915
Запиндрищинск	Запад	-113,958

Таким образом, вам будет легко, задать необходимый порядок сортировки вашего запроса и не будет вызывать особенных трудностей! Пробуйте!

Шаг 19 - Подводим итог по запросам!

Итак, наконец, одно из направлений, а именно однотабличные запросы, мы, наконец, закончили! Теперь давайте подведем маленький итог и сформулируем правила выполнения однотабличных запросов. В принципе такой тип запросов легко читаем, достаточно внимательно посмотреть на оператор **SELECT** и, как правило, становится все ясно, тем не менее, давайте опишем все сухими постулатами! Иногда это полезно! Итак:

1. ВЗЯТЬ ТАБЛИЦУ, УКАЗАННУЮ В ОПЕРАТОРЕ **FROM**.
2. ЕСЛИ ИМЕЕТСЯ ПРЕДЛОЖЕНИЕ **WHERE**, ПРИМЕНИТЬ ЗАДАННОЕ В НЕМ УСЛОВИЯ ПОИСКА К КАЖДОЙ СТРОКЕ ТАБЛИЦЫ И ОСТАВИТЬ ТОЛЬКО ТЕ СТРОКИ, ДЛЯ КОТОРЫХ ЭТО УСЛОВИЕ ВЫПОЛНЯЕТСЯ, Т.Е. ИМЕЕТ ЗНАЧЕНИЕ **TRUE(!)**. СТРОКИ, ДЛЯ КОТОРЫХ УСЛОВИЕ ПОИСКА ИМЕЕТ ЗНАЧЕНИЕ **FALSE** ИЛИ **NULL** - ОТБРОСИТЬ.
3. ДЛЯ КАЖДОЙ ИЗ ОСТАВШИХСЯ СТРОК ВЫЧИСЛИТЬ ЗНАЧЕНИЕ КАЖДОГО ЭЛЕМЕНТА В СПИСКЕ ВОЗВРАЩАЕМЫХ СТОЛБЦОВ И СОЗДАТЬ ОДНУ СТРОКУ ТАБЛИЦЫ РЕЗУЛЬТАТОВ ЗАПРОСА. ПРИ КАЖДОЙ ССЫЛКЕ НА СТОЛБЕЦ ИСПОЛЬЗУЕТСЯ ЗНАЧЕНИЕ СТОЛБЦА ДЛЯ ТЕКУЩЕЙ СТРОКИ.
4. ЕСЛИ УКАЗАНО КЛЮЧЕВОЕ СЛОВО **DISTINCT**, УДАЛИТЬ ИЗ ТАБЛИЦЫ РЕЗУЛЬТАТОВ ЗАПРОСА ВСЕ ПОВТОРЯЮЩИЕСЯ СТРОКИ.
5. ЕСЛИ ИМЕЕТСЯ ПРЕДЛОЖЕНИЕ **ORDER BY** ОТСОРТИРОВАТЬ РЕЗУЛЬТАТЫ ЗАПРОСА.

Вот в принципе такие правила, может быть витиевато, но на то они и правила. Если вы будете их придерживаться, то все будет замечательно и проблем не возникнет!!! :) Прежде чем начинать новую тему, давайте рассмотрим, оператор **UNION**, он позволяет объединить результаты двух запросов. Тем более что, как раз он и предварит тему многотабличных запросов. Сказать честно, я за всю свою практику такую систему объединения двух **SELECT**'ов не применял, хотя может кому-то это и нужно! :)))

Давайте рассмотрим для начала два простых запроса, первый: Вывести список всех товаров, цены которых превышают \$500.

```
SQL> SELECT MFR_ID, PRODUCT_ID
2   FROM PRODUCTS
3  WHERE PRICE > 500.00
4  /
```

MFR PRODU

--- -----

```
IMM 779C
BIC 41003
REI 2A44L
IMM 775C
ACI 4100Z
REI 2A44R
IMM 773C
```

7 строк выбрано.

Получаем следующее. 7 строк и два столбца.

А теперь второй запрос: Вывести список всех товаров, которых было заказано более чем \$4.0 за один раз.

```
SQL> SELECT DISTINCT MFR, PRODUCT
2   FROM ORDERS
3  WHERE AMOUNT > 4.00
4  /
```

MFR PRODU

--- -----

ACI 41002
ACI 41004
ACI 4100X
ACI 4100Y
ACI 4100Z
BIC 41003
IMM 775C
IMM 779C
QSA XK47
REI 2A44L
REI 2A44R
REI 2A45C

12 строк выбрано.

Получаем 12 строк и два столбца.

А вот теперь применим **UNION**! Вывести список всех товаров, цены которых превышают \$500 или которых было заказано более чем \$4.0 за один раз.

```
SQL> SELECT MFR_ID, PRODUCT_ID
2   FROM PRODUCTS
3  WHERE PRICE > 500.00
4 UNION
5 SELECT DISTINCT MFR, PRODUCT
6   FROM ORDERS
7  WHERE AMOUNT > 4.00
8  /
```

MFR PRODU

--- -----

ACI 41002
ACI 41004
ACI 4100X
ACI 4100Y
ACI 4100Z
BIC 41003
IMM 773C
IMM 775C
IMM 779C
QSA XK47
REI 2A44L

```
REI 2A44R
REI 2A45C
```

13 строк выбрано.

Интересно, правда, 13 строк и два столбца, а почему их не 19ть? Ведь один из запросов вернул, 7мь, а второй 12ть! Вот в этом и есть объединение, посмотрите внимательнее на значения в первом и втором запросе:

```
MFR PRODU
--- -----
IMM 779C
BIC 41003
REI 2A44L
IMM 775C
ACI 4100Z
REI 2A44R
IMM 773C
```

```
MFR PRODU
--- -----
ACI 41002
ACI 41004
ACI 4100X
ACI 4100Y
ACI 4100Z
BIC 41003
IMM 775C
IMM 779C
QSA XK47
REI 2A44L
REI 2A44R
REI 2A45C
```

Правда, много похожего, а объединенный не стал повторяться, а просто показал 13ть строк, вот и все! При таком запросе возвращаемые столбцы, если они повторяются, то повторяющиеся столбцы просто отбрасываются. Так как при действии предложения **DISTINCT** хотя в одном из запросов он присутствует явно! Главное, чтобы запросы содержали одинаковое количество столбцов и тип столбцов в обоих запросах был одинаковый! И еще **ORDER BY** в таких запросах не применяется! А вот если, все таки нужно вернуть все строки объединенного запроса, то просто укажите **ALL** после оператора **UNION**:

```
SQL> SELECT MFR_ID, PRODUCT_ID
2 FROM PRODUCTS
3 WHERE PRICE > 500.00
4 UNION ALL
5 SELECT DISTINCT MFR, PRODUCT
6 FROM ORDERS
7 WHERE AMOUNT > 4.00
8 /
```

```
MFR PRODU
```

```
--- -----
```

```
IMM 779C
BIC 41003
REI 2A44L
IMM 775C
ACI 4100Z
REI 2A44R
IMM 773C
ACI 41002
ACI 41004
ACI 4100X
ACI 4100Y
ACI 4100Z
BIC 41003
IMM 775C
IMM 779C
QSA XK47
REI 2A44L
REI 2A44R
REI 2A45C
```

19 строк выбрано.

Оп! Вот и все 19ть строк! Что и требовалось! Теперь надеюсь стало понятно. Единственное скажу, что удаление повторяющихся строк, в таких запросах, занимает много времени, так как эти типы запросов ресурсоемки! :) Хотя я и говорил, что **ORDER BY** не используют, можно в принципе, попробовать следующее:

```
SQL> SELECT MFR_ID, PRODUCT_ID
2 FROM PRODUCTS
3 WHERE PRICE > 500.00
4 UNION ALL
5 SELECT DISTINCT MFR, PRODUCT
6 FROM ORDERS
7 WHERE AMOUNT > 4.00
8 ORDER BY 1,2
9 /
```

```
MFR PRODU
```

```
--- -----
```

```
ACI 4100X
ACI 4100Y
ACI 4100Z
ACI 4100Z
ACI 41002
ACI 41004
BIC 41003
BIC 41003
IMM 773C
IMM 775C
IMM 775C
IMM 779C
```

```
IMM 779C
QSA XK47
REI 2A44L
REI 2A44L
REI 2A44R
REI 2A44R
REI 2A45C
```

19 строк выбрано.

Ух, ты! Сработало! Только имена столбцов не используйте, так как может возникнуть путаница! Лучше их порядковые номера! Вот такие примеры! :) Так же можно писать многократные запросы на объединение:

Например:

```
SELECT * FROM A
UNION (SELECT *
        FROM B
        UNION
        SELECT *
        FROM C)
```

```
-----
Вася
Петя
Коля
Маша
```

4 строки выбрано.

Скобки показывают какой оператор **UNION** должен выполняться первым. Так же независимо от того исключаются повторяющиеся строки или нет, выражения например такого типа полностью эквивалентны:

```
A UNION (B UNION C)
(A UNION B) UNION C
(A UNION C) UNION B
```

Если применять предложение **ALL**, то следующий тип объединения, так же одно и то же!

```
A UNION ALL (B UNION ALL C)
(A UNION ALL B) UNION ALL C
(A UNION ALL C) UNION ALL B
```

Если вы применяете и **UNION** и **UNION ALL**, то выражение типа:

```
A UNION ALL B UNION C
```

Переписать как:

A UNION ALL (B UNION C)
или
(A UNION ALL B) UNION C

То результаты таких запросов будут РАЗЛИЧНЫ!!!

Не путайтесь, в таких случаях и, применяя такие выражения, четко представляйте себе, чего вы хотите получить! Собственно вот такие дела, пробуйте еще раз все самостоятельно и усваивайте материал! :)))

Шаг 20 - Подходим ближе! Основные компоненты БД Oracle

Не скрою, перед тем как начать этот шаг, мне пришлось, кое что полистать. :) Так как, вобрать в себя все то, что я вам собираюсь рассказать, не так то просто! Материал очень обширный и состоит из множества понятий и определений! Давайте договоримся сразу, я попытаюсь, донести все это, как можно понятнее и проще. По этому если буду, делать обобщения, то не в ущерб самому предмету, согласитесь, все знать не возможно, а мне просто доставляет удовольствие все это проходить вместе с вами и за одно освежать все в памяти!!! Если я где-либо ошибусь, то можете мне на это указать! Я не обижусь!!! Так как не являюсь истиной в последней инстанции. Давайте собственно, переходить к делу! Зададимся таким тривиальным вопросом, что же такое собственно **база данных** (БД) и в частности БД **Oracle**? По своей сути сам термин БД, подразумевает систему обработки данных и определяет физическую и логическую структуру данных в ней находящуюся! Немного запутано, но понять можно! А сама БД, находится в ведении **системы управления реляционными базами данных** (СУРБД)! Естественно каждая БД, состоит из набора компонент. Часть из них, используется внутри самой системы и необходима собственно для функционирования самой СУБД, а часть, доступна любому внешнему процессу. Так вот первые называются **системными объектами**, а вторые **пользовательскими объектами**. Давайте для начала, разберемся с **системными объектами**. Они в БД **Oracle**, включают такие основные понятия как:

1. Файлы параметров инициализации.
2. Управляющие файлы.
3. Оперативные и архивные файлы журналов регистрации транзакций.
4. Файлы трассировки.
5. Идентификаторы записей **ROWID** (Их еще называют "Атомарная матрица Oracle" :))
6. Блоки ORACLE.

Файл параметров инициализации имеющий имя **init.ora**, собственно является основным средством настройки БД. Он представляет из себя обычный **ASCII** файл, содержащий ряд параметров, которые БД, использует при старте и последующем разворачивании в ОС. Правда здесь есть один нюанс, сама БД ищет, файл инициализации с именем **initSID.ora**. Где **SID**, напомним, если кто забыл, это служебное имя вашего экземпляра БД. В нашем случае, если вы устанавливали БД, как я вам предлагал, это будет **PROBA** (например, моя экспериментальная БД имеет в качестве **SID** значение **HOME**. Не самое удачное решение, хотя она у меня уже скоро развалится, и я ее переделаю, а за одно выберу другое имя!). Так вот ваш файл инициализации БД будет находиться в каталоге **\$ORACLE_HOME\DATABASE**, **\$ORACLE_HOME** - это в вашем случае каталог **C:\Oracle\Ora81**, значит в совокупности получается **C:\Oracle\Ora81\DATABASE**. Там должен лежать файл с именем **initPROBA.ora**, загляните что у него внутри, должно быть что-то вроде:

```
IFILE='C:\Oracle\admin\proba\pfile\init.ora'
```

Вот теперь надеюсь ясно, параметр **IFILE** просто указывает вашему экземпляру БД, где искать именно свой, а не чей попало, файл инициализации. Найдя этот файл ваша БД, счастливо стартует и начинает свою трудовую деятельность! Параметр **IFILE** вообще-то и применяется для лучшего структурирования вашего севера **Oracle**! А вот в каталоге **\$ORACLE_HOME\dfs**, есть еще один файл **init.ora**, если его внимательно изучить, то можно настроить любую БД **Oracle** в различных вариантах исполнения. Малая, средняя, крупная БД, ну и т. д. А теперь давайте заглянем внутрь вашего, файла **init.ora**. Можно увидеть примерно следующее:

```
db_name = proba
```

```
db_domain = com

instance_name = proba

service_names = proba.com

db_files = 1024

control_files = ("C:\Oracle\oradata\proba\control01.ctl", "C:\Oracle\oradata\proba\control02.ctl")

db_file_multiblock_read_count = 8

db_block_buffers = 49285

shared_pool_size = 134582272

log_checkpoint_interval = 10000

log_checkpoint_timeout = 1800

processes = 50

parallel_max_servers = 5

log_buffer = 32768

#audit_trail = true # if you want auditing
#timed_statistics = true # if you want timed statistics
max_dump_file_size = 10240 # limit trace file size to 5M each

# Global Naming -- enforce that a dblink has same name as the db it connects to
global_names = true

# Uncomment the following line if you wish to enable the Oracle Trace product
# to trace server activity. This enables scheduling of server collections
# from the Oracle Enterprise Manager Console.
# Also, if the oracle_trace_collection_name parameter is non-null,
# every session will write to the named collection, as well as enabling you
# to schedule future collections from the console.
# oracle_trace_enable = true

oracle_trace_collection_name = ""
# define directories to store trace and alert files
background_dump_dest = C:\Oracle\admin\proba\bdump
user_dump_dest = C:\Oracle\admin\proba\udump

db_block_size = 8192

remote_login_passwordfile = exclusive

os_authent_prefix = ""
```

```
distributed_transactions = 10
compatible = 8.1.0
sort_area_size = 66560
```

В вашем случае его содержимое может быть различным, в зависимости кто, как ставился сам сервер, но основные параметры будут примерно одинаковыми, так что не огорчайтесь, если увидите что-то не похожее на мой пример! :) Все параметры мы прямо сейчас разбирать не станем, остановимся лишь на нескольких. Тем более что, мы еще не раз будем возвращаться к этому файлу.

```
db_name = proba
```

Это собственно и есть тот самый **SID**, вашего экземпляра БД. Экземпляров, может быть, много и у каждого свой уникальный **SID**!

```
db_domain = com
```

А это вторая часть, доменного имени, которая идет сразу за дот (.). То есть **proba.com** или как-то еще, как вам больше нравится!

```
service_names = proba.com
```

А это имя сервиса вашего экземпляра БД, то есть два предыдущих параметра вместе. Пока все понятно?

Идем дальше! Собственно все параметры инициализации вашего экземпляра БД можно просмотреть через представление **v\$parameter**. Таких "представлений" в самой БД сотни. С их помощью о вашем экземпляре, можно узнать все. Именно их применяют в своей работе люди, гордо носящие имя Администраторы БД! Можно сказать, что один из них, ваш покорный слуга, так как пару сотен "представлений" я уже изучил! Давайте откроем **SQLPlus** и дадим такой запрос, естественно воспользовавшись знаниями об однотабличных запросах из прошлых шагов:

```
SELECT a.name, a.value
FROM v$parameter a
ORDER BY a.name
/
```

```
.
.
.
.
```

NAME	VALUE
db_block_lru_latches	1
db_block_max_dirty_target	49285
db_block_size	8192
db_domain	com
db_file_direct_io_count	64
db_file_multiblock_read_count	8
db_file_name_convert	
db_files	1024

dblink_encrypt_login	FALSE
db_name	proba
dbwr_io_slaves	0
db_writer_processes	1
disk_asynch_io	TRUE
distributed_transactions	10
dml_locks	264
enqueue_resources	1308
ent_domain_name	
event	
fast_start_io_target	49285
fast_start_parallel_rollback	LOW
fixed_date	
.	
.	
.	

Приведу только часть, запроса. Так как в моем случае, было 195 строк! Но из этой части достаточно, хорошо видно, что все параметры представлены в очень удобном виде и очень наглядны! Напомню, что изменять параметры лучше в самом файле **init.ora**! И не забудьте, что изменения вступят в силу только после рестарта вашего экземпляра БД! Но пока настоятельно не рекомендую что-либо менять, особенно параметр **db_block_size**, если это сделать, то может случиться непоправимое! :))) Вот собственно пока все о первом системном объекте. Тема получилась объемной, по этому продолжим в следующий раз!

Шаг 21 - Основные компоненты БД Oracle - продолжаем!

Итак, замечательно! После несколько затянувшегося процесса изучения оператора **SELECT**, что собственно само по себе не маловажно и мы еще будем его разбирать, наконец подходим вплотную собственно к самому серверу **Oracle**! Итак, в прошлый раз мы немного разобрались с файлом **init.ora**. Теперь, двигаемся дальше!

2. Управляющие файлы.

Этот файл, еще он называется "контролфайл", содержит в себе практически всю информацию, необходимую для функционирования вашей БД. Такую, например как, набор символов используемый БД, файлы регистрации транзакций используемые БД, файлы данных БД и их статус, обновления, положения и т.д. Эти файлы, создаются во время создания собственно вашего экземпляра БД и с течением времени практически не меняются, за исключением того, если вы как администратор производите какие-либо действия с экземпляром. Например, добавляете файл данных нового табличного пространства. Меняете размер существующих файлов данных и т.д. Без контрольного файла БД не стартует! Или если он поврежден! Контролфайл имеет бинарную структуру, по этому редактировать его в ручную нельзя! Поврежденный контролфайл можно восстановить, эту процедуру я вам покажу чуть позже! А находятся эти файлы, в вашем случае в каталоге **C:\Oracle\Oradata\Proba**. Обычно этих файлов два. Так как БД работает с размноженными контролфайлами. За расположение этих файлов отвечает секция **CONTROL_FILES** в файле **init.ora** вашего экземпляра БД. И естественно все котролфайлы перечислены в представлении **v\$controlfile**.

Например:

```
SELECT * FROM v$controlfile
```

Получаем:

```
STATUS NAME
-----
C:\ORACLE\ORADATA\PROBA\CONTROL01.CTL
C:\ORACLE\ORADATA\PROBA\CONTROL02.CTL
```

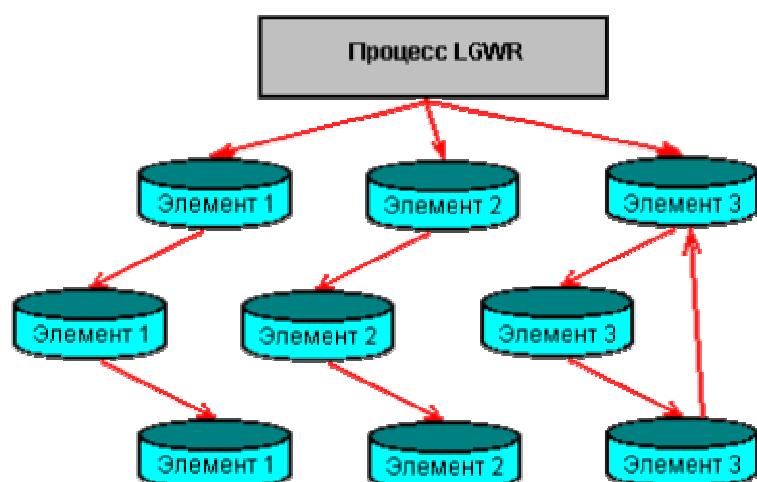
3. Оперативные и архивные файлы журналов регистрации транзакций.

При создании экземпляра БД, после его старта, в системной области, под названием **SGA** практически сразу запускается группа фоновых процессов. Один из них и пожалуй самый важный **LGWR**! Это процесс регистрации транзакций. Помимо всего прочего, он ведет так называемые "журналы регистрации транзакций". Записывает он это все в файлы расположенные там же где и контролфайлы БД, а именно **C:\Oracle\Oradata\Proba** в вашем случае. Называются эти файлы:

```
redo01.log
redo02.log
redo03.log
.
.
.
redo(n).log
```

При работе используется как минимум две группы файлов! Запись проводится циклически в обе группы.

По мере заполнения групп **LGWR** открывает новые, а заполненные процесс **ARCH** переписывает на резервные накопители, если база запущена с опцией архивирования, что не всегда выполняется. Хотя это полезно, но отнимает часть ресурсов процессора.



Просмотреть состояние этих журналов нам поможет такой запрос:

```
SELECT b.member, a.bytes, a.members, a.status
FROM v$log a, v$logfile b
WHERE a.group# = b.group#
ORDER BY b.member
/
```

Получаем:

MEMBER	BYTES	MEMBERS	STATUS
C:\ORACLE\ORADATA\PROBA\REDO01.LOG	1048576	1	CURRENT
C:\ORACLE\ORADATA\PROBA\REDO02.LOG	1048576	1	INACTIVE

Как видите, мы снова обращаемся к представлениям **v\$log** и **v\$logfile** и снова получаем полезную информацию!

Как видно файл **REDO01.LOG** является активным в данный момент! Идем дальше!

4. Файлы трассировки

Файлов трассировки, существует несколько видов. Основное их назначение фиксировать все события происходящие в сервере БД, в том числе все критические. В дальнейшем, по их содержимому можно выяснить вследствие чего произошел тот или иной сбой! Хотя, если все нормально **Oracle** обычно не сбойт! Но тем не менее. Основной файл трассировки системных событий имеет имя **sidALRT.log**, где **sid** (замете, маленькими буквами!) имя экземпляра. Положение этого файла определяет секция **background_dump_dest** файла **init.ora** в нашем

случае **C:\Oracle\admin\proba\bdump**. Там же находятся и файлы трассировки фоновых процессов! А секция файла **init.ora**, **user_dump_dest** определяет местоположение пользовательских файлов трассировки. В вашей БД это будет **C:\Oracle\admin\proba\udump**. Пользовательские файлы трассировки, имеют префикс **ora**, затем уникальный номер и расширение **.trc**. Эти файлы формируются в случае возникновения критической ситуации пользовательского процесса. Запустить режим трассировки пользовательского процесса. Постоянно, можно введя в **SQLPlus** команду:

```
ALTER SESSION SET SQL_TRACE=TRUE  
/
```

Или установить в файле **init.ora**, в секции **SQL_TRACE** значение **TRUE**! Но такой ход, может повлечь за собой немало неприятностей, ваши файлы трассировки неимоверно распухнут! И разобрать в них что либо, можно будет уже с трудом! По этому делайте это осторожно, по мере необходимости! :))

5. Идентификаторы записей ROWID

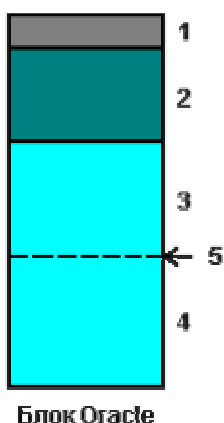
А вот самое интересное! "АТОМАРНАЯ МАТРИЦА"! (МАТРИЦА!!!:))))! Теоретически, каждая запись в БД должна однозначно идентифицироваться! Такую возможность мы получаем в сервере **Oracle** с помощью информационной структуры **ROWID**. **ROWID** представляет из себя двухбайтовую величину, которая соответствует физическому положению любой строки в БД! **ROWID** имеет формат **BBBBBBBB.RRRR.FFFF** где:

- **BBBBBBBB** - шестнадцатиричный номер блока в файле данных, в котором находится строка.
- **RRRR** - шестнадцатиричный номер строки в блоке.
- **FFFF** - файл в состав, которого входит это блок.

Например, **ROWID = 000078F.0003.0002**. Это значит, что строка находится во втором файле данных, в блоке номер **78F** файла данных, и является четвертой строкой (счет, идет от 0000!) в блоке. При создании новой строки она сразу получает свой уникальный **ROWID**! И самый быстрый способ найти строку это указать ее **ROWID**! Хотя это не такая простая задача!!! Это далеко не все про **ROWID**, тем не менее, для начала понимания уже достаточно! Так что в следующий раз займемся самым интересным, блоками **Oracle**.

Шаг 22 - Основные компоненты БД Oracle - Блоки Oracle

Итак, давайте попробуем разобраться, с самой фундаментальной частью БД **Oracle**, которая называется "блок БД". Блоки являются самой элементарной единицей выборки данных. И стоят на самом нижнем уровне организации собственно самой БД. Блоки хранят всю информацию, из которой собственно и состоит ваша БД. Сразу оговариваюсь, не путайте блоки БД с блоками файловой системы, какая бы она не была на вашем сервере, **FAT**, **FAT32**, **NTFS** и т.д. Блоки БД, естественно базируются на блоках файловой системы, но с последними ничего общего не имеют!!! :) Хотя блоки БД, должны быть, кратны блокам файловой системы. Например, для **UNIX** это 8192, 16384 и т.д. Выбирать, размер блока БД можно самостоятельно при создании экземпляра БД, либо система задает этот параметр сама. В том случае как мы с вами устанавливали БД, размер блока у вас, скорее всего будет 2048 байт. В этом легко убедиться, проверив значение параметра **db_block_size**, в уже известном вам файле **init.ora**. Естественно при чтении данных из БД, считывается столько байт, сколько входит в один блок или кратное этому числу, в зависимости от количества информации, например, в запросе. Так же запомните сразу, единожды установленный размер блока БД, в процессе уже не может быть изменен!!! Если вы, по какой либо причине захотите изменить, размер блока БД, то необходимо сохранить ваши данные, затем удалить ваш экземпляр БД, и создать его вновь с новым значением блока БД! Что же представляет из себя блок БД? На рисунке представлено схематичное изображение блока.



Цифрой "1" обозначен **заголовок блока**, в нем хранится информация о том, к какому сегменту БД, принадлежит блок, количество одновременных транзакций и т.д. Цифрой "2" обозначено пространство, зарезервированное параметром **PCTFREE** для дальнейших обновлений данных в блоке. Что это за параметр мы сейчас разберем. Цифра "3" и "4" это пространство готовое к использованию, а вот цифра "5" показывает границу параметра **PCTUSED**. Что, это за параметры? **PCTFREE** - указывает какое количество (в процентах) пространства будет зарезервировано для дальнейших обновлений данных в блоке, а **PCTUSED** задает часть объема блока, которое должно освободиться прежде чем БД включит его в список доступных для ввода новых строк. Давайте разберем это все, так чтобы стало понятно! Например, вы создали таблицу **MYTABLE**, в которую поместили скажем 100 записей, ваша таблица заняла, фигурально один блок. Теперь, при создании БД параметр **PCTFREE**, в нашем случае, имел значение 10%, а параметр **PCTUSED** имел значение 40%. Что, это значит? Теперь ваш блок, имеет 10% пространства для обновления находящейся в нем таблицы, а 90% пространства отводятся под саму таблицу. То есть, если ваша таблица заполнила 90% процентов блока, то он вычеркивается самой БД из списка доступных для ввода новых строк БД. Но, те 10% процентов так и остались в блоке, для обновления вашей **MYTABLE**. Теперь скажем, вы обновили данные в вашей **MYTABLE** таким образом, что она стала занимать не

90% процентов пространства блока, а скажем 49% блока. Итак, $90\% - 49\% = 41\%$, а параметр **PCTUSED = 40%**, замечательно, блок снова включается в список для записи новых строк! Теперь сюда можно поместить, скажем, таблицу **NEWTABLE**! Вот таким образом эти два параметра работают и управляют блоками БД. При этом ни один из параметров никогда не должен иметь значение 100%! С помощью **PCTFREE** и **PCTUSED** можно регулировать производительность самой БД. Забегая немного вперед скажу, что экземпляры БД имеют два типа при построении. Так называемые **OLTP** и **DSS** системы, первая это БД рассчитанная на тысячи активных транзакций, а вторая это хранилище данных используемое в основном для чтения, так вот правильная настройка блоков и их размер обязательно необходимое условие для производительного функционирования обоих типов БД! Посмотреть ваши параметры настройки блоков, как я уже говорил, можно в файле **init.ora** в секции **db_block_size**, например:

```
init.ora
.
.
db_block_size = 2048
.
.
```

А значение параметров **PCTFREE** и **PCTUSED** можно посмотреть, войдя пользователем **SYS** или **SYSTEM** в **SQL*Plus** и написав такой запрос:

```
SQL> SELECT a.OWNER, a.TABLE_NAME, a.TABLESPACE_NAME, a.PCT_FREE, a.PCT_USED
2 FROM DBA_TABLES a
3 WHERE OWNER = 'MILLER'
4 /
```

OWNER	TABLE_NAME	TABLESPACE_NAME	PCT_FREE	PCT_USED
MILLER	CUSTOMERS	USERS	10	40
MILLER	OFFICES	USERS	10	40
MILLER	ORDERS	USERS	10	40
MILLER	PRODUCTS	USERS	10	40
MILLER	SALESREPS	USERS	10	40

5 rows selected

Как видно, в нашем случае **PCTFREE = 10%** и **PCTUSED = 40%**, в чем мы с вами и убедились! В дальнейшем мы еще, вернемся к этой теме, так как она еще далеко не полностью раскрыта. Но пока на этом с системными объектами БД мы закончили. Советую, еще раз все осмыслить и хорошенечко и запомнить на будущее.

Шаг 23 - Основные компоненты БД Oracle - Пользовательские объекты БД

Вот, наконец, с системными, объектами мы немного разобрались, теперь пора переходить к объектам, которые имеют названия "пользовательские объекты БД". Пользовательские объекты это те объекты, которые содержатся в контексте БД, собственно для них сама БД и создавалась. Этот тип объектов не находится в исключительном ведении самой БД, а скорее в ведении пользователей самой БД. К такого рода объектам в частности, относятся и файлы данных, так как при создании табличных пространств, создаются и файлы данных, но файлы данных являются так же "физическими объектами БД", об этом не следует забывать. Итак, рассмотрим первое понятие - "файлы данных".

Файл данных БД, представляет собой реальный файл, операционной системы. Он доступен для просмотра, но выполнять с ним какие либо действия рекомендуется только с помощью средств БД! Файлы данных хранят "табличные пространства", что это такое чуть позже, а пока скажу только, что один файл данных хранит только одно "табличное пространство"! А создаются файлы данных с помощью, команд **CREATE TABLESPACE** и **ALTER TABLESPACE**. Размер, этих файлов определяется при создании и может быть изменен в процессе работы, как в сторону увеличения, так и в сторону уменьшения, но не меньше чем объем данных, которые в нем находятся! Так же, если вы создали файл данных (объявив новое табличное пространство), скажем 20Мгб, то файл и будет размером 20Мгб, не смотря на то, что будет в нем, таблица с одной строкой или с миллионом строк! Просмотреть все файлы данных ("табличные пространства") можно заглянув в каталог, **C:\Oracle\ORADATA\proba**, все файлы с расширением **dbf** и есть файлы данных или "табличные пространства". Либо можно дать, такой запрос в **SQL*Plus**, естественно войдя пользователем **SYSTEM** с паролем **MANAGER**:

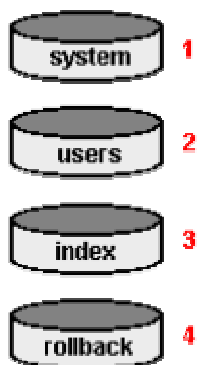
```
SQL> SELECT tablespace_name, file_name, bytes
2 FROM dba_data_files
3 ORDER BY tablespace_name, file_name
4 /
```

TABLESPACE_NAME	FILE_NAME	BYTES
INDX	C:\ORACLE\ORADATA\PROBA\INDX01.DBF	2097152
OEM_REPOSITORY	C:\ORACLE\ORADATA\PROBA\OEMREP01.DBF	5242880
RBS	C:\ORACLE\ORADATA\PROBA\RBS01.DBF	26214400
SYSTEM	C:\ORACLE\ORADATA\PROBA\SYSTEM01.DBF	146800640
TEMP	C:\ORACLE\ORADATA\PROBA\TEMP01.DBF	2097152
USERS	C:\ORACLE\ORADATA\PROBA\USERS01.DBF	3145728

6 rows selected

Отсюда хорошо видно, что мы имеем 6-ть файлов данных, или - "табличные пространства".

Итак, что это такое? Это собственно и есть те самые файлы данных, основные табличные пространства, которые создаются при автоматической установке БД, хорошо видно из предыдущего запроса, а именно:



1. **SYSTEM** - хранит все словари данных и системные объекты.
2. **USERS** - для хранения пользовательских объектов.
3. **INDX** - табличное пространство для организации индексов БД.
4. **RBS** - табличное пространство сегментов отката.

Вспомогательные:

- **OEM_REPOSITORY** - специальное пространство, которое использует администратор БД (чуть позже).
- **TEMP** - для нужд, остальных пространств.

Данная классификация весьма не полная, но пока это, для того чтобы было понятнее. Табличные пространства, это еще одна сущность сервера БД **Oracle**. То есть вся система собственно и базируется на табличных пространствах. Их основное отличие от файлов данных, это то, что табличное пространство может содержать несколько файлов данных, по этому не путайте эти два понятия! Табличное пространство, следуя строгим определениям, не что иное, как логическая структура, используемая для группировки данных с однотипными методами доступа. Надеюсь это понятно. Так же табличные пространства, это основные объекты операций резервного копирования и восстановления. Кстати, есть скептики, которые считают, что разделение табличных пространств на отдельные файлы системы не очень хорошо, но с этим можно поспорить, если учитывать физическую структуру БД. Хотя в таких БД как **InterBase**, **Access**, все хранится в одном физическом файле. Но это не всегда оправдывает себя. И структура организации табличных пространств группой файлов все же имеют свое преимущество. Так же, данные о табличных пространствах можно получить с помощью такого запроса:

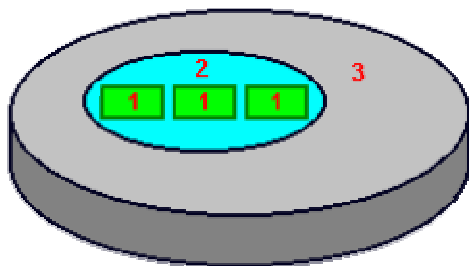
```
SQL> SELECT a.TABLESPACE_NAME, a.PCT_INCREASE, a.STATUS  
2 FROM DBA_TABLESPACES a  
3 ORDER BY TABLESPACE_NAME  
4 /
```

TABLESPACE_NAME	PCT_INCREASE STATUS
INDX	50 ONLINE
OEM_REPOSITORY	50 ONLINE
RBS	50 ONLINE
SYSTEM	50 ONLINE
TEMP	50 ONLINE
USERS	50 ONLINE

6 rows selected

Как видите, мы снова, воспользовались, словарем данных, который так же хранится в табличном пространстве **SYSTEM**. Надеюсь теперь понятно, что же такое табличное пространство. :)

Теперь - "**расширения**" (**extents**). Сформулирую сразу. Расширения - это объекты информационной структуры данных, образованные непрерывными блоками БД **Oracle**, одним или несколькими. Так как каждый сегмент БД состоит из одного или нескольких расширений, "экстентов". Скажу прямо, у меня долго был туман в голове, прежде чем я все-таки понял, что такое расширение. Во-первых, что такое "сегмент", забегаю чуть вперед, (это приходится делать постоянно, так как без этого не обойтись!) это созданный пользователем объект БД, а именно таблица, процедура, функция и т.д. Каждый сегмент, имеет одно или несколько расширений, количество расширений для сегмента БД, зависит от параметра **CREATE**. Так вот объясняю понятнее. Допустим, вы создавали таблицу учета избирателей по округу. Вы создали таблицу, затем стали заливать в нее данные, что-то получилось, что-то не получилось. Вы добавляли записи, удаляли, меняли, структуру таблицы и т.д. в результате у вас получился полный хаос в табличном пространстве, ваша таблица, по мере заполнения и создания, стала занимать скажем 1000000 блоков, в пяти расширениях. Если эта таблица, после того как вы закончили имеет, скажем, 2000000 записей, но находится в таком плачевном состоянии, то сделать к ней производительный запрос вряд ли удастся, если ее даже оптимизировать и проиндексировать. Я поступал так. После того, как моя таблица была готова, я создавал, новое табличное пространство и заливал туда мою готовую таблицу в результате все ложилось, одним расширением ("экстентом")! То есть таблица не имела рваных блоков и расширений. Что собственно неизбежно при создании. Сейчас есть более элегантные методы "переукладки" табличного пространства, например, в **Oracle 9i** такое средство администрирования, как **Enterprise Manager** может оптимизировать табличное пространство, как **speeddisk** в **Windows**! Но как это делается "ручками" тоже полезно знать, если вы истинный администратор БД! Вот собственно, что такое "расширение":



- **1** - Это цепочка блоков образующих расширение или экстент.
- **2** - Это само расширение.
- **3** - Это, какое-либо табличное пространство.

Вот такое приближенное изображение РАСШИРЕНИЯ в структуре БД. Так же если, дать запрос вот такого вида, то можно получить характеристику расширений для пользователя **MILLER**. Здесь к стати сразу ясно, что такое **SEGMENT**! Именно то, что я и говорил раньше:

```
SQL> SELECT a.OWNER, a.SEGMENT_NAME, a.SEGMENT_TYPE, a.TABLESPACE_NAME,  
2 a.EXTENT_ID, a.BLOCK_ID, a.BYTES  
3 FROM DBA_EXTENTS a  
4 WHERE OWNER = 'MILLER'
```


5 /

OWNER BYTES	SEGMENT_NAME	SEGMENT_TYPE	TABLESPACE_NAME	EXTENT_ID	BLOCK_ID
MILLER	CUSTOMERS	TABLE	USERS	0	837 81920
MILLER	OFFICES	TABLE	USERS	0	997 81920
MILLER	ORDERS	TABLE	USERS	0	1077 81920
MILLER	PRODUCTS	TABLE	USERS	0	1157 81920
MILLER	SALESREPS	TABLE	USERS	0	1237 81920
MILLER	SYS_C001257	INDEX	USERS	0	797 81920
MILLER	SYS_C001258	INDEX	USERS	0	877 81920
MILLER	SYS_C001259	INDEX	USERS	0	957 81920
MILLER	SYS_C001263	INDEX	USERS	0	1037 81920
MILLER	SYS_C001264	INDEX	USERS	0	1117 81920

10 rows selected

Пока с этим все. В следующий раз займемся в плотную сегментами БД.

Шаг 24 - Основные компоненты БД Oracle - Сегменты БД - Таблицы

Давайте рассмотрим, что же такое сегменты БД. Начнем с самого основного и наиболее используемого собственно к самой БД, а именно, ТАБЛИЦЫ. Одним из основных сегментов БД, а я считаю, что и есть то ради чего, собственно, и задумывались такие понятия как База Данных. Таблица в БД, представляет из, себя и обязательно включает следующие элементы:

1. Имя таблицы.
2. **n**-е количество столбцов, с уникальными именами. (не стоит использовать в имени столбца слова типа - **NUMBER**, **VARCHAR** и т.д. так как собственно, это ключевые слова **PL/SQL**)
3. **n**-е количество, записей, т.е. собственно сами данные, которые хранятся, в таблице, в виде (по аналогии с языками программирования) двухмерного, не выровненного массива! Если хотите, можете со мной поспорить! :)

Основные типы данных, столбцов таблицы БД, я вам уже приводил в "[Шаг 7 - Язык SQL и примитивные типы данных](#)", так что загляните туда. Так же замечу, что мы еще остановимся на типах данных, когда в плотную подойдем к **PL/SQL**. Пока я пытаюсь создать для вас целостную картину БД, так как если не разъяснить немного эти основные понятия, то **PL/SQL** будет трудно освоить! Чего-то я отвлекся! :) Итак немного забегаю вперед, разберем это понятие чуть ближе!

Создание таблицы.

Создадим в нашей схеме **MILLER**, которую мы уже имеем и работаем с ней, пробную табличку и разберемся как это все работает. Итак, запускайте **SQL/Plus**, создаем таблицу с именем, скажем, **PEOPLE**. Для этого, необходимо отправить в **SQL/Plus** команду **CREATE TABLE**:

```
CREATE TABLE PEOPLE -- команда создания таблицы -----*
( ID NUMBER,         -- числовое поле -----*
  NM VARCHAR2(50),   -- символьное поле -----*
  FAMIL VARCHAR2(50),
  OTCH VARCHAR2(50),
  DROG DATE          -- поле даты -----*
)
/
```

```
SQL> CREATE TABLE PEOPLE
2 ( ID NUMBER,
3   NM VARCHAR2(50),
4   FAMIL VARCHAR2(50),
5   OTCH VARCHAR2(50),
6   DROG DATE
7 )
8 /
```

Table created

Табличка успешно создана! После этого не забудьте ввести команду **COMMIT**! Иначе после завершения сеанса, можете не найти вашу табличку! Вот так:

```
COMMIT
```

```
/
SQL> COMMIT
2 /
```

Commit complete

Это означает, что "транзакция", то есть ваши осмысленные действия БД приняла и подтвердила. Теперь давайте убедимся, что наша таблица действительно создана и появилась успешно в схеме **MILLER**. Отправьте следующий запрос к представлению **USER_TABLES**, которое содержит информацию обо всех таблицах данной схемы:

```
SELECT a.TABLE_NAME, a.TABLESPACE_NAME, a.NUM_ROWS FROM USER_TABLES a
WHERE a.TABLE_NAME = 'PEOPLE'
/
```

Получаем следующее:

```
SQL> SELECT a.TABLE_NAME, a.TABLESPACE_NAME, a.NUM_ROWS FROM USER_TABLES a
2 WHERE a.TABLE_NAME = 'PEOPLE'
3 /
```

TABLE_NAME	TABLESPACE_NAME	NUM_ROWS
PEOPLE	USERS	

ДА! Таблица создана, располагается в табличном пространстве **USERS**, и имеет ноль записей. Давайте дадим еще вот такой запрос к представлению **USER_TAB_COLUMNS**, которое содержит информацию обо всех столбцах всех таблиц данной схемы:

```
SELECT a.TABLE_NAME, a.COLUMN_NAME, a.DATA_TYPE, a.DATA_LENGTH FROM
USER_TAB_COLUMNS a
WHERE a.TABLE_NAME = 'PEOPLE'
/
```

Получаем примерно следующее:

```
SQL> SELECT a.TABLE_NAME, a.COLUMN_NAME, a.DATA_TYPE, a.DATA_LENGTH FROM
USER_TAB_COLUMNS a
2 WHERE a.TABLE_NAME = 'PEOPLE'
3 /
```

TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_LENGTH
PEOPLE	ID	NUMBER	22
PEOPLE	NM	VARCHAR2	50
PEOPLE	FAMIL	VARCHAR2	50
PEOPLE	OTCH	VARCHAR2	50
PEOPLE	DROG	DATE	7

Вот теперь видно, что из себя представляет наша таблица в схеме! Полностью соответствует тому что мы с вами и проделывали. Замечательно, но кому нужна пустая таблица? Давайте

введем в нее немного данных! И посмотрим, что из этого получится! Ввести данные в таблицу нам поможет, нет не киножурнал "Хочу все знать", а оператор **INSERT**. Итак, введите следующее:

```
INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
      VALUES(1, 'John', 'Godwin', 'Petrovich', TO_DATE('03-12-1967','DD-MM-YYYY'))
/

INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
      VALUES(2, 'Bob', 'Doris', 'Martovich', TO_DATE('01-02-1960','DD-MM-YYYY'))
/

INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
      VALUES(3, 'Frank', 'Black', 'Milleniumich', TO_DATE('03-07-1953','DD-MM-YYYY'))
/

COMMIT
/
```

После отправки команд увидите, что каждая запись отапортовала о своем успешном введении и все изменения в таблице подтверждены!

```
SQL> INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
  2      VALUES(1, 'John', 'Godwin', 'Petrovich', TO_DATE('03-12-1967','DD-MM-
  3      YYYY'))
  3 /

1 row inserted

SQL> INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
  2      VALUES(2, 'Bob', 'Doris', 'Martovich', TO_DATE('01-02-1960','DD-MM-
  3      YYYY'))
  3 /

1 row inserted

SQL> INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
  2      VALUES(3, 'Frank', 'Black', 'Milleniumich', TO_DATE('03-07-1953','DD-
  3      MM-YYYY'))
  3 /

1 row inserted

SQL> COMMIT
  2 /

Commit complete
```

А затем убедимся, что в таблице появились значения дав следующий запрос:

```
SELECT * FROM PEOPLE
/
```

Теперь, видно, что таблица имеет три записи:

```
SQL> SELECT * FROM PEOPLE  
2 /
```

ID	NM	FAMIL	OTCH	DROG
1	Bob	Doris	Martovich	01.02.1960
1	Frank	Black	Milleniumich	03.07.1953
1	John	Godwin	Petrovich	03.12.1967

Так же еще добавлю, что информация обо всех таблицах содержится в представлении **DBA_TABLES**, а обо всех столбцах всех таблиц БД в представлении **DBA_TAB_COLUMNS**. Пока на этом остановимся, дальше еще много объектов, с которыми необходимо познакомиться. А пока, можете поработать с созданием табличек! :)

Шаг 25 - Основные компоненты БД Oracle - Сегменты БД, продолжаем

Теперь, рассмотрим сегмент ИНДЕКС таблиц БД. Индексы необходимы для ускорения поиска данных в таблицах. В индексе хранятся значения одного или нескольких столбцов, а так же столбец **ROWID** для каждого из хранимых значений. При поиске в таблице используется значение **ROWID** конкретной записи, и она возвращается непосредственно. В БД **Oracle** используется несколько типов индексов. Один из них называется **B*-Tree-Index** (двоичный древовидный индекс). Индексы такого типа формируются командой **CREATE INDEX**. Показывать как это делается я пока не буду, к этому мы подойдем позже, а пока просто разберем типы этих сегментов БД. Существует так же "кластерный индекс", он используется при построении такого типа данных как "кластеры таблиц". Существуют так же **bitmap** индексы, которые формируются по битовой маске столбцов таблиц. Но в основном вам предстоит работать с **B*-Tree-Index**-ми, которых для начала вполне хватит. Добавлю, что данные обо всех индексах БД, а так же индексируемых столбцах БД можно найти в представлениях **DBA_INDEXES**, **DBA_IND_COLUMNS**. К стати введите в **SQL*Plus** команду **DESC DBA_INDEXES**, **DESC DBA_IND_COLUMNS**. Могу вас уверить, получите много полезной информации из столбца **Comments** описания! :) К стати очень полезная команда!

Следующий сегмент БД называется СЕГМЕНТ ОТКАТА. У него даже название подходящее, само собой напрашивается. Что это такое? Сегмент отката, это если выразиться правильным языком, "элементы информационной структуры БД", которые сохраняют информацию о том, что было в блоках данных до того как их начали изменять. Или говоря другими словами, стартовала транзакция. Это значит, что данные в БД начали изменяться. Как только этот процесс начинается, все, что было ранее, скажем в таблице, записывается, в сегмент отката (**rollback segments**). Далее транзакция начинает свое черное дело, и вдруг ей понадобились данные, которые сейчас находятся в сегменте отката, они оттуда легко извлекаются! Такая операция носит название "плотное чтение" (**consistent read**). После того, как вы подтвердили транзакцию, введя оператор **COMMIT**, все данные в сегменте отката помечаются как не действительные, и как говорится, "Шура, пилите гирю! - Она золотая!!!". Если нужно все вернуть на круги своя, оборвав действия текущей транзакции, введите оператор **ROLLBACK**. Все останется как было! Кстати, при работе сегмент отката присоединяет как минимум два расширения, и как следствие нужно сделать так, чтобы ему хватило текущих расширений для работы. Иначе, он потребует еще и вся операция может затянуться или привести к аварийному откату транзакции, если места в текущем табличном пространстве не хватит для работы сегмента отката! Понятно, что я здесь излагаю. Это нужно хорошо представлять, иначе можно лишиться себя любимого важных данных! А правильная настройка сегментов отката и табличных пространств поможет вам избежать неприятных моментов!

Самое интересное, СЛОВАРИ ДАННЫХ БД. В словарях данных хранится просто огромное количество информации о БД, ее пользователях, объектах и т.д.!!! Они применяются администратором БД для правильной настройки и эксплуатации БД. Обеспечение ее бесперебойной работы 24 часа в сутки, 365 дней в году! :) Словари данных доступны пользователям БД, **SYS** и **SYSTEM**, а так же пользователям имеющим привилегии на их использование. Доступ к словарям идет только в режиме чтения!!! Это важно!!! Не пытайтесь что-либо поменять в словарях вручную, последствия будут самые не предсказуемые!!! :) Словари состоят из 4-х компонентов:

1. **X\$**- таблиц, внутренние таблицы БД.
2. **\$**- Таблицы словаря данных
3. **V\$**- таблиц представления текущей активности.
4. Представления словаря.

X\$- таблицы используются самой БД, их содержимое зашифровано и извлекать из них данные нет особой необходимости! **\$**- Таблицы словаря данных формируются при создании БД файлом **SQL.BSQ** и принадлежат пользователю **SYS**. Содержит информацию обо всех конструкциях БД. **V\$**- таблицы представления текущей активности, являются основным инструментом администратора БД! С их помощью можно вести полный мониторинг БД. Принадлежат так же пользователю **SYS**. Представления словаря формируются из **X\$**- таблиц и имеют три основных типа:

1. **ALL_**
2. **DBA_**
3. **USER_**

В них соответственно можно найти информацию. В **ALL_** обо всех объектах БД и пользователей, **DBA_** принадлежит администратору и используется им. **USER_** все объекты конкретной схемы пользователя. Вот кратенько о сегментах "словари данных". Но, я так же советую дать такой запрос, войдя в БД пользователем **SYSTEM**:

```
SELECT * FROM DICTIONARY
/
```

Например, я получил 817(!) сток! Здесь, можно найти краткое описание каждого словаря данных! Вот вам кусочек, запроса для примера:

```
SQL> SELECT * FROM DICTIONARY
2 /
```

TABLE_NAME	COMMENTS
ALL_ALL_TABLES	Description of all object and relational tables accessible to the user
ALL_ARGUMENTS	Arguments in object accessible to the user
ALL_CATALOG	All tables, views, synonyms, sequences accessible to the user
ALL_CLUSTERS	Description of clusters accessible to the user
ALL_CLUSTER_HASH_EXPRESSIONS	Hash functions for all accessible clusters
ALL_COLL_TYPES	Description of named collection types accessible to the user
ALL_COL_COMMENTS	Comments on columns of accessible tables and views
ALL_COL_PRIVS	Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
ALL_COL_PRIVS_MADE	Grants on columns for which the user is owner or grantor
ALL_COL_PRIVS_RECD	Grants on columns for which the user, PUBLIC or enabled role is the grantee
ALL_CONSTRAINTS	Constraint definitions on accessible tables
ALL_CONS_COLUMNS	Information about accessible columns in constraint definitions
ALL_CONTEXT	Description of all active context namespaces under the current session
ALL_DB_LINKS	Database links accessible to the user
ALL_DEF_AUDIT_OPTS	Auditing options for newly created objects
ALL_DEPENDENCIES	Dependencies to and from objects accessible to the user
ALL_DIMENSIONS	Description of the dimension objects accessible to the DBA
ALL_DIM_ATTRIBUTES	Representation of the relationship between a dimension level and a functionally dependent column
.	.
.	.

Достаточно хорошо видно, что можно найти как имя нужного словаря, так и то, что он содержит, если еще подкрепить это все командой **DESC**, то я думаю, что скоро родится еще 10, а может 20 администраторов БД! :) Естественно, желательно немного знать **English**. И дело я думаю, пойдет!

Шаг 26 - Снова SELECT - многотабличные запросы

Что ж, мы с вами немного познакомились с объектами **Oracle**, успели кое-что разобрать и понять. Теперь давайте вернемся к оператору **SELECT**, так как мы в [шаге 19](#) закончили однотабличные запросы, осталось еще кое-что, а именно многотабличные запросы к БД. Итак, посмотрим на рисунок 1.

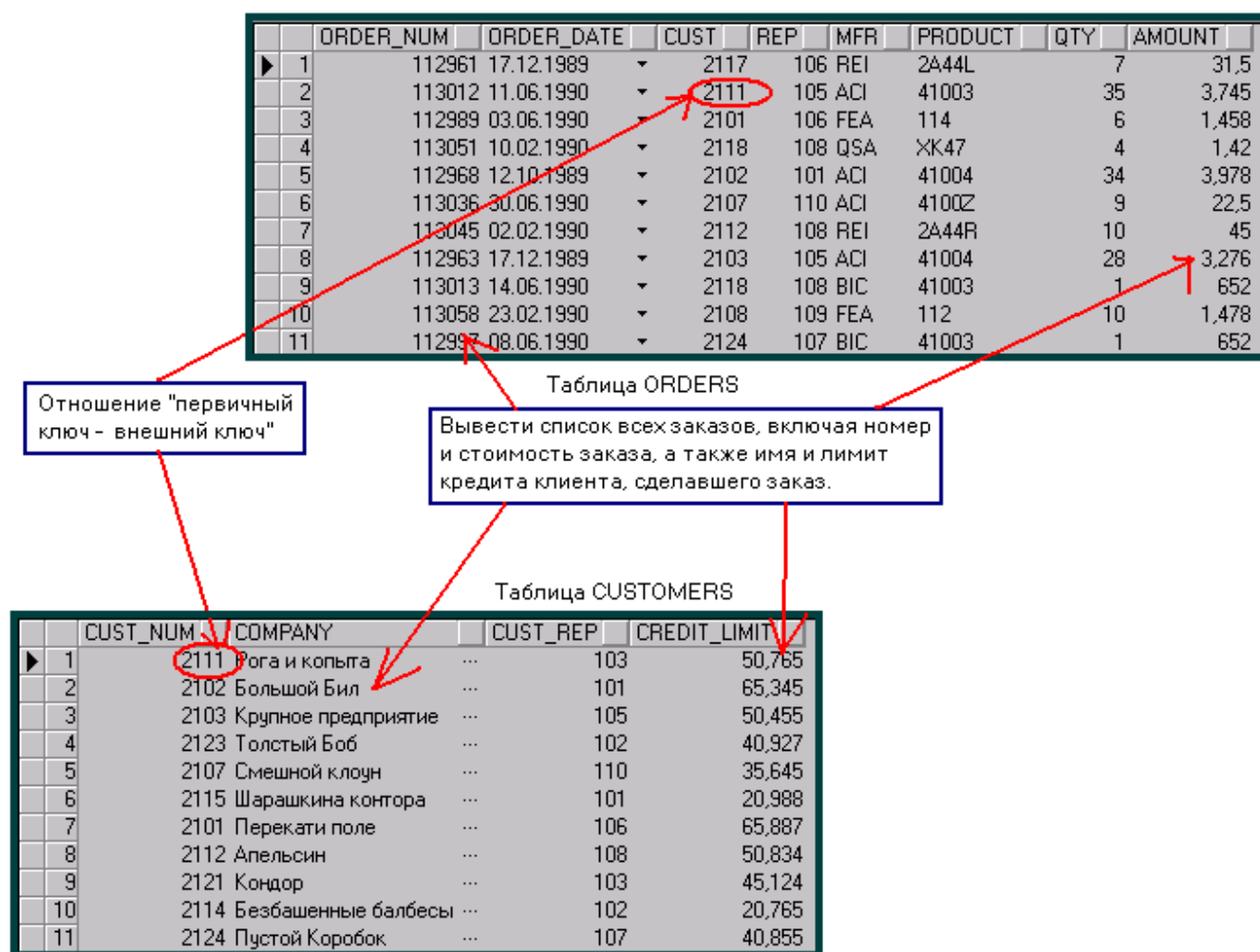


Рис. 1 Запрос охватывающий две таблицы

Здесь хорошо видно как строится многотабличный запрос. Я попытался изобразить как можно понятнее. Имеется две таблицы, **ORDERS** и **CUSTOMERS**. Формулировка запроса звучит следующим образом. Вывести список всех заказов, включая номер и стоимость заказа, а так же имя клиента и лимит кредита. Как видно, информация, которую необходимо выбрать находится в двух таблицах. Как же произвести выборку из двух таблиц сразу? Достаточно просто! Необходимо ввести запрос следующего вида:

```
SQL> SELECT ORDER_NUM, AMOUNT, COMPANY, CREDIT_LIMIT
2 FROM ORDERS, CUSTOMERS
3 WHERE CUST = CUST_NUM
```

4 / ORDER_NUM	AMOUNT COMPANY	A CREDIT_LIMIT
112961	31,5 Сметенные огнем	35,324
113012	3,745 Рога и копыта	50,765
112989	1,458 Перекати поле	65,887
113051	1,42 Просто Балбесы	60,653
112968	3,978 Большой Бил	65,345
113036	22,5 Смешной клоун	35,645
113045	45 Апельсин	50,834
112963	3,276 Крупное предприятие	50,455
113013	652 Просто Балбесы	60,653
113058	1,478 Унесенные ветром	55,323
112997	652 Пустой Коробок	40,855
112983	702 Крупное предприятие	50,455
113024	7,1 Безбашенные балбесы	20,765
113062	2,43 Пустой Коробок	40,855
112979	15 Безбашенные балбесы	20,765
113027	4,104 Крупное предприятие	50,455
113007	2,925 Апельсин	50,834
113069	31,35 Молочная компания	25,634
113034	632 Смешной клоун	35,645
112922	760 Просто Балбесы	60,653
112975	2,1 Рога и копыта	50,765
113055	150 Унесенные ветром	55,323
113048	3,75 Зашаренные	50,126
112993	1,896 Злыдень Карпорейтед	65,902
113065	2,13 Злыдень Карпорейтед	65,902
113003	5,625 Унесенные ветром	55,323
113049	710 Просто Балбесы	60,653
112987	27,5 Крупное предприятие	50,455
113057	600,34 Рога и копыта	50,765
113042	22,5 Волопас супермаркет	20,923

30 rows selected

Данные получены! Сам запрос, вроде бы не отличается от тех, которые мы строили раньше. Но есть одно отличие, оператор **SELECT** в предложении **FROM** содержит не одну, а две таблицы и предложение **WHERE** сравнивает два столбца из двух разных таблиц, а именно **ORDERS.CUST** и **CUSTOMERS.CUST_NUM**! Вот это и называется, если говорить языком формулировок "объединение двух таблиц по равенству". Надеюсь, все очень хорошо видно из рисунка 1.

Идем дальше, в основном, почти все многотабличные запросы, это запросы типа - предок/потомок. Такие запросы построены на том, что у одной таблицы имеется первичный ключ, а у другой внешний (вторичный) ключ. Сравнивая значение первичного ключа с внешним ключом получаем объединение таблиц по принципу предок/потомок. Посмотрим на Рисунок 2:

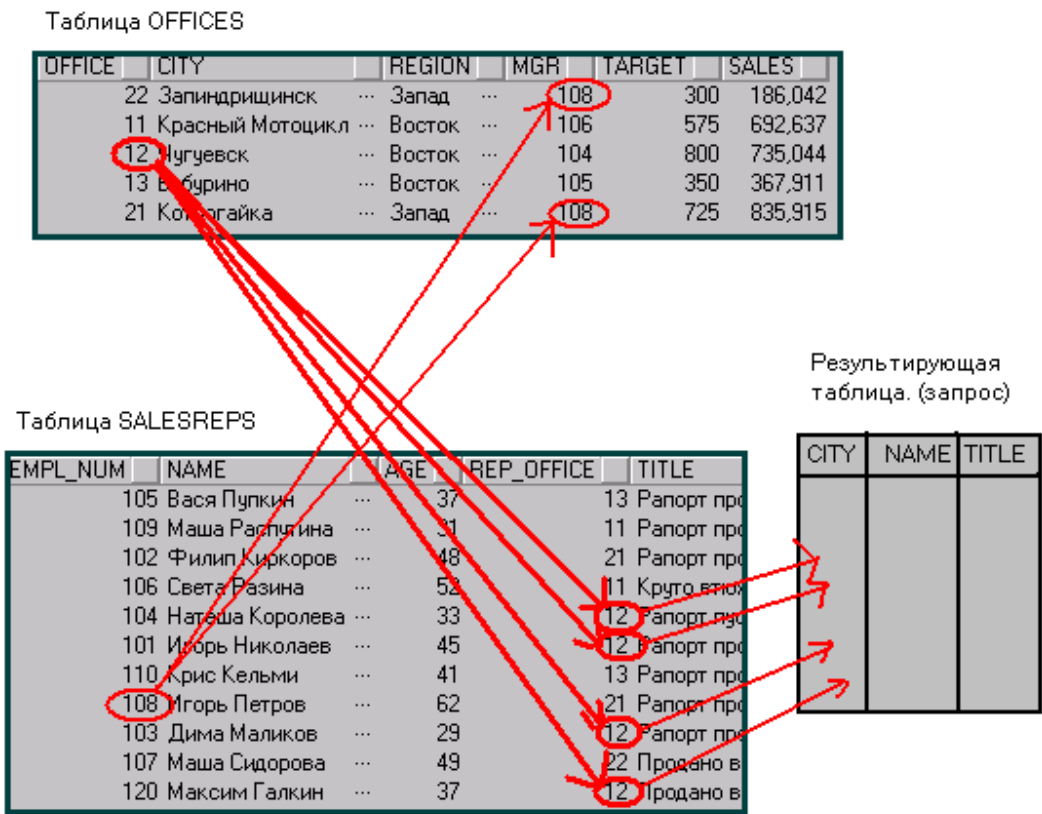


Рис. 2 Запрос с использованием отношения предок/потомок между таблицами OFFICES и SALESREPS

Здесь хорошо видно, что первичный ключ таблицы **OFFICES** (столбец **OFFICE**) является предком для внешнего ключа таблицы потомка **SALESREPS** (столбец **REP_OFFICE**). Здесь хорошо видно, что значение 12 из предка объемлют четыре значения в таблице потомке. Такие отношения таблиц имеют еще понятие ссылочной целостности и нормализации таблиц, но этого мы еще коснемся позже. А сейчас давайте сформулируем запрос на объединение двух таблиц с использованием отношения предок/потомок. Итак, вывести список всех служащих, включая города и регионы, в которых они работают.

```
SQL> SELECT NAME, CITY, REGION
2 FROM SALESREPS, OFFICES
3 WHERE REP_OFFICE = OFFICE
4 /
```

NAME	CITY	REGION
Вася Пупкин	Бубурино	Восток
Маша Распутина	Красный Мотоцикл	Восток
Филип Киркоров	Котрогайка	Запад
Света Разина	Красный Мотоцикл	Восток
Наташа Королева	Чугуевск	Восток
Игорь Николаев	Чугуевск	Восток
Крис Кельми	Бубурино	Восток
Игорь Петров	Котрогайка	Запад

Дима Маликов	Чугуевск	Восток
Маша Сидорова	Запіндрищинск	Запад
Максим Галкин	Чугуевск	Восток

11 rows selected

Получаем результирующий запрос(таблицу) с объединением по равенству с использованием отношения предок/потомок. Здесь, сами значения ключей мы не выводим вследствие того, что для результата запроса они не наглядны. Гораздо понятнее названия городов и регионов, тем более, что запоминать цифры, которые им соответствуют нет смысла! :) Как видно, для этих двух таблиц можно сформировать запрос где они поменяются ролями! Используя то же отношение предок потомок, но для столбцов **MGR** и **EMPL_NUM**. Дадим такой запрос - вывести список офисов, включая их имена и рапорты! :)

```
SQL> SELECT CITY, NAME, TITLE
2 FROM OFFICES, SALESREPS
3 WHERE MGR = EMPL_NUM
4 /
```

CITY	NAME	TITLE
Запіндрищинск	Игорь Петров	Рапорт продажа
Красный Мотоцикл	Света Разина	Круто втюхал
Чугуевск	Наташа Королева	Рапорт пусто
Бубурино	Вася Пупкин	Рапорт продажа
Котрогайка	Игорь Петров	Рапорт продажа

Получаем результат, только теперь таблицы поменялись ролями! А в остальном, все практически то же самое, что и в предыдущем запросе! Так же при объединении таблиц можно применять все стандартные условия поиска. Например, изменим предыдущий запрос на такое условие - вывести список офисов, включая их имена и рапорты, а так же план продаж, которых превышает \$600.00:

```
SQL> SELECT CITY, NAME, TITLE
2 FROM OFFICES, SALESREPS
3 WHERE MGR = EMPL_NUM AND
4 TARGET > 600.00
5 /
```

CITY	NAME	TITLE
Чугуевск	Наташа Королева	Рапорт пусто
Котрогайка	Игорь Петров	Рапорт продажа

2 rows selected

Получаем, только две записи, но уже видно, что работает условие поиска. Так же имеется возможность связывать таблицы, когда имеется более чем два связанных столбца в таблицах. Таким образом, к таблицам **ORDERS** и **PRODUCTS**, можно дать запрос следующего вида - вывести список всех заказов, в том числе и их стоимость и описание товаров:

```
SQL> SELECT ORDER_NUM, AMOUNT, DESCRIPTION
2 FROM ORDERS, PRODUCTS
```



```
3 WHERE MFR = MFR_ID AND PRODUCT = PRODUCT_ID
```

```
4 /
```

ORDER_NUM	AMOUNT	DESCRIPTION
112961	31,5	Доска профильная
113012	3,745	Рейка деревянная
112989	1,458	Электродвигатель
113051	1,42	Труба алюминиевая
112968	3,978	Рейка пластмассовая
113036	22,5	Монитор LG
113045	45	Телевизор SAMSUNG
112963	3,276	Рейка пластмассовая
113013	652	Стекломаст рулоны
113058	1,478	Стол офисный
112997	652	Стекломаст рулоны
112983	702	Рейка пластмассовая
113024	7,1	Труба алюминиевая
113062	2,43	Электродвигатель
112979	15	Монитор LG
113027	4,104	Носки черные
113007	2,925	Наушники SONY
113069	31,35	Осветитель ртутный
113034	632	Бочка металлическая
112922	760	Носки черные
112975	2,1	Бочка пластмассовая
113055	150	Карандаш простой
113048	3,75	Профиль специальный
112993	1,896	Бочка металлическая
113065	2,13	Труба алюминиевая
113003	5,625	Профиль специальный
113049	710	Труба алюминиевая
112987	27,5	Коробка картонная
113057	600,34	Карандаш простой
113042	22,5	Телевизор SAMSUNG

30 rows selected

Здесь, хорошо видно, что связь предок/потомок, построена по значениям двух пар столбцов данных таблиц, а именно **MFR**, **MFR_ID** и **PRODUCT**, **PRODUCT_ID**. Таким, образом, можно составлять достаточно гибкую логику в запросах к двум или более таблицам. Пока с этим все, продолжим в следующий раз.

Шаг 27 - Снова SELECT - Таблиц становится больше!

Итак, одну и две таблицы мы опрашивали, а что, если их будет больше? Да, в общем-то можно и больше, хотя я в своей практике более трех таблиц в запросе объединяю редко! :) Нет пока особой нужды, но знать как это делается не помешает. Итак, как обычно записываем наши формулировки запросов и смотрим и разбираем, что же там происходит! :)

Вывести список заказов стоимостью выше \$700, включая имя служащего, принявшего заказ и имя клиента сделавшего его.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
FROM ORDERS, CUSTOMERS, SALESREPS
WHERE CUST = CUST_NUM AND
      REP = EMPL_NUM AND
      AMOUNT > 700
/
```

Получаем следующее:

```
SQL> SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
2      FROM ORDERS, CUSTOMERS, SALESREPS
3 WHERE CUST = CUST_NUM AND
4      REP = EMPL_NUM AND
5      AMOUNT > 700
6 /
```

ORDER_NUM	AMOUNT	COMPANY	NAME
112983	702	Крупное предприятие	Вася Пупкин
112922	760	Просто Балбесы	Игорь Петров
113049	710	Просто Балбесы	Игорь Петров

3 rows selected

А, теперь посмотрим на рисунок:

Таблица CUSTOMERS

	CUST_NUM	COMPANY	CUS
1	2111	Воса и копыта	...
2	2102	Большой Бил	...
3	2103	Крупное предприятие	...
4	2123	Толстый Боб	...
5	2107	Смешной клоун	...
6	2115	Шарашкина контора	...
7	2101	Перекасти поле	...

Таблица SALESREPS

	EMPL_NUM	NAME	AGE	REP_OFFI
1	105	Вася Пупкин	...	37
2	109	Маша Распутина	...	31
3	102	Филип Киркоров	...	48
4	106	Света Разина	...	52
5	104	Наташа Королева	...	33
6	101	Игорь Николаев	...	45
7	110	Крис Кельми	...	41

Таблица ORDERS

ORDER_NUM	ORDER_DATE	CUST	REP	MFR
112961	17.12.1989	2117	105	REI
113012	11.06.1990	2111	105	ACI
112989	03.06.1990	2101	106	FEA
113051	10.02.1990	2118	108	QSA
112968	12.10.1989	2102	101	ACI
113036	30.06.1990	2107	110	ACI

Результаты запроса			
ORDER_NUM	AMOUNT	COMPANY	NAME

Рис. 1 Запрос к трем таблицам

Из рисунка видно, что запрос построен по равенству столбцов и использует в запросе три(!) таблицы. Как видно, столбец **CUST** таблицы **ORDERS** является внешним ключом для таблицы **CUSTOMERS**, а столбец **REP** той же таблицы **ORDERS** является внешним ключом для таблицы **SALESREPS**! Иначе говоря, запрос связывает каждый заказ с соответствующим клиентом и служащим. Вот так и получился, запрос к трем таблицам.

Давайте для более полного, понимания, выполним еще вот такой запрос: Вывести список заказов стоимостью выше \$700, включая имя клиента разместившего заказ и имя служащего закрепленного за этим клиентом.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
FROM ORDERS, CUSTOMERS, SALESREPS
WHERE CUST = CUST_NUM AND
      CUST_REP = EMPL_NUM AND
      AMOUNT > 700
/
```

Соответственно получаем:

```
SQL> SELECT ORDER_NUM, AMOUNT, COMPANY, NAME
2      FROM ORDERS, CUSTOMERS, SALESREPS
3 WHERE CUST = CUST_NUM AND
4      CUST_REP = EMPL_NUM AND
5      AMOUNT > 700
6 /
```

ORDER_NUM	AMOUNT	COMPANY	NAME
112983	702	Крупное предприятие	Вася Пупкин

112922	760 Просто Балбесы	Игорь Петров
113049	710 Просто Балбесы	Игорь Петров

3 rows selected

Как-то странно, но предыдущий запрос дал тот же результат! Да и бог с ним, здесь поменялись роли столбцов. Теперь столбец **CUST** работает как внешний ключ для таблицы **CUSTOMERS**, а столбец **CUST_REP** срабатывает внешним ключом для таблицы **SALESREPS**. Теперь логика запроса несколько поменялась. Данный запрос связывает каждый заказ с клиентом, а каждого клиента с закрепленным за ним служащим. Надеюсь, что не слишком запутал вас и разобраться как все это работает в общем-то не сложно! :)

Так же в промышленных масштабах иногда выполняют запросы к трем и более таблицам, вот где нужно иметь полное представление о связях в таблицах БД! И мы с вами можем проделать нечто подобное на нашей учебной БД.

Давайте попробуем, дать вот такой запрос: Вывести список заказов стоимостью выше \$700, включая имя клиента, разместившего заказ и имя служащего, закрепленного за этим клиентом, а так же офис, в котором работает этот служащий.

```
SELECT ORDER_NUM, AMOUNT, COMPANY, NAME, CITY
FROM ORDERS, CUSTOMERS, SALESREPS, OFFICES
WHERE CUST = CUST_NUM AND
      CUST_REP = EMPL_NUM AND
      REP_OFFICE = OFFICE AND
      AMOUNT > 700
/
```

Четыре(!) таблицы! Получаем:

```
SQL> SELECT ORDER_NUM, AMOUNT, COMPANY, NAME, CITY
2      FROM ORDERS, CUSTOMERS, SALESREPS, OFFICES
3 WHERE CUST = CUST_NUM AND
4      CUST_REP = EMPL_NUM AND
5      REP_OFFICE = OFFICE AND
6      AMOUNT > 700
7 /
```

ORDER_NUM	AMOUNT	COMPANY	NAME	CITY
112983	702	Крупное предприятие	Вася Пупкин	Бубурино
112922	760	Просто Балбесы	Игорь Петров	Котрогайка
113049	710	Просто Балбесы	Игорь Петров	Котрогайка

3 rows selected

Здесь так же используется связь предок/потомок. Теперь мы связали заказ с клиентом клиента - с закрепленным за ним служащим, а служащего с его офисом! Вот так связывается хоть сотня таблиц, главное не получить мозговую травму! :) Пробуйте!

Шаг 28 - Снова SELECT - Объединения таблиц

Итак, отношение предок/потомок это в сущности отношение "один ко многим", но строить условия поиска только по этому принципу вовсе не обязательно. Вот, например, такой запрос по датам:

```
-- // -----  
Найти все заказы, полученные в тот день, когда на работу был принят новый  
служащий.  
  
SELECT ORDER_NUM, AMOUNT, ORDER_DATE, NAME  
FROM ORDERS, SALESREPS  
WHERE ORDER_DATE = HIRE_DATE  
/
```

Получаем 6 строк:

```
SQL> SELECT ORDER_NUM, AMOUNT, ORDER_DATE, NAME  
2      FROM ORDERS, SALESREPS  
3 WHERE ORDER_DATE = HIRE_DATE  
4 /
```

ORDER_NUM	AMOUNT	ORDER_DATE	NAME
112968	3,978	12.10.1989	Маша Распутина
112979	15	12.10.1989	Маша Распутина
112975	2,1	12.10.1989	Маша Распутина
112968	3,978	12.10.1989	Максим Галкин
112979	15	12.10.1989	Максим Галкин
112975	2,1	12.10.1989	Максим Галкин

6 rows selected

Это пары строк из таблиц **ORDERS** и **SALESREPS** имеющие одинаковое значение в столбцах **ORDER_DATE** и **HIRE_DATE**.

Такая связь рождает отношение "многие ко многим". Что здесь собственно происходит, да просто в те дни, когда Маша Распутина и Максим Галкин были приняты на работу, было сделано три заказа соответственно, каждое, совпадение дат и дает три строки! Можно было сделать и на оборот, какие служащие были приняты в те дни когда были сделаны заказы... и так далее! Это в общем отвлеченные понятия, но оператор **SELECT** сработал безукоризненно. Итак, давайте сделаем небольшой итог:

1. В объединении предок/потомок работает связь "один ко многим".
2. В других объединениях, так же может существовать отношение "один ко многим", если по крайней мере в одной таблице связанный столбец содержит уникальные значения во всех строках.
3. В объединениях, созданных на произвольном, связывании столбцов, существует отношение "многие ко многим".

Так же давайте рассмотрим объединение таблиц по неравенству. Например, вот такой запрос: Получить все комбинации служащих офисов, где плановый объем продаж служащего больше, чем план какого либо офиса.

```
SELECT NAME, QUOTA, CITY, TARGET
FROM SALESREPS, OFFICES
WHERE QUOTA > TARGET
/
```

Соответственно получаем:

```
SQL> SELECT NAME, QUOTA, CITY, TARGET
2      FROM SALESREPS, OFFICES
3 WHERE QUOTA > TARGET
4 /
```

NAME	QUOTA CITY	TARGET
Вася Пупкин	350 Запиндрищинск	300
Филип Киркоров	350 Запиндрищинск	300
Крис Кельми	400 Запиндрищинск	300
Игорь Петров	350 Запиндрищинск	300
Максим Галкин	400 Запиндрищинск	300
Крис Кельми	400 Бубурино	350
Максим Галкин	400 Бубурино	350

7 rows selected

Условие поиска **QUOTA > TARGET** отбирает пары, в которых значение столбца **QUOTA** из таблицы **SALESREPS**, превышает значение столбца **TARGET** из таблицы **OFFICES**. Обратите внимание на то, что выбранные столбцы связаны только таким образом! Данный пример является несколько надуманным, но все же показывает как различные условия поиска легко применимы в приложениях, например, для систем принятия решений, а так же исследующих более сложные взаимосвязи в БД. К стати, особенно когда дается запрос к 5ти и более таблицам начинаешь понимать, что нифига не понимаешь! :) Можете попробовать сами!

Шаг 29 - Снова SELECT - имена, псевдонимы...

Итак, продолжим, в нашей учебной базе данных, вернее сказать, в учебных табличках, есть две таблицы с одинаковыми именами столбцов. Это таблицы **OFFICES** и **SALESREPS**. В них обеих есть столбец с именем **SALES**. Что если по какой-либо причине нам понадобится дать, например, вот такой запрос: Показать имя, офис, и объем продаж каждого служащего.

Он будет выглядеть примерно так:

```
SELECT NAME, SALESREPS.SALES, CITY
FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE
/
```

После выполнения получаем ошибку **Oracle Server**:

```
SQL> SELECT NAME, SALES, CITY
2      FROM SALESREPS, OFFICES
3      WHERE REP_OFFICE = OFFICE
4 /
```

```
SELECT NAME, ->SALES, CITY
FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE
```

ORA-00918: столбец определен неоднозначно

Здесь я стрелочкой показал какой столбец вызвал эту ошибку. Если бы мы дали обычный запрос, например, вот такого вида: Показать названия городов, в которых фактический объем продаж превышает плановый.

```
SELECT CITY, SALES
FROM OFFICES
WHERE SALES > TARGET
/
```

Показать имена служащих, у которых объем продаж больше \$1000.

```
SELECT NAME, SALES
FROM SALESREPS
WHERE SALES > 200
/
```

Эти однотабличные запросы сработали бы без ошибок, но нас это на сегодня не устраивает. Так как же поступать? А очень просто, нужно применить так называемое "полное имя столбца", что это значит? А, то, что мы должны однозначно идентифицировать столбец согласно необходимой нам логике запроса!

Примерно вот так: Показать имя, офис, и объем продаж каждого служащего.


```

SELECT NAME, SALESREPS.SALES, CITY
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE
/

```

Соответственно получаем:

```

SQL> SELECT NAME, SALESREPS.SALES, CITY
      2      FROM SALESREPS, OFFICES
      3      WHERE REP_OFFICE = OFFICE
      4 /

```

NAME	SALES CITY
Вася Пупкин	367,911 Бубурино
Маша Распутина	392,725 Красный Мотоцикл
Филип Киркоров	474,05 Котрогайка
Света Разина	299,912 Красный Мотоцикл
Наташа Королева	142,594 Чугуевск
Игорь Николаев	305,673 Чугуевск
Крис Кельми	75,985 Бубурино
Игорь Петров	361,865 Котрогайка
Дима Маликов	286,775 Чугуевск
Маша Сидорова	186,042 Запндрищинск
Максим Галкин	386,042 Чугуевск

11 rows selected

Вот теперь запрос сработал, так как мы от него и ожидали! Правда, запись оператора **SELECT** стала немного длиннее. Но это не столь существенно. :) Теперь давайте рассмотрим еще один аспект при работе с многотабличными запросами, а именно оператор вида **SELECT ***, который используется для чтения всех столбцов. Рассмотрим запрос вида: Сообщить всю информацию, о служащих и офисах где они работают.

```

SELECT *
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE
/

```

После отправки его в **SQL*Plus** мы получим 14ть(!) столбцов, я не буду приводить результат запроса, так как это слишком громоздко, так что приведу только примерную конфигурацию вывода: Сообщить всю информацию, о служащих и офисах где они работают.

```

SELECT *
  FROM SALESREPS, OFFICES
 WHERE REP_OFFICE = OFFICE
/

```

EMPL_NUM	NAME	AGE	REP_OFFICE	TITLE	HIRE_DATE	MANAGER	QUOTA
.....							

```

105 Вася Пупкин      37      13 Рапорт продажа  12.02.1988      104      350 . . . . .
..
109 Маша Распутина  31      11 Рапорт продажа  12.10.1989      106      300 . . . . .
...
.
.
.

```

При работе с тремя и более таблицами оператор вида **SELECT *** не практичен, и его применение в многотабличных запросах можно поставить под сомнение. Однако, если его использовать несколько не стандартно, то в общем его можно и применять, но осторожно, так как в некоторых стандартах такие "фортелы" запрещены! :) Например, чтобы ограничить количество столбцов в предыдущем запросе можно записать: Сообщить всю информацию, о служащих и офисах где они работают.

```

SELECT SALESREPS.*, CITY, REGION
FROM SALESREPS, OFFICES
WHERE REP_OFFICE = OFFICE
/

```

```

SQL> SELECT SALESREPS.*, CITY, REGION
2      FROM SALESREPS, OFFICES
3      WHERE REP_OFFICE = OFFICE
4 /

```

Получим то же что и выше, но теперь столбцов будет одиннадцать, девять из таблицы **SALESREPS**, и два соответственно из таблицы **OFFICES**. Теперь делайте выводы относительно применять или не применять! :) Что ж, двигаемся дальше!

На очереди "**ПСЕВДОНИМЫ ТАБЛИЦ**".

Допустим, что у нас есть пользователь **SAM** с таблицей **BIRTHDAYS**, где содержатся дни рождения наших работников.

Тогда запрос вида:

```

SELECT SALESREPS.NAME, QUOTA, SAM.BIRTHDAYS.BIRTH_DAY
FROM SALESREPS, BIRTHDAYS
WHERE SALESREPS.NAME = SAM.BIRTHDAYS.NAME
/

```

Должен выполняться и вывести то, что мы ожидаем, но при написании мы видим, что получаются слишком длинные имена для столбцов и таблиц выборки. Вот здесь могут пригодиться "ПСЕВДОНИМЫ ТАБЛИЦ". Общий синтаксис записи псевдонимов строится следующим образом:

```

----- FROM ----- имя таблицы ----- ----- псевдоним таблицы -----,
-----

```

Перепишем наш предыдущий запрос используя псевдонимы таблиц:

```
SELECT S.NAME, S.QUOTA, B.BIRTHDAYS.BIRTH_DAY  
FROM SALESREPS S, SAM.BIRTHDAYS B  
WHERE S.NAME = B.NAME  
/
```

Теперь наглядно видно, что запись вида **SAM.BIRTHDAYS B**, где **B** и есть псевдоним таблицы, получается гораздо компактнее, и не нужно повторять сложные и трудно запоминаемые имена таблиц и столбцов. В некоторых **SQL** серверах допускается запись вида - **SAM.BIRTHDAYS AS B**, но в нашем случае достаточно записи **SAM.BIRTHDAYS B**. Вот теперь, надеюсь, понятно, как избежать излишней писанины с помощью псевдонимов таблиц. Можете пробовать сами для закрепления материала! :)

Шаг 30 - Снова SELECT - самообъединения, правила и т.д.

Разберем еще один интересный аспект "самообъединения". Некоторые запросы используют отношения существующие внутри одной таблицы. Например, требуется вывести список имен всех служащих и их руководителей. Вся эта информация содержится в одной таблице **SALESREPS**. Каждому служащему соответствует одна строка в таблице **SALESREPS**, а столбец **MANAGER** содержит идентификатор служащего, являющегося руководителем! Немного запутано? Но не так сложно, как может показаться. Просто столбец **MANAGER**, это внешний ключ для самой таблицы **SALESREPS**! Вот и все! Тогда, если пытаться создать запрос вида "первичный ключ - внешний ключ", можно было бы записать примерно так:

```
SELECT NAME, NAME
FROM SALESREPS, SALESREPS
WHERE MANAGER = EMPL_NUM
/
```

Но здесь есть некоторая глупость! Нельзя дважды ссылаться на одну и ту же таблицу в части **FROM**! Что ж, тогда можно было бы записать так:

```
SELECT NAME, NAME
FROM SALESREPS
WHERE MANAGER = EMPL_NUM
/
```

- "Постойте",- скажете вы, но это же однотабличный запрос! И будете правы, действительно он выродился до уровня простого выражения. Здесь получается по условию **MANAGER = EMPL_NUM** ищутся строки, в которых служащий является своим руководителем, но таких строк нет и по этому запрос ничего не вернет! А вот здесь и напрашивается использовать то, что мы рассматривали в [предыдущем шаге](#). Правильно псевдонимы таблиц! Давайте перепишем наш запрос, вот так:

```
SELECT EMPS.NAME, MGRS.NAME
FROM EMPS, MGRS
WHERE EMPS.MANAGER = MGRS.EMPL_NUM
/
```

Теперь все верно, только нужно добавить исходную таблицу для псевдонимов, вот так:

```
SELECT EMPS.NAME, MGRS.NAME
FROM SALESREPS EMPS, SALESREPS MGRS
WHERE EMPS.MANAGER = MGRS.EMPL_NUM
/
```

Вот собственно и получился запрос к двум таблицам, но с небольшим обманом, зато по всем правилам!

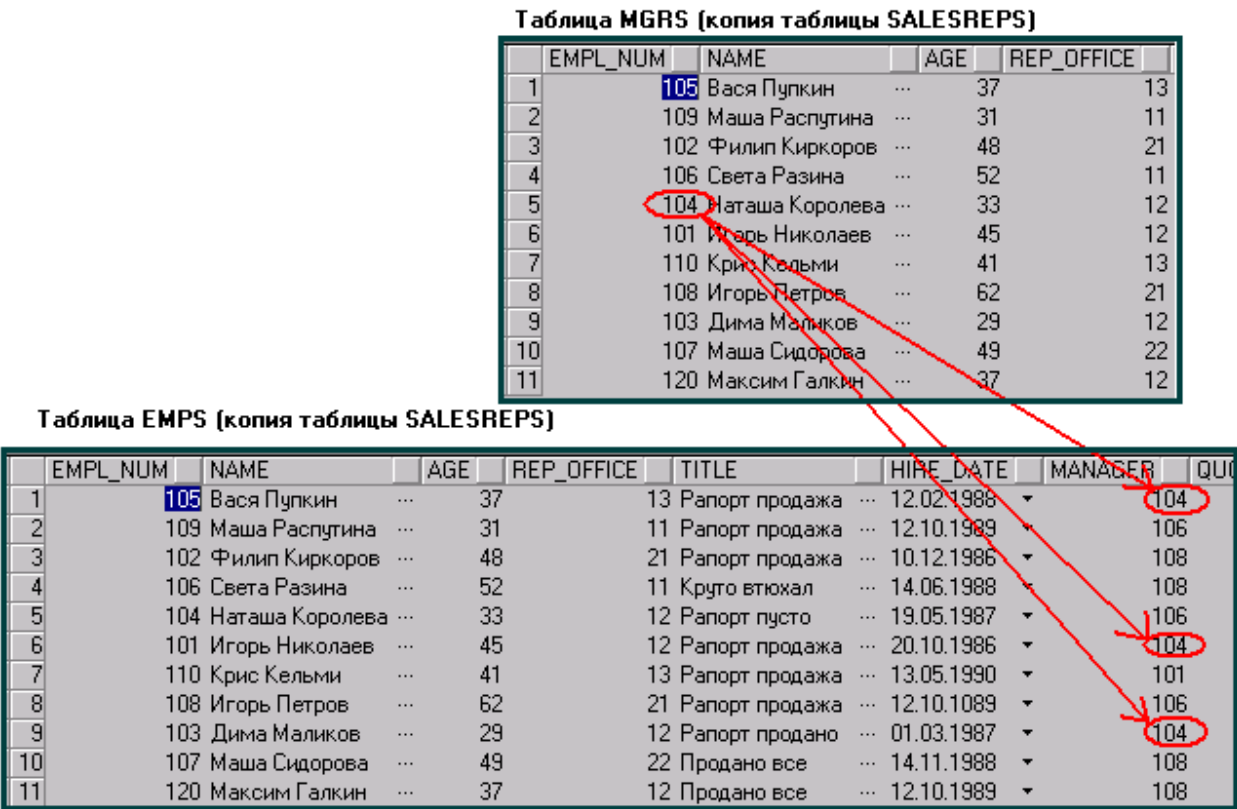


Рис. 1 Самообъединение таблицы SALESREPS

На рисунке хорошо видно как все это получается при использовании псевдонимов столбцов. Кстати можно сделать и так:

```
SELECT SALESREPS.NAME, MGRS.NAME
FROM SALESREPS, SALESREPS MGRS
WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM
/

SQL> SELECT SALESREPS.NAME, MGRS.NAME
2      FROM SALESREPS, SALESREPS MGRS
3      WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM
4  /
```

NAME	NAME
Вася Пупкин	Наташа Королева
Маша Распутина	Света Разина
Филип Киркоров	Игорь Петров
Света Разина	Игорь Петров
Наташа Королева	Света Разина
Игорь Николаев	Наташа Королева
Крис Кельми	Игорь Николаев
Игорь Петров	Света Разина
Дима Маликов	Наташа Королева

Маша Сидорова
Максим Галкин

Игорь Петров
Игорь Петров

11 rows selected

И вот, наконец, получаем результат самообъединения таблицы **SALESREPS**. В этом запросе я использовал один раз полное имя таблицы, а второй раз ее псевдоним! Собственно вы можете делать как вам нравится! :) Теперь, я приведу более "навороченные примеры самообъединения", а вы просто посмотрите и постарайтесь уяснить как это работает.

Запишем следующий запрос: Вывести список служащих, планы которых превышают планы их руководителей.

```
SELECT SALESREPS.NAME, SALESREPS.QUOTA, MGRS.QUOTA
FROM SALESREPS, SALESREPS MGRS
WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM
AND SALESREPS.QUOTA > MGRS.QUOTA
/
```

```
SQL> SELECT SALESREPS.NAME, SALESREPS.QUOTA, MGRS.QUOTA
2      FROM SALESREPS, SALESREPS MGRS
3      WHERE SALESREPS.MANAGER = MGRS.EMPL_NUM
4      AND SALESREPS.QUOTA > MGRS.QUOTA
5  /
```

NAME	QUOTA	QUOTA
Вася Пупкин	350	200
Маша Распутина	300	275
Игорь Николаев	300	200
Крис Кельми	400	300
Игорь Петров	350	275
Дима Маликов	275	200
Максим Галкин	400	350

7 rows selected

Получаем семь строк и три столбца, одно из условий использует неравенство в выражении поиска.

А вот, еще один запрос: Вывести список служащих, которые работают со своими руководителями в различных офисах, включая имена и офисы как служащих, так и руководителей.

```
SELECT EMPS.NAME, EMP_OFFICE.CITY, MGRS.NAME, MGR_OFFICE.CITY
FROM SALESREPS EMPS, SALESREPS MGRS, OFFICES EMP_OFFICE,
OFFICES MGR_OFFICE
WHERE EMPS.REP_OFFICE = EMP_OFFICE.OFFICE
AND MGRS.REP_OFFICE = MGR_OFFICE.OFFICE
AND EMPS.MANAGER = MGRS.EMPL_NUM
AND EMPS.REP_OFFICE <> MGRS.REP_OFFICE
/
```

Получаем:

```
SQL> SELECT EMPS.NAME, EMP_OFFICE.CITY, MGRS.NAME, MGR_OFFICE.CITY
2      FROM SALESREPS EMPS, SALESREPS MGRS, OFFICES EMP_OFFICE,
3      OFFICES MGR_OFFICE
4      WHERE EMPS.REP_OFFICE = EMP_OFFICE.OFFICE
5      AND MGRS.REP_OFFICE = MGR_OFFICE.OFFICE
6      AND EMPS.MANAGER = MGRS.EMPL_NUM
7      AND EMPS.REP_OFFICE <> MGRS.REP_OFFICE
8 /
```

NAME	CITY	NAME	CITY
Вася Пупкин	Бубурино	Наташа Королева	Чугуевск
Света Разина	Красный Мотоцикл	Игорь Петров	Котрогайка
Наташа Королева	Чугуевск	Света Разина	Красный Мотоцикл
Крис Кельми	Бубурино	Игорь Николаев	Чугуевск
Игорь Петров	Котрогайка	Света Разина	Красный Мотоцикл
Маша Сидорова	Запндрищинск	Игорь Петров	Котрогайка
Максим Галкин	Чугуевск	Игорь Петров	Котрогайка

7 rows selected

Снова семь строк и теперь уже четыре столбца, но таблиц уже две! Можете сами придумать что-нибудь по сложнее, это будет полезно! Замечу кстати, что при увеличении количества таблиц в запросе вы все более и более используете ресурсы сервера **SQL**, следовательно в БД рассчитанных на оперативную обработку транзакций (**OLTP**) время реакции на такие запросы будет увеличиваться, хотя это уже отдельная тема! :) В конечном итоге объединение представляет собой частный случай более общей комбинации данных из двух таблиц. Это есть ни что иное как "ДЕКАРТОВО ПРОИЗВЕДЕНИЕ" или просто "ПРОИЗВЕДЕНИЕ" таблиц.

Допустим, если дать запрос вида: Показать все возможные комбинации служащих и городов

```
SELECT NAME, CITY
FROM SALESREPS, OFFICES
/

SQL> SELECT NAME, CITY
2      FROM SALESREPS, OFFICES
3 /
```

NAME	CITY
Вася Пупкин	Запндрищинск
Маша Распутина	Запндрищинск
Филип Киркоров	Запндрищинск
Света Разина	Запндрищинск
Наташа Королева	Запндрищинск
Игорь Николаев	Запндрищинск
Крис Кельми	Запндрищинск
Игорь Петров	Запндрищинск
Дима Маликов	Запндрищинск
Маша Сидорова	Запндрищинск
Максим Галкин	Запндрищинск

Вася Пупкин	Красный Мотоцикл
Маша Распутина	Красный Мотоцикл
Филип Киркоров	Красный Мотоцикл
Света Разина	Красный Мотоцикл
Наташа Королева	Красный Мотоцикл
Игорь Николаев	Красный Мотоцикл
Крис Кельми	Красный Мотоцикл
Игорь Петров	Красный Мотоцикл
Дима Маликов	Красный Мотоцикл

NAME	CITY
------	------

Маша Сидорова	Красный Мотоцикл
Максим Галкин	Красный Мотоцикл
Вася Пупкин	Чугуевск
Маша Распутина	Чугуевск
Филип Киркоров	Чугуевск
Света Разина	Чугуевск
Наташа Королева	Чугуевск
Игорь Николаев	Чугуевск
Крис Кельми	Чугуевск
Игорь Петров	Чугуевск
Дима Маликов	Чугуевск
Маша Сидорова	Чугуевск
Максим Галкин	Чугуевск
Вася Пупкин	Бубурино
Маша Распутина	Бубурино
Филип Киркоров	Бубурино
Света Разина	Бубурино
Наташа Королева	Бубурино
Игорь Николаев	Бубурино
Крис Кельми	Бубурино
Игорь Петров	Бубурино

NAME	CITY
------	------

Дима Маликов	Бубурино
Маша Сидорова	Бубурино
Максим Галкин	Бубурино
Вася Пупкин	Котрогайка
Маша Распутина	Котрогайка
Филип Киркоров	Котрогайка
Света Разина	Котрогайка
Наташа Королева	Котрогайка
Игорь Николаев	Котрогайка
Крис Кельми	Котрогайка
Игорь Петров	Котрогайка
Дима Маликов	Котрогайка
Маша Сидорова	Котрогайка
Максим Галкин	Котрогайка

55 rows selected

Получаем 55 строк, чего и следовало ожидать - 5ть офисов и одиннадцать служащих ($5 * 11 = 55$). Вот собственно пока и все с многотабличными запросами и с объединениями. Я кое-что пока опускаю, так как мы с этим встретимся позднее и тогда вам будет понятнее то, что мы будем собственно изучать! Осталось совсем немного с оператором **SELECT** и в скором времени мы приступим к **PL/SQL**!

Шаг 31 - Снова SELECT - Применение агрегатных функций

Продолжим изыскания в области оператора **SELECT**, ну не деться от него никуда, такова жизнь и больше никакова! :) Итак, основные виды мы уже разобрали. Остается подведение итогов по столбцам таблиц, а именно правильно - "агрегатные функции". Агрегатные (или их еще называют СТАТИЧЕСКИЕ) функции, позволяют подводить промежуточные итоги оперируя числовыми или иного вида исчислимыми столбцами. Агрегатная функция принимает в качестве аргумента столбец таблицы целиком и возвращает одно значение. Например, агрегатная функция **AVG()** принимает столбец и вычисляет среднее значение всех чисел находящихся в столбце. То есть, берет все записи этого столбца, складывает все числа в этом столбце, а потом делит полученное на количество слагаемых (вспомнили школьный курс математики)!

Например, дадим такой запрос: Каковы средний плановый и средний фактический объем продаж в вашей компании?

```
SELECT AVG(QUOTA), AVG(SALES)
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT AVG(QUOTA), AVG(SALES)
      2      FROM SALESREPS
      3 /

AVG(QUOTA) AVG(SALES)
-----
318,181818 298,143090
```

Два, столбца и одна строка, а вот так это выглядит графически:

Таблица SALESREPS

E_DATE	MANAGER	QUOTA	SALES
02.1988	104	350	367,917
01.1989	106	300	392,728
02.1986	108	350	474,05
06.1988	108	275	299,912
05.1987	106	200	142,594
01.1986	104	300	305,673
05.1990	101	400	75,985
01.1989	106	350	361,865
03.1987	104	275	286,775
11.1988	108	300	186,042
01.1989	108	400	386,042

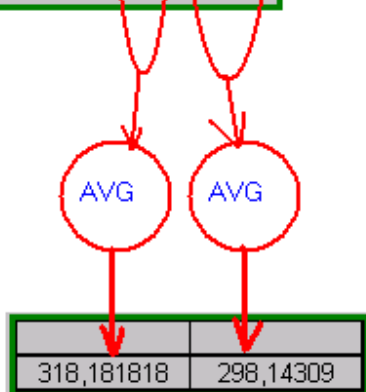


Рисунок 1 Простой агрегатныйзапрос

Надеюсь понятно, для начала? Давайте рассмотрим эти функции более внимательно, благо, их не так много:

- **SUM()** - Вычисляет сумму всех значений находящихся в столбце.
- **AVG()** - Вычисляет среднее всех значений содержащихся в столбце.
- **MIN()** - Находит наименьшее среди всех значений содержащихся в столбце.
- **MAX()** - Находит наибольшее среди всех значений содержащихся в столбце.
- **COUNT()** - Подсчитывает количество значений, содержащихся в столбце.
- **COUNT(*)** - Подсчитывает количество строк в таблице результатов запроса.

А вот, так это выглядит в представлении синтаксиса команд:

```
-- SUM ----- ( --- выражение ----- ) -----
-- DISTINCT -- имя столбца ---

-- AVG ----- ( --- выражение ----- ) -----
-- DISTINCT -- имя столбца ---

-- MIN ----- ( выражение ) -----

-- MAX ----- ( выражение ) -----

-- COUNT ----- ( ----- имя столбца ) -----
```

```
-- COUNT(*) -----  
-- DISTINCT --
```

Аргументом агрегирования может служить не только имя столбца, но и любое математическое выражение. Например, как в следующем запросе: Каков средний процент выполнения плана в вашей компании?

```
SELECT AVG(100 * (SALES/QUOTA))  
FROM SALESREPS  
/
```

Получаем:

```
SQL> SELECT AVG(100 * (SALES/QUOTA))  
2      FROM SALESREPS  
3  /
```

```
AVG(100*(SALES/QUOTA))  
-----  
94,4461905745769
```

А можно и так: Каков средний процент выполнения плана в вашей компании?

```
SELECT AVG(100 * (SALES/QUOTA)) PERCENT  
FROM SALESREPS  
/
```

```
SQL> SELECT AVG(100 * (SALES/QUOTA)) PERCENT  
2      FROM SALESREPS  
3  /
```

```
PERCENT  
-----  
94,4461905
```

В данном случае, более осмысленное имя столбца результата, хоть это не главное. Идем дальше. Давайте попробуем вычислять суммы столбцов. Применяем функцию **SUM()**, не забывайте, что столбец должен быть числовым! Например, вот так: Каковы средний плановый и средний фактический объемы продаж, служащих нашей компании?

```
SELECT SUM(QUOTA), SUM(SALES)  
FROM SALESREPS  
/
```

Получаем:

```
SQL> SELECT SUM(QUOTA), SUM(SALES)  
2      FROM SALESREPS  
3  /
```

```
SUM(QUOTA) SUM(SALES)  
-----
```

3500 3279,574

Еще: Какова сумма всех заказов принятых Димой Маликовым?

```
SELECT SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE NAME = 'Дима Маликов'
AND REP = EMPL_NUM
/
```

Получаем:

```
SQL> SELECT SUM(AMOUNT)
2      FROM ORDERS, SALESREPS
3      WHERE NAME = 'Дима Маликов'
4      AND REP = EMPL_NUM
5  /

SUM(AMOUNT)
-----
602,44
```

Да, рванул парень! :)))) Давайте, приведем еще несколько простых примеров с функцией агрегирования **AVG()**. Например такие: Вычислить среднюю цену товаров от производителя "ACI".

```
SELECT AVG(PRICE)
FROM PRODUCTS
WHERE MFR_ID = 'ACI'
/
```

Получаем:

```
SQL> SELECT AVG(PRICE)
2      FROM PRODUCTS
3      WHERE MFR_ID = 'ACI'
4  /

AVG(PRICE)
-----
411,821428
```

Вычислить среднюю стоимость заказа, размещенного компанией "Асме Mfg". (идентификатор клиента 2103)

```
SELECT AVG(AMOUNT)
FROM ORDERS
WHERE CUST = 2103
/
```

Получаем:

```
SQL> SELECT AVG(AMOUNT)
2      FROM ORDERS
3      WHERE CUST = 2103
4 /
```

```
AVG(AMOUNT)
-----
184,22
```

Для начала я думаю достаточно, можете пока поэкспериментировать самостоятельно с нашими учебными таблицами.

Шаг 32 - Снова SELECT - Агрегатные функции далее...

Двигаемся дальше. С агрегированием осталось немного. Нахождение экстремумов, а именно функции **MIN()**, **MAX()**. Не смотря на свою, с первого взгляда, незатейливость, подумаешь, найти минимум, да максимум. Но не все так просто. Эти функции работают как с числовыми столбцами, так и с датами и даже со строковыми переменными. Самое простое применение этих функций - это работа с числами.

Например, дадим запрос вида: Каковы наибольший и наименьший плановые объемы продаж?

```
SELECT MIN(QUOTA), MAX(QUOTA)
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT MIN(QUOTA), MAX(QUOTA)
2          FROM SALESREPS
3 /

MIN(QUOTA) MAX(QUOTA)
-----
200        400
```

Это столбцы, содержащие числа. Дадим еще один запрос: Когда был сделан самый первый из всех содержащихся в базе данных заказов?

```
SELECT MIN(ORDER_DATE)
FROM ORDERS
/
```

Получаем:

```
SQL> SELECT MIN(ORDER_DATE)
2          FROM ORDERS
3 /

MIN(ORDER_DATE)
-----
27.02.1989
```

Это уже даты, то есть промежутки времени и все зависит от длины промежутков. И еще простой запрос: Каков наибольший процент выполнения плана среди всех служащих?

```
SELECT MAX(100 * (SALES/QUOTA))
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT MAX(100 * (SALES/QUOTA))
2      FROM SALESREPS
3 /

MAX(100*(SALES/QUOTA))
-----
135,442857142857
```

Это снова числа. Так же сами можете поработать со строками. Но учтите, что разные виды кодировок применяемые в серверах **SQL**, могут давать различные результаты. Например, прописные буквы в моей системе идут первыми, потом строчные, а например в системе **EBCDIC** фирмы **IBM** все наоборот. Так, что имейте это в виду. А как, например, просто подсчитать количество записей? Для этого есть функция **COUNT()**. Эта функция всегда возвращает целое число, не зависимо от типа передаваемых данных.

Например: Сколько клиентов у нашей компании?

```
SELECT COUNT(CUST_NUM)
FROM CUSTOMERS
/
```

Получаем:

```
SQL> SELECT COUNT(CUST_NUM)
2      FROM CUSTOMERS
3 /

COUNT(CUST_NUM)
-----
22
```

Это целое число, так как клиентов не может быть 22.5, это если только вирусы поработали. :) Еще один запрос: Сколько служащих перевыполнили план?

```
SELECT COUNT(NAME)
FROM SALESREPS
WHERE SALES > QUOTA
/
```

Получаем:

```
SQL> SELECT COUNT(NAME)
2      FROM SALESREPS
3      WHERE SALES > QUOTA
4 /

COUNT(NAME)
-----
7
```

Как видим и условие нам не чуждо, пожалуйста задавайте любое! А вот еще один запрос на количество чего-либо: Сколько имеется заказов стоимостью более \$250 ?

```
SELECT COUNT(AMOUNT)
FROM ORDERS
WHERE AMOUNT > 250
/
```

Получаем:

```
SQL> SELECT COUNT(AMOUNT)
2      FROM ORDERS
3      WHERE AMOUNT > 250
4  /
```

```
COUNT(AMOUNT)
-----
7
```

Здесь то же самое, только условие немного другое. Вообще, если вдуматься здесь дословно звучит: "сколько стоимостей заказов?", по моему это глупо! Для этого существует функция **COUNT(*)**, она просто подсчитывает строки, а не значения данных. Тогда было бы грамотнее записать что-то вроде:

```
SELECT COUNT(*)
FROM ORDERS
WHERE AMOUNT > 250
/
```

Получаем:

```
SQL> SELECT COUNT(*)
2      FROM ORDERS
3      WHERE AMOUNT > 250
4  /
```

```
COUNT(*)
-----
7
```

Вот теперь, все как-то встало на свои места. Вообще **COUNT(*)**, очень полезная функция и вы в этом еще не раз убедитесь, но не советую ею злоупотреблять, так как она относится к разряду ресурсоемких и там где возможно ее лучше избегать! Итак понять, что делает запрос с агрегатной функцией на первый взгляд не сложно. Но если список возвращаемых столбцов содержит несколько агрегатных функций, да еще и включающие в себя математические выражения. Такой запрос является достаточно сложным для понимания! Например, запрос вида: Найти среднюю стоимость заказов, общую стоимость заказов, среднюю стоимость заказов в процентах от лимитов кредита клиентов, а также среднюю стоимость заказов в процентах от плановых объемов продаж служащих.

```
SELECT AVG(AMOUNT), SUM(AMOUNT), (100 * AVG(AMOUNT/CREDIT_LIMIT)), (100 *
AVG(AMOUNT/QUOTA))
FROM ORDERS, CUSTOMERS, SALESREPS
WHERE CUST = CUST_NUM AND REP = EMPL_NUM
/
```

Получаем:

```
SQL> SELECT AVG(AMOUNT), SUM(AMOUNT), (100 * AVG(AMOUNT/CREDIT_LIMIT)), (100 *  
1  AVG(AMOUNT/QUOTA))  
2      FROM ORDERS, CUSTOMERS, SALESREPS  
3      WHERE CUST = CUST_NUM AND REP = EMPL_NUM  
4  /  
  
AVG(AMOUNT) SUM(AMOUNT) (100*AVG(AMOUNT/CREDIT_LIMIT)) (100*AVG(AMOUNT/QUOTA))  
-----  
170,0368333  5101,105          347,302701523109      50,8331082251082
```

Достаточно замудренный запрос в жизни Вы, как правило, будете делать чуть проще, но кому как нравится. Для того, чтобы с этим всем разобраться попробуйте разложить запрос на две части, сначала представьте как все будет работать без агрегатных функций, а затем агрегируйте и все станет на свои места. Примерно вот так:

```
SELECT AMOUNT, AMOUNT, AMOUNT/CREDIT_LIMIT, AMOUNT/QUOTA  
FROM ORDERS, CUSTOMERS, SALESREPS  
WHERE CUST = CUST_NUM AND REP = EMPL_NUM  
/
```

Вот теперь видно, что происходит, добавьте все агрегаты и все заработает снова! Но запомните, что одна агрегатная функция не может быть аргументом для другой. Это формируется правилом "нельзя вкладывать агрегатные функции"! Хотя если очень интересно, можете попробовать! :) Также в списке возвращаемых столбцов нельзя одновременно указывать агрегатные функции и простые имена столбцов. Поскольку такой запрос не имеет смысла! Например:

```
SELECT NAME, SUM(SALES)  
FROM SALESREPS
```

Так как первый столбец создает таблицу, а второй возвратит только одно значение, что вызовет ошибку! Но здесь, есть еще хитрости, которые мы рассмотрим позже. Пока можете закрепить то, что уже проработано.

Шаг 33 - SELECT - Агрегаты и старый знакомый NULL!

А, вот интересно, а что будет происходить с функциями типа **AVG()**, **MIN()**, **MAX()**, **SUM()**, **COUNT()**, если значение столбца будет содержать значение нашего доброго старого знакомого - **NULL**? По правилам **ANSI/ISO** сказано, что "агрегатные функции игнорируют значение NULL"! Вот если честно, ну достал этот **NULL**, просто сил нет! :) Итак давайте проверим, на примере, что же будет происходить. Дадим вот такой запрос:

```
SELECT COUNT(*), COUNT(SALES), COUNT(QUOTA)
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT COUNT(*), COUNT(SALES), COUNT(QUOTA)
2 FROM SALESREPS
3 /

COUNT(*) COUNT(SALES) COUNT(QUOTA)
-----
11         11         10
```

Странный какой то результат? В чем тут вопрос. Таблица вроде одна, а вот значения в запросе разные. А все дело в том что, одно из полей **QUOTA** - содержит **NULL**. Отсюда и вся не разбериха. Функция **COUNT** вида **COUNT(поле)**, при работе, как и было сказано, игнорирует значение **NULL**, а **COUNT(*)** просто подсчитывает общее число строк. Ей все равно есть там **NULL** или нет! :) Как в том мультике - "Он нас всех посчитал!" :) Просто запомните вышесказанное и не будете делать в дальнейшем ошибок! А, вот функции **MIN()**, **MAX()** - особо не искажают результат при наличии **NULL**, так как так же его игнорируют. Но **AVG()**, **SUM()** - может при наличии **NULL** немного ввести вас в заблуждение! Например, посмотрите на следующий запрос:

```
SELECT SUM(SALES), SUM(QUOTA), (SUM(SALES) - SUM(QUOTA)), (SUM(SALES - QUOTA))
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT SUM(SALES), SUM(QUOTA), (SUM(SALES) - SUM(QUOTA)), (SUM(SALES - QUOTA))
2 FROM SALESREPS
3 /

SUM(SALES) SUM(QUOTA) (SUM(SALES)-SUM(QUOTA)) (SUM(SALES-QUOTA))
-----
3279,574   3100      179,574             103,589
```

С **SUM(SALES)**, **SUM(QUOTA)** все вроде бы ясно, но вот **(SUM(SALES)-SUM(QUOTA))** и **(SUM(SALES-QUOTA))** как-то странно выглядит. По закону математики собственно, оба выражения должны были вернуть одинаковые результаты, но на проверку это не так! Почему? Давайте подумаем. Выражение **SUM(SALES-QUOTA)** принимает в качестве аргумента только 10-ть значений, которые не равны **NULL**. В следствии того, что одно из полей **QUOTA**, содержит

значение **NULL**. То выражение **SUM(значение столбца - NULL)** возвращает **NULL**! **SUM** его игнорирует! Что и следовало ожидать! Таким образом одно из выражений полностью исключается! Тогда, встает вопрос - какое, выражение верно? А, вот ответ, вас еще более запутает! ОБА! Первое выражение вычисляет именно то, что и означает "СУММА по SALES - СУММА по QUOTA". И соответственно и второе выражение, так же вычисляет именно то, что и должно - "СУММА (SALES-QUOTA)"! Важно понять, что оба выражения справедливы и не содержат ошибок!!! Но значения возвращаемые выражениями отличаются, при наличии в полях таблицы значения **NULL**! По этому, важно: ХОРОШО ПРЕДСТАВЛЯТЬ, ЧТО ВЫ ХОТИТЕ ПОЛУЧИТЬ В ДАННОМ КОНКРЕТНОМ СЛУЧАЕ ПРИ ПОСТРОЕНИИ ЗАПРОСА! Иначе можно, наделать много ошибок и окончательно в них запутаться!!! :(Вот такой он "кровопийца", этот **NULL** и это еще не все где мы с ним будем разбираться! Вообще то, я стараюсь по мере, сил (а, сил у меня не меряно!!!) не забивать вам голову сухими формулировками! Но здесь как раз тот случай когда, следует их процитировать, итак:

1. Если какие либо из значений содержащихся в столбце, равны **NULL**, при вычислении результата функции они исключаются!
2. Если все значения в столбце равны **NULL**, то функции **AVG()**, **SUM()**, **MIN()**, **MAX()** возвращают значения **NULL**! Функция **COUNT()** возвращает ноль!
3. Если в столбце нет значений (т.е. столбец пуст), то функции **AVG()**, **SUM()**, **MIN()**, **MAX()** возвращают значения **NULL**! Функция **COUNT()** возвращает ноль!
4. Функция **COUNT(*)** подсчитывает количество строк и не зависит от наличия или отсутствия в столбце значений **NULL**! Если строка в столбце нет, то эта функция возвращает ноль!

Вот собственно и все вкратце, что касается **NULL** и агрегатных функций! Можете проверить все сами! :) Еще один интересный момент, касающийся функции **DISTINCT**. Ее тоже можно использовать с агрегатными функциями. Например в таких запросах:

1. Сколько различных названий рапортов существует в нашей компании?

```
SELECT COUNT(DISTINCT TITLE)
FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT COUNT(DISTINCT TITLE)
2  FROM SALESREPS
3  /

COUNT(DISTINCTTITLE)
-----
5
```

2. В скольких офисах есть служащие превысившие плановые объемы продаж?

```
SELECT COUNT(DISTINCT REP_OFFICE)
FROM SALESREPS
WHERE SALES > QUOTA
/
```

Получаем:

```
SQL> SELECT COUNT(DISTINCT REP_OFFICE)
2  FROM SALESREPS
3  WHERE SALES > QUOTA
4  /
```

```
COUNT(DISTINCTREP_OFFICE)
```

```
-----
4
```

Вкратце опишу основные понятия, при работе с **DISTINCT** и агрегатами. Если вы используете **DISTINCT** и агрегатную функцию, то ее аргументом может быть только имя столбца, выражение не может быть аргументом. В функциях **MIN()**, **MAX()** так же нет смысла использовать **DISTINCT**! В функции **COUNT()** в принципе можно использовать **DISTINCT**, но это требуется не часто. А вот к функции **COUNT(*)** вообще не применимо **DISTINCT**, так как она просто подсчитывает число строк! Так же в одном запросе **DISTINCT** можно употреблять только один раз! Если оно применяется с аргументом, агрегатной функции, его уже нельзя использовать ни с одним другим аргументом! Вот, такие правила для **DISTINCT** с агрегатами, поэкспериментируйте сами и сможете убедиться!

Шаг 34 - SELECT - Агрегаты и группировка данных

Агрегатные функции, которые мы рассматривали ранее, по своей сути формируют конечный итог для запрашиваемой таблицы. То есть, получаем только одну итоговую строку. Например: Какова средняя стоимость заказа?

```
SELECT AVG(AMOUNT)
FROM ORDERS
/
```

Получаем итоговый результат:

```
SQL> SELECT AVG(AMOUNT)
2 FROM ORDERS
3 /
```

```
AVG(AMOUNT)
-----
170,0368333
```

Такое положение дел не всегда устраивает при выполнении такого рода запросов. Например, необходимо найти промежуточный результат. В этом случае нам поможет запрос с "группировкой". А именно выражение **GROUP BY** оператора **SELECT**. Для начала дадим вот такой запрос с применением выражения **GROUP BY**: Какова средняя стоимость заказа для каждого служащего?

```
SELECT REP, AVG(AMOUNT)
FROM ORDERS
GROUP BY REP
/
```

Получаем сгруппированный запрос:

```
SQL> SELECT REP, AVG(AMOUNT)
2 FROM ORDERS
3 GROUP BY REP
4 /
```

```
REP AVG(AMOUNT)
-----
101    58,826
102     5,694
103   301,22
105  148,125
106   16,479
107 228,5933333
108 311,2064285
109    3,5515
110   327,25
```

9 rows selected

А вот, как это можно представить графически:

Таблица ORDERS

	ORDER_NUM	ORDER_DATE	CUST	REP	MFR	PRODUCT	QTY	AMOUNT
1	112961	17.12.1989	▼	2117	106 REI	2A44L	7	31,5
2	113012	11.06.1990	▼	2111	105 ACI	41003	35	3,745
3	112989	03.06.1990	▼	2101	106 FEA	114	6	1,458

Таблица с ГРУППИРОВКОЙ

	ORDER_NUM	REP	AMOUNT
1	112961	106	31,5
2	113012	105	3,745
3	112989	106	1,458
4	113051	108	1,42
5	112968	101	3,925
6	113036	110	22,5
7	113045	108	45
8	112963	105	3,276
9	113013	108	652
10	113058	109	1,478
11	112997	107	652
12	112983	105	702
13	113024	108	7,1
14	113062	107	2,43
15	112979	102	15

Результаты запроса	
REP	AVG(AMOUNT)
101	58,826
106	16,479
108	311,2064285
110	327,25

Group
by

Рис. 1 Выполнение запроса с группировкой

Поле **REP** в данном случае является полем группировки, то есть все значения поля **REP** собираются в группы и для каждой отдельно взятой группы просчитывается выражение **AVG(AMOUNT)**! Просто и ясно. Если говорить сухим языком формулировок, то это будет примерно следующее:

1. Заказы делятся на группы по одной группе для каждого служащего. В каждой группе все заказы имеют одно и тоже значение поля **REP**.
2. Для каждой группы вычисляется среднее значение столбца **AMOUNT** по всем строкам входящим в группу и генерируется одна итоговая строка результатов. Эта строка содержит значение столбца **REP** для группы и среднюю стоимость заказа для данной группы.

Надеюсь, теперь стало немного понятнее. Следовательно, запрос с применением выражения **GROUP BY**, называется "ЗАПРОС С ГРУППИРОВКОЙ"! А имя столбца, следующего за этим выражением, называется "столбцом группировки". Давайте для закрепления рассмотрим еще несколько запросов с группировкой.

Каков диапазон плановых объемов продаж для каждого офиса?

```
SELECT REP_OFFICE, MIN(QUOTA), MAX(QUOTA)
FROM SALESREPS
GROUP BY REP_OFFICE
/
```

Получаем искомый результат:

```
SQL> SELECT REP_OFFICE, MIN(QUOTA), MAX(QUOTA)
2 FROM SALESREPS
3 GROUP BY REP_OFFICE
4 /
```

REP_OFFICE	MIN(QUOTA)	MAX(QUOTA)
11	275	300
12	200	400
13	350	350
21	350	350
22	300	300

6 rows selected

Еще один запрос: Сколько служащих работают в каждом офисе?

```
SELECT REP_OFFICE, COUNT(*)
FROM SALESREPS
GROUP BY REP_OFFICE
/
```

Получаем результат:

```
SQL> SELECT REP_OFFICE, COUNT(*)
2 FROM SALESREPS
3 GROUP BY REP_OFFICE
4 /
```

REP_OFFICE	COUNT(*)
11	2
12	4
13	1
21	2
22	1

1

6 rows selected

И снова обратите внимание на последнюю строку, там уютно устроился наш старый добрый **NULL**, его я прописал намеренно, так как **SQLPlus** вернет пустоту, а вот **COUNT(*)** его видит!!! :)

Вот еще интересный запрос с группировкой: Сколько клиентов обслуживает каждый служащий?

```
SELECT COUNT(DISTINCT CUST_NUM), 'CUSTOMERS FOR SALESREPS', CUST_REP
FROM CUSTOMERS
GROUP BY CUST_REP
/
```

Получаем:

```
SQL> SELECT COUNT(DISTINCT CUST_NUM), 'CUSTOMERS FOR SALESREPS', CUST_REP
2 FROM CUSTOMERS
3 GROUP BY CUST_REP
4 /
```

```
COUNT(DISTINCTCUST_NUM) 'CUSTOMERSFORSALESREPS' CUST_REP
```

```
-----
3 CUSTOMERS FOR SALESREPS      101
4 CUSTOMERS FOR SALESREPS      102
3 CUSTOMERS FOR SALESREPS      103
1 CUSTOMERS FOR SALESREPS      104
2 CUSTOMERS FOR SALESREPS      105
2 CUSTOMERS FOR SALESREPS      106
1 CUSTOMERS FOR SALESREPS      107
2 CUSTOMERS FOR SALESREPS      108
2 CUSTOMERS FOR SALESREPS      109
1 CUSTOMERS FOR SALESREPS      110
1 CUSTOMERS FOR SALESREPS      120
```

11 rows selected

Здесь обратите внимание, на использование псевдополя '**CUSTOMERSFORSALESREPS**'. Просто, чтобы было нагляднее. Так же группировать результаты запроса можно и по нескольким столбцам. Например, вот так:

Подсчитать общее количество заказов по каждому клиенту для каждого служащего.

```
SELECT REP, CUST, SUM(AMOUNT)
FROM ORDERS
GROUP BY REP, CUST
/
```

Получаем следующее:

```
SQL> SELECT REP, CUST, SUM(AMOUNT)
2 FROM ORDERS
3 GROUP BY REP, CUST
4 /
```

```
REP  CUST SUM(AMOUNT)
-----
101  2102    3,978
101  2108     150
101  2113    22,5
102  2106    4,026
102  2114     15
102  2120    3,75
103  2111   602,44
105  2103   736,88
105  2111    3,745
106  2101    1,458
106  2117    31,5
```

107	2109	31,35
107	2124	654,43
108	2112	47,925
108	2114	7,1
108	2118	2123,42
109	2108	7,103
110	2107	654,5

18 rows selected

Тем не менее, при группировке по двум столбцам нельзя создать группы и подгруппы с двумя уровнями итоговых результатов. Но, тем не менее, возможно применить сортировку, чтобы результаты запроса шли в нужном порядке. Хотя обычно результаты запроса при использовании **GROUP BY**, сортируются автоматически. Рассмотрим такой запрос:

Подсчитать общее количество заказов по каждому клиенту для каждого служащего; отсортировать результаты запроса по клиентам и служащим.

```
SELECT REP, CUST, SUM(AMOUNT)
FROM ORDERS
GROUP BY REP, CUST
ORDER BY REP, CUST
/
```

Получаем то, что уже видели, но немного в другом порядке:

REP	CUST	SUM(AMOUNT)
101	2102	3,978
101	2108	150
101	2113	22,5
102	2106	4,026
102	2114	15
102	2120	3,75
103	2111	602,44
105	2103	736,88
105	2111	3,745
106	2101	1,458
106	2117	31,5
107	2109	31,35
107	2124	654,43
108	2112	47,925
108	2114	7,1
108	2118	2123,42
109	2108	7,103
110	2107	654,5

18 rows selected

Вот таким образом выражение **GROUP BY** заставляет **SELECT** обрабатывать группы. Пробуйте сами и закрепляйте свои познания! :)

Шаг 35 - SELECT - "Группировка" данных далее

Как вы, наверное, уже поняли с помощью одного запроса с применением агрегатных функций нельзя получить детальные и промежуточные результаты. Такие приемы возможны только с применением встроенного языка программирования **PL/SQL**, к которому мы уже почти подошли и как только разделаемся с **SELECT** - так как без него никуда, возьмемся за это дело. Тем не менее, в сервере **MS SQL** существует выражение **COMPUTE**, которое по своей сути подрывает все основы построения реляционных запросов. Но с его помощью, возможно получение такого рода результатов не прибегая к написанию хранимых процедур. Но это так к слову и для информации о том, что мелко мягкие вообще любят подрывать основы :) Так же на запросы с группировкой накладывается ряд ограничений. Например, столбцы с группировкой должны представлять собой реальные столбцы таблиц. Нельзя группировать строки на основе вычисляемого выражения. Так же существуют ограничения на элементы возвращаемых значений. Возвращаемым столбцом может быть:

1. Константа.
2. Агрегатная функция, возвращающая одно значение для всех строк входящих в группу.
3. Столбец группировки, который по определению имеет одно и тоже значение во всех строках группы.
4. Выражение, включающее в себя перечисленные выше элементы.

Обычно в список возвращаемых столбцов запросов с группировкой входят столбец группировки и агрегатная функция. Если не указать агрегат, то можно просто обойтись выражением **DISTINCT** без использования предложения **GROUP BY**! А так же, если в запрос не включить столбец группировки, вы не сможете определить, к какой именно группе относится та или иная строка результата запроса! Так же в **SQL92** игнорируется информация о первичных и вторичных ключах, при анализе запроса с группировкой.

Вот, например: Подсчитать общее количество заказов для каждого служащего.

```
SELECT EMPL_NUM, NAME, SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE REP = EMPL_NUM
GROUP BY EMPL_NUM
/
```

Уууупс! А вот и ошибочка!

```
SQL> SELECT EMPL_NUM, NAME, SUM(AMOUNT)
2 FROM ORDERS, SALESREPS
3 WHERE REP = EMPL_NUM
4 GROUP BY EMPL_NUM
5 /
```

```
SELECT EMPL_NUM, NAME, SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE REP = EMPL_NUM
GROUP BY EMPL_NUM
```

ORA-00979: выражение не является выражением GROUP BY

В данном случае имеется в виду поле **NAME**. Так как оно явно не вписывается в запросе! Хотя, если рассуждать с точки зрения природы данных, все вроде бы правильно, но не совсем.

Столбец **EMPL_NUM** является первичным ключом таблицы **SALESREPS**, поэтому столбец **NAME** должен иметь одно значение для каждой группы! Правильно, просто нужно указать этот столбец в выражении группировки вот так: Подсчитать общее количество заказов для каждого служащего.

```
SELECT EMPL_NUM, NAME, SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE REP = EMPL_NUM
GROUP BY EMPL_NUM, NAME
/
```

Вот теперь правильно! :)

```
SQL> SELECT EMPL_NUM, NAME, SUM(AMOUNT)
2 FROM ORDERS, SALESREPS
3 WHERE REP = EMPL_NUM
4 GROUP BY EMPL_NUM, NAME
5 /
```

EMPL_NUM	NAME	SUM(AMOUNT)
101	Игорь Николаев	176,478
102	Филип Киркоров	22,776
103	Дима Маликов	602,44
105	Вася Пупкин	740,625
106	Света Разина	32,958
107	Маша Сидорова	685,78
108	Игорь Петров	2178,445
109	Маша Распутина	7,103
110	Крис Кельми	654,5

9 rows selected

А можно сделать еще проще. Вот так: Подсчитать общее количество заказов для каждого служащего.

```
SELECT NAME, SUM(AMOUNT)
FROM ORDERS, SALESREPS
WHERE REP = EMPL_NUM
GROUP BY NAME
/
```

Получаем:

```
SQL> SELECT NAME, SUM(AMOUNT)
2 FROM ORDERS, SALESREPS
3 WHERE REP = EMPL_NUM
4 GROUP BY NAME
5 /
```

NAME	SUM(AMOUNT)
Вася Пупкин	740,625
Дима Маликов	602,44
Игорь Николаев	176,478
Игорь Петров	2178,445
Крис Кельми	654,5
Маша Распутина	7,103
Маша Сидорова	685,78
Света Разина	32,958
Филип Киркоров	22,776

9 rows selected

Теперь надеюсь, все стало ясно с группировками, если что не понятно можете писать письма! :) А вот теперь давайте вернемся к нашему старому знакомому - ну, конечно же, **NULL**. Что будет, если в одном из полей группировки будет присутствовать **NULL**? К какой группе его отнести? В предложении **WHERE** по правилам сравнение **NULL** и **NULL** скажем на равенство не дает **TRUE**, а будет равно **NULL**! В предложении **GROUP BY** это крайне не удобно, так как каждый **NULL** будет генерировать новую группу. Поэтому в стандарте **ANSI/ISO** определено, что в предложении **GROUP BY** значения **NULL** **РАВНЫ!!!** Следовательно, он не будут вносить неразбериху в запросах с группировкой! Что, собственно и требовалось в данном случае. Для примера создадим табличку **COLORIS** и проверим наши рассуждения.

Создаем:

```
CREATE TABLE COLORIS
(
  NM VARCHAR2(50),
  HAIR VARCHAR2(50),
  EYES VARCHAR2(50)
)
/

SQL> CREATE TABLE COLORIS
2 (
3 NM VARCHAR2(50),
4 HAIR VARCHAR2(50),
5 EYES VARCHAR2(50)
6 )
7 /
```

Table created

Табличка создана! Не забудьте дать оператор **COMMIT** вот так:

```
COMMIT
/
```

Если все верно, то получите следующее:

```
SQL> COMMIT
```


2 /

Теперь операцией копирования через буфер обмена отправьте в **SQLPlus** следующее:

```
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Cindy', 'Brown', 'Blue')
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Louise', NULL, 'Blue')
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Harry', NULL, 'Blue')
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Samantha', NULL, NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Joanne', NULL, NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('George', 'Brown', NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Mary', 'Brown', NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Paula', 'Brown', NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Kevin', 'Brown', NULL)
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Joel', 'Brown', 'Brown')
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Susan', 'Blonde', 'Blue')
/
INSERT INTO COLORIS(NM, HAIR, EYES)
VALUES('Marie', 'Blonde', 'Blue')
/
COMMIT
/
```

Если все успешно прошло получите примерно следующее:

```
SQL> INSERT INTO COLORIS(NM, HAIR, EYES)
2      VALUES('Cindy', 'Drown', 'Blue')
3 /

1 row inserted

.
```

```
.  
. .  
SQL> INSERT INTO COLORIS(NM, HAIR, EYES)  
2     VALUES('Marie', 'Blonde', 'Blue')  
3 /  
  
1 row inserted  
  
SQL> commit  
2 /  
  
Commit complete
```

Табличка заполнена данными! Как мы это проделали, напомним, применив оператор **INSERT**. Он отправляет данные в таблицы БД. О нем мы еще поговорим в дальнейшем. А вот теперь давайте наконец дадим запрос и проверим теорию равенства **NULL** в выражениях **GROUP BY**:

```
SELECT HAIR, EYES, COUNT(*)  
FROM COLORIS  
GROUP BY HAIR, EYES  
/
```

Получаем:

```
SQL> SELECT HAIR, EYES, COUNT(*)  
2 FROM COLORIS  
3 GROUP BY HAIR, EYES  
4 /
```

HAIR	EYES	COUNT(*)
Blonde	Blue	2
Brown	Blue	1
Brown	Brown	1
Brown	NULL	4
NULL	Blue	2
NULL	NULL	2

Хорошо видно, как **NULL** сформировал собственную группу! Вот теперь кажется разобрались в особенностях работы агрегатов со значениями **NULL** в БД **Oracle**. :)

Шаг 36 - SELECT - "Группировка" и отбор HAVING

Итак, наконец, мы почти приблизились к завершению оператора **SELECT**! Еще немного и можно будет сказать, что мы все обсудили! Но, а пока рассмотрим, как выбрать данные при группировке, то есть выводить не все данные, а только те, которые нас интересуют. Ранее для отбора строк по условию мы пользовались выражением **WHERE**. Для отбора групп по условию существует оператор **HAVING**. Его синтаксис аналогичен выражению **WHERE** и мало того их можно использовать вместе! Давайте рассмотрим следующий запрос:

Какова средняя стоимость заказа для каждого служащего из числа тех, у которых общая стоимость заказов превышает \$300?

```
SELECT REP, AVG(AMOUNT)
FROM ORDERS
GROUP BY REP
HAVING SUM(AMOUNT) > 300
/
```

Вот и результат:

```
SQL> SELECT REP, AVG(AMOUNT)
2  FROM ORDERS
3  GROUP BY REP
4  HAVING SUM(AMOUNT) > 300
5  /
```

	REP	AVG(AMOUNT)

	103	301,22
	105	148,125
	107	228,59333333
	108	311,2064285
	110	327,25

5 rows selected

Как видно выражение **HAVING SUM(AMOUNT) > 300** сработало как условие при группировке строк! А теперь посмотрим на рисунок.

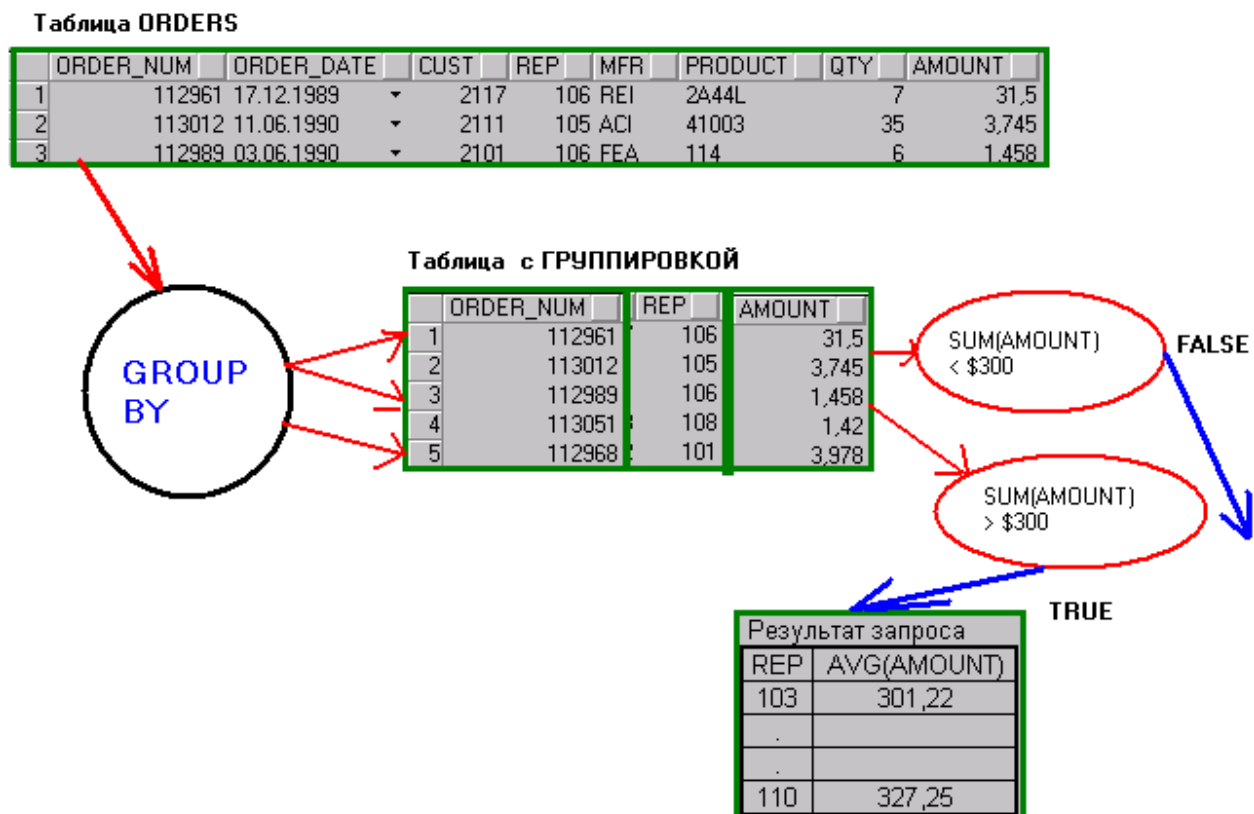


Рис. 1 Работа предложения HAVING

Здесь видно, если условие **SUM(AMOUNT) > 300** ложно, то эта группа из результирующего набора отбрасывается. Если истинно, то группа попадает в результирующий набор! Ничего сложного!

Давайте рассмотрим еще один пример: Для каждого офиса, в котором работают два и более человек, вычислить общий плановый и фактический объем продаж для всех служащих офиса.

```
SELECT CITY, SUM(QUOTA), SUM(SALESREPS.SALES)
FROM OFFICES, SALESREPS
WHERE OFFICE = REP_OFFICE
GROUP BY CITY
HAVING COUNT(*) >= 2
/
```

Получаем:

```
SQL> SELECT CITY, SUM(QUOTA), SUM(SALESREPS.SALES)
2 FROM OFFICES, SALESREPS
3 WHERE OFFICE = REP_OFFICE
4 GROUP BY CITY
5 HAVING COUNT(*) >= 2
6 /
```

```
CITY          SUM(QUOTA) SUM(SALESREPS.SALES)
```

Контрогайка	700	835,915
Красный Мотоцикл	575	692,637
Чугуевск	1175	1121,084

Здесь хорошо видно, что присутствуют оба выражения и **WHERE** и **HAVING**, каждый выполняет свою функцию в запросе. Обратите так же внимание, что в предложении **HAVING** используются агрегатные функции при формировании условия поиска! Давайте опишем, как этот запрос выполняется:

1. Объединяются таблицы **OFFICES** и **SALESREPS** для того, чтобы найти город, в котором работает служащий.
2. Группируются строки объединенной таблицы по офисам.
3. Исключаются группы, содержащие две или менее строки - это те строки, которые не удовлетворяют критерию предложения **HAVING**.
4. Вычисляются общие плановые и фактические объемы продаж для каждой группы.

Вот так строится этот запрос, если вы все это поняли. Замечательно. Тогда посмотрите кое что посложнее:

Показать цену, количество на складе и общее количество заказанных единиц для каждого наименования товара, если для него общее количество заказанных единиц превышает 75 процентов от количества товара на складе.

```
SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
FROM PRODUCTS, ORDERS
WHERE MFR = MFR_ID
GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
ORDER BY QTY_ON_HAND DESC
/
```

Получаем следующий набор:

```
SQL> SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
2 FROM PRODUCTS, ORDERS
3 WHERE MFR = MFR_ID
4 GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
5 HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
6 ORDER BY QTY_ON_HAND DESC
7 /
```

DESCRIPTION	PRICE	QTY_ON_HAND	SUM(QTY)
Лампа настольная	55	277	223
Рейка деревянная	107	207	223
Носки черные	76	167	223
Рейка пластмассовая	117	139	223
Труба алюминиевая	355	38	32
Карандаш простой	25	37	223
Подушка ватная	177	37	32
Нож специальный	475	32	30

Монитор LG	2500	28	223
Наушники SONY	975	28	30
Коробка картонная	2,75	25	223
Рубероид рулоны	250	24	30
Электродвигатель	243	15	26
Бочка пластмассовая	350	14	60
Доска профильная	4500	12	60
Телевизор SAMSUNG	4500	12	60
Профиль специальный	1875	9	30
Осветитель ртутный	1425	5	30
Тарелка фарфоровая	180	0	2

19 rows selected

А вот здесь, ничего разжевывать не буду, это вам задание на дом! Напишите план запроса и пришлите на мой ящик, а я посмотрю кто как понимает данный материал! :) Ограничения, накладываемые на предложение **HAVING** те же, что и ограничения для запросов с группировкой, которые мы рассматривали в [предыдущем шаге](#), но есть ряд дополнений. Предложение **HAVING** должно содержать как минимум одну агрегатную функцию, если это не так, то лучше применять предложение **WHERE**. Так как предложение **WHERE** применимо к отдельным строкам, а предложение **HAVING** к группам строк и об этом не стоит забывать! А, остальное все как обычно! :) Да, что касается **NULL** в условии поиска **HAVING**, они точно такие же, как и для предложения **WHERE**, т.е. "Если условие поиска имеет значение **NULL**, группа строк исключается и строка в результатах запроса для нее не генерируется". К слову предложение **HAVING**, в принципе можно применять и без **GROUP BY**, но тогда результат запроса рассматривается как одна группа состоящая из всех строк. На практике это применяется довольно редко. Вот так работает **HAVING** и все, что к нему прилагается!

Шаг 37 - Вопрос ответ и идем дальше !

Вот тут мне пришло письмо, по [прошлomu шагу](#). Очень радует, что уже есть обратная связь с теми для кого мы собственно все это делаем! Во-первых, я допустил ошибку в [прошлом шаге](#), не написав полностью условие в рассматриваемом запросе. Так что давайте разберемся со всем этим! :) Вопрос был такой:

```
>Домашнее задание от 05.10.03.
>Разобрать запрос:
>
>SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
>FROM PRODUCTS, ORDERS
>WHERE MFR = MFR_ID
>GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
>HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
>ORDER BY QTY_ON_HAND DESC
>/
>
> 1. Сначала в одну собираются две таблицы PRODUCTS и
>   ORDERS на основе равенства полей MFR = MFR_ID
> 2. Потом формируются группы по полям MFR_ID, PRODUCT_ID,
>   DESCRIPTION, PRICE, QTY_ON_HAND и считается сумма поля
>   QTY для каждой группы.
> 3. После этого исключаются из списка выводимых группы,
>   в которых SUM(QTY) > (0.75 * QTY_ON_HAND)
> 4. И в конце получившаяся последовательность строк сортируется
>   по убыванию значения поля QTY_ON_HAND
>
>Вот вроде всё.. Но есть вопрос... Зачем в ORDER BY стоит PRODUCT_ID?
>По логике поле DESCRIPTION должно быть уникальным (ведь пользователю
>не удобно оперировать с кодами). Нельзя ли ORDER BY сократить до
>DESCRIPTION, PRICE, QTY_ON_HAND?
>Заранее благодарен.
```

Отвечаю! В принципе концептуально Вы совершенно правы! Правда Вы наверное имели в виду **GROUP BY**, в своем вопросе. И так же я приведу текст правильной формулировки этого запроса:

Показать цену, количество на складе и общее количество заказанных единиц для каждого наименования товара, если для него общее количество заказанных единиц превышает 75 процентов от количества товара на складе.

```
SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
FROM PRODUCTS, ORDERS
WHERE MFR = MFR_ID
      AND PRODUCT = PRODUCT_ID
GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
ORDER BY QTY_ON_HAND DESC
/
```

Я забыл вписать эту строку "**AND PRODUCT = PRODUCT_ID**"! Вот по этим двум полям дополнительно и указано **GROUP BY**! Так как по правилам **ANSI/ISO** необходимо указывать все поля фигурирующие в запросе.

Получаем:

```
SQL> SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
2 FROM PRODUCTS, ORDERS
3 WHERE MFR = MFR_ID
4       AND PRODUCT = PRODUCT_ID
5 GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND
6 HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
7 ORDER BY QTY_ON_HAND DESC
8 /
```

DESCRIPTION	PRICE	QTY_ON_HAND	SUM(QTY)
Труба алюминиевая	355	38	32
Карандаш простой	25	37	30
Электродвигатель	243	15	16
Телевизор SAMSUNG	4500	12	15
Осветитель ртутный	1425	5	22

А вообще этот запрос можно записать и так, собственно ничего не изменится:

```
SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
FROM PRODUCTS, ORDERS
WHERE MFR = MFR_ID
      AND PRODUCT = PRODUCT_ID
GROUP BY DESCRIPTION, PRICE, QTY_ON_HAND
HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
ORDER BY QTY_ON_HAND DESC
/
```

Получаем:

```
SQL> SELECT DESCRIPTION, PRICE, QTY_ON_HAND, SUM(QTY)
2 FROM PRODUCTS, ORDERS
3 WHERE MFR = MFR_ID
4       AND PRODUCT = PRODUCT_ID
5 GROUP BY DESCRIPTION, PRICE, QTY_ON_HAND
6 HAVING SUM(QTY) > (0.75 * QTY_ON_HAND)
7 ORDER BY QTY_ON_HAND DESC
8 /
```

DESCRIPTION	PRICE	QTY_ON_HAND	SUM(QTY)
Труба алюминиевая	355	38	32
Карандаш простой	25	37	30
Электродвигатель	243	15	16
Телевизор SAMSUNG	4500	12	15
Осветитель ртутный	1425	5	22

Но в целом для полноты картины не плохо записать **GROUP BY**, как **GROUP BY MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND**.

Так вернее синтаксически! Хотя не принципиально! Так что, особой ошибки здесь нет! Но некоторые правила все же стоит соблюдать! Вот собственно, если говорить по большому счету, с **SELECT** мы более менее разобрались! В следующий раз я думаю, начнем очень серьезную тему а именно **PL/SQL**! Что является основной тематикой моего раздела. Осталось еще кое-что по вложенным запросам и объединениям таблиц, но я подумал и решил, что оставлю это на рассмотрение в совокупности с **PL/SQL**! Так будет интереснее и нагляднее! А, так же не будем забывать и о сервере **Oracle**, в частности администрирование и т.д. И думаю постепенно мы это все одолеем, а путь у нас не близкий!!! :)

Шаг 38 - PL/SQL - вводный курс

Итак, наконец, мы добрались до основного пункта! Язык **PL/SQL** - это процедурный язык представляющий собой расширение стандарта **ANSI** языка **SQL**, разработанного фирмой **Oracle**. Собственно **SQL**, не является процедурным языком, да и, по большому счету, он вообще не относится к языкам программирования.

PL/SQL - это процедурный язык пошагового программирования, инкапсулирующий язык **SQL**. В результате получается хорошо развитый язык программирования третьего поколения (**3GL**), подобный языку **C++**, **Pascal** и т. д. В своей сути **PL/SQL** блочно ориентирован.

PL/SQL имеет строгие правила области видимости переменных, поддерживает параметризованные вызовы процедур и функций и так же унаследовал от языка **ADA** такое средство, как пакеты (**package**).

PL/SQL предусматривает строгий контроль типов, все ошибки несовместимости типов выявляются на этапе компиляции и выполнения. Так же поддерживается явное и неявное преобразование типов.

PL/SQL - поддерживает сложные структуры данных, так же предусмотрена перегрузка подпрограмм, для создания гибкой среды прикладного программирования.

Язык **PL/SQL** - имеет элемент **Exception Handler** (обработчик исключительных ситуаций) для синхронной обработки ошибок на этапе выполнения кода **PL/SQL**.

Так же строго говоря, язык **PL/SQL** не является объектно-ориентированным, хотя имеет некоторые средства для создания и работы с объектами БД на уровне объектно-ориентированных языков программирования. Запутанно, правда? Но тем не менее, это так и есть и в дальнейшем вы в этом убедитесь! :) Что-то очень похожее на ООП в **PL/SQL** имеется.

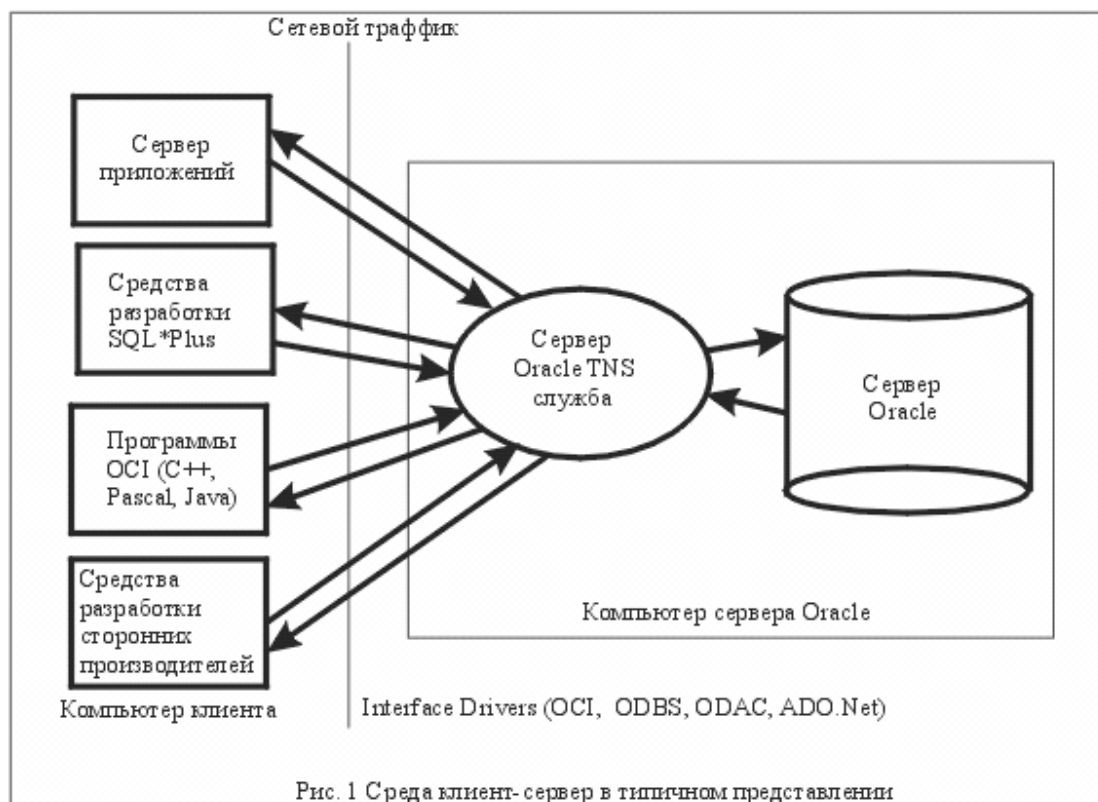
PL/SQL - является машинно независимым языком программирования! И это действительно так! Например, не нужно изучать **PL/SQL** для **Windows** или для **Unix**! Сам код программных блоков **PL/SQL** не зависит от платформы, на которой они выполняются. Сам язык остается таким же как и он есть!!! Вот вам и мультиплатформенность!

PL/SQL - поддерживает стандартные интерфейсы работы с языками высокого уровня такими как **C**, **C++** - через предкомпиляторы поставляемые фирмой **Oracle**. Например, для работы с языком **C** есть такое средство как **OCI (Oracle Call Interface)**.

Так же **PL/SQL** имеет ряд встроенных средств для работы с **Internet**, прямо из хранимых процедур. Имеет поддержку создания **HTTP** запросов так же непосредственно из хранимых процедур. Вот, что из себя представляет встроенный язык **PL/SQL** сервера **Oracle**. Однако прежде чем приступить, к изучению **PL/SQL** необходимо сказать несколько слов о его среде выполнения.

В типичной клиент/серверной среде обычно, самое узкое место это - сеть и то как она построена и отлажена вашим администратором сети. Кстати взаимодействие администратора сети и администратора БД, является немаловажным фактором!

Посмотрим на рисунок:



Здесь хорошо видно, что используется при работе с сервером БД **Oracle**. Клиенты использующие программы на языках высокого уровня, с большим количеством единичных запросов **SQL**. Средства разработки **SQL*Plus**, серверы приложений, и т.д. Между клиентами и сервером, как правило, стоит некий посредник, так же определяющий скорость обработки запросов поступающих от клиента к серверу. На жаргоне БД админов, их еще называют "толстыми" или "тонкими" клиентами. Например, я на первом этапе работы с БД использовал сервер БД **InterBase**, а клиентов создавал на негнбиаемом **Borland Delphi** с использованием "толстого" клиента **IBX** компонент. Затем я подрос и стал работать с сервером БД **Oracle**, а клиентские части разрабатывал уже на **Borland C++**, с использованием "тонкого" клиента **ODAC.NET** компонент, фирмы **CrLab** (www.crlab.com), сегодня для работы с **Oracle** сервером я использую **MS Visual Studio.Net**, а именно среду разработки **C#** с "толстым" клиентом **OraADO.NET** той же **CrLab**. Собственно получается довольно не плохо! Что такое "толстый" и "тонкий" клиенты, расскажу чуть позже. Пока надеюсь ясно как сервер **Oracle** реализует взаимодействие с внешним миром.

Идем дальше:

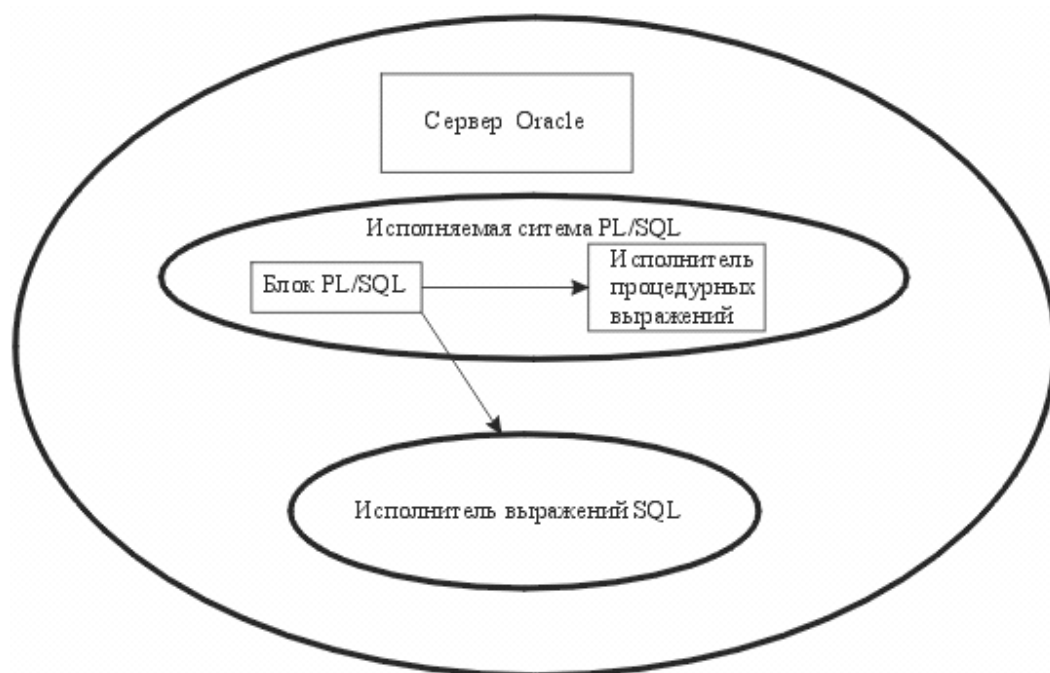


Рис. 2 Система исполнитель языка PL/SQL (является компонентом сервера БД Oracle)

Программы, написанные на языке **PL/SQL**, выполняются системой-исполнителем языка, которая представляет собой часть сервера БД **Oracle**. Независимо от средства, с помощью которого формируется исполняемый код, он посылается на сервер **Oracle**. Система исполнитель языка **PL/SQL** сканирует, разбирает и компилирует код. После этого код, готов к выполнению. Выполняется код посредством передачи его **SQL Statement Executor** (системе исполнителю **SQL**-кода). Набор данных, полученных в результате исполнения запроса, поступает в систему исполнитель **PL/SQL** для дальнейшей обработки. Вот таким образом действует этот механизм. Далее давайте рассмотрим, в чем преимущество использования **PL/SQL** как процедурного языка.

Посмотрим на рисунок:

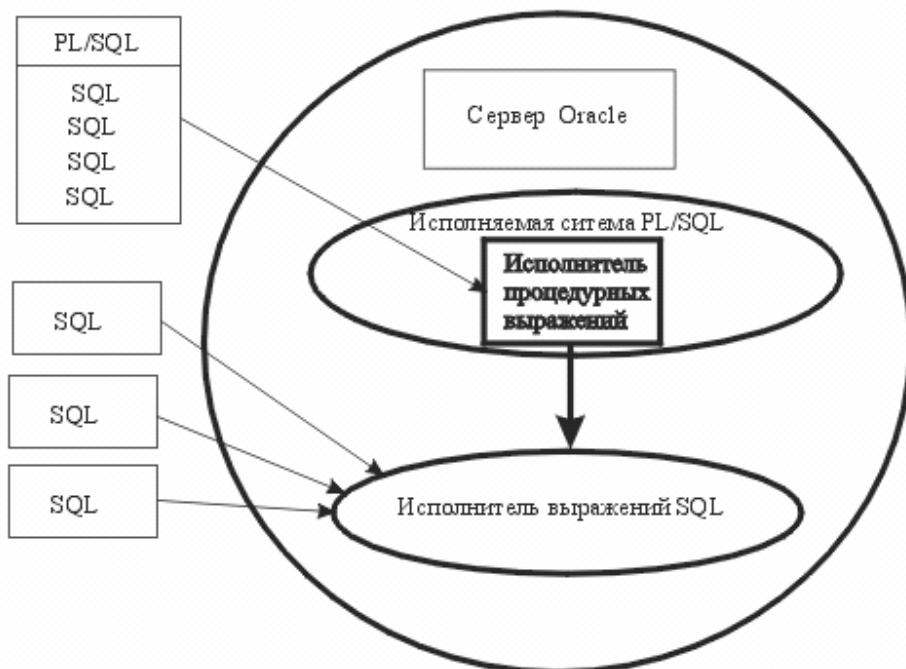


Рис. 3 Группировка SQL кода в единый блок языка PL/SQL, позволяет значительно уменьшить загрузку сети

Основное преимущество при работе с **PL/SQL** как языком БД является то, что на исполнение серверу посылается группа **SQL** предложений, а не единичные запросы сформированные, скажем какой-либо программой не использующей **PL/SQL** как средство работы с БД. Действительно, получив **PL/SQL** блок, сервер приступает к его обработке, а клиенту остается только ожидать результата операции. Тогда как единичные запросы к БД, порождают большой трафик и тормозят работу сети в целом. То есть, говоря точным языком, на сервере снижается число активных транзакций, вследствие того, что за одну активную транзакцию, обрабатывается большее число операторов **PL/SQL**! Вот таким образом, строится в целом работа с блоками **PL/SQL**.

Шаг 39 - PL/SQL - топам дальше .. топ .. топ

Итак, в [прошлый раз](#) мы рассмотрели часть понятий, используемых внутренней структурой компилятора **PL/SQL**. Идем дальше. Дальнейшим развитием процесса разработки программного кода и уменьшение трафика сети при работе с **PL/SQL**, является использование откомпилированных и именованных блоков **PL/SQL**. А именно процедур и функций. Понятие "именование", означает хранение имени подпрограммы, включая ее код. В **PL/SQL**, присутствуют так же неименованные блоки. До них мы еще доберемся. Посмотрим на рисунок:

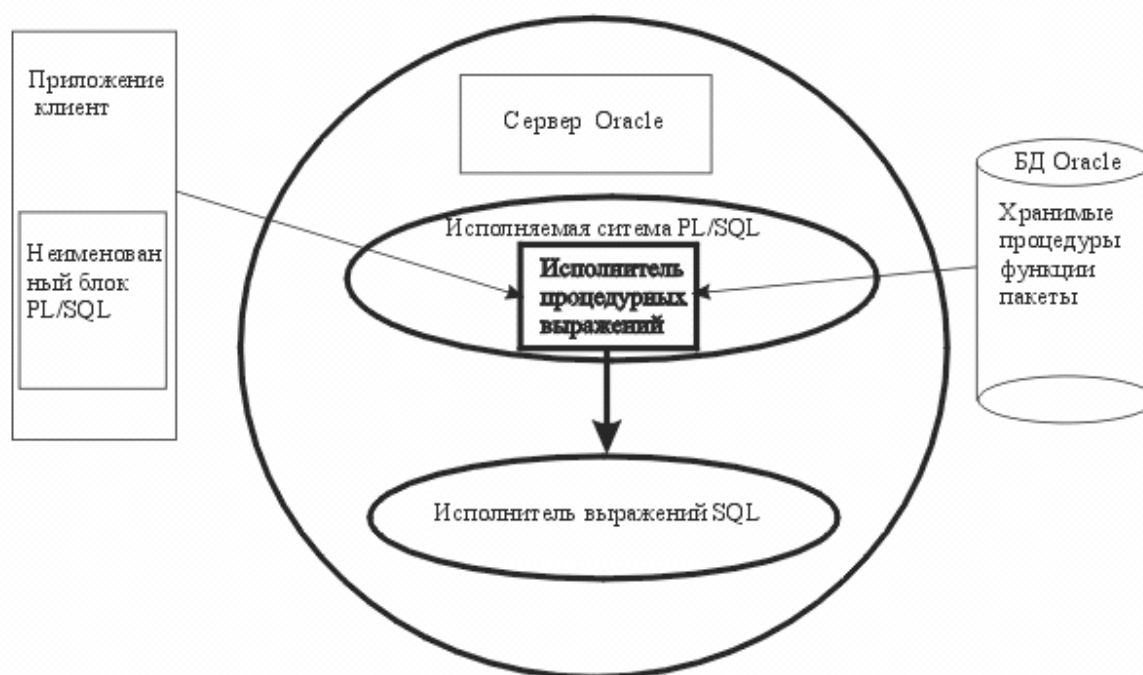


Рис.1 Система-исполнитель языка PL/SQL при выполнении хранимых процедур

Здесь показано, как выполняются хранимые процедуры и функции, вызываемые неименованными блоками. Подпрограммы могут выполнять достаточно сложные вычисления, а так же обрабатывать ошибки. Такой вызов подпрограмм еще называют "удаленный вызов процедур" (**Remote Procedure Call**). Либо возможно так же вызовы одной подпрограммы другой! Использование хранимых процедур и функций позволяет вам добиться максимальной производительности сервера БД, а так же эффективного повторного использования ранее созданного кода. Все блоки **PL/SQL**, в процессе компиляции хранятся в системном КЭШе - называемом **SGA** (**System Global Area**), там же находятся все таблицы схемы, переменные и т.д. А вот теперь запоминайте хорошенько! Пространство, выделенное для каждого отдельного блока **PL/SQL**, называется "**КУРСОР**"! Все **PL/SQL** блоки сохраняются в **SGA** посредством алгоритма **Least Recently Used** (звучит как - использованные наиболее давно). Любой **SQL** код внутри блока **PL/SQL**, так же получает собственные разделяемые области **SQL**. Так же по завершении компиляции подпрограммы запоминаются в словаре данных. Код всех подпрограмм является - реентерабельным! То есть к подпрограмме может обращаться несколько схем согласно прав использования подпрограмм. Все подпрограммы загруженные в область **SGA** являются разделяемыми. Вот пока на этом остановимся и поговорим на тему средств создания программ на **PL/SQL**. Обычно для этого достаточно блокнота, который встроен в **Windows** и среды **SQL*Plus**. Все можно приступить к написанию программ на **PL/SQL**! Просто и со вкусом, для начала я вам советую так и делать.

Но со временем, если кому хочется работать на более удобных средствах, есть большое количество сред разработки **PL/SQL** кода. Начиная от программ поставляемых самой фирмой **Oracle** и заканчивая множеством программ сторонних производителей. Например, я использую для написания кода, да и самого текста шагов **EmEditor Version 3.36** (www.emeditor.com). Очень удобная штучка, он поддерживает синтаксис всех языков программирования! И как простенький текстовый редактор очень удобен. А вот средство, для компиляции написанных блоков я уже очень давно использую **PL/SQL Developer version 5.0.1.480**! Очень удобная и развитая программа! Его можно скачать и посмотреть [вот здесь](#), правда это не бесплатная штука! Так что, как говорится, на вкус и цвет товарищей нет! Выбирайте, что вам удобнее и можете приступать! Как я уже сказал, блокнота и **SQL/Plus** для начала вполне хватит! :) Так что, пока за красивыми программками можете не гоняться! Вот собственно все, продолжим далее!

Шаг 40 - PL/SQL - дальше и дальше в пути

"хорошая сторона ошибки в том,
что вы узнаете ее когда совершите снова"
Марк Твен

Продолжаем PL/SQL! Итак, как известно, наилучший способ изучить язык программирования - это погрузиться в него с головой! То есть начать писать программы на том языке, который вы собственно хотите изучить. По этому, помаленьку приступим к изучению самого языка. В **PL/SQL** действует соглашение, что все символы приводятся к верхнему регистру по этому объявления типа:

```
a VARCHAR2(5);  
A VARCHAR2(5);
```

Одинаковы! По этому, например, я взял сразу за правило все писать в верхнем регистре сразу, так меньше путаницы! Так же, естественно, запрещено использовать зарезервированные имена встроенных функций и пакетов! Каждый законченный оператор обрамляется символом ";". Вообще, если говорить прямо, например когда, я начал работать с **PL/SQL**, уже на первом этапе я четко увидел, что сам **PL/SQL** очень похож на язык **Pascal**. По этому, когда я со всем этим занимался, я как раз работал очень много на **Pascal** и для меня не было особых трудов привыкнуть к **PL/SQL**! Он мне дался довольно легко и без лишних усилий, что думаю получится и у вас! Итак, давайте разберемся для начала со всеми специальными символами, я думаю это будет полезно! Обычно разбор всех языков с этого и начинается!

Типы специальных символов **PL/SQL**:

1. Арифметические операторы:

+	Сложение и унарный плюс
-	Вычитание и унарный минус
*	Умножение
/	Деление
**	Возведение в степень

2. Операторы отношения (используются в логических выражениях):

=	Равенство
<	Меньше
>	Больше
<>	Не равно
!=	Не равно (альтернатива)
~=	Не равно (альтернатива)
^=	Не равно (альтернатива)
<=	Меньше или равно

>=	Больше или равно
--------------	------------------

3. Выражение и списки (используются в операторах, объявлениях типов данных, объявлениях списков параметров, ссылках на переменные и таблицы):

:=	Присвоение
(Начало списка или подвыражения
)	Конец списка или подвыражения
,	Отдельные элементы списка (как в списке параметров)
..	Оператор диапазона используется в операторах FOR-IN
 	Конкатенация строк
=>	Ассоциация (используется в списке параметров)
;	Конец выражения
%	Атрибут курсора или типа объекта
.	Спецификация объекта
@	Индикатор удаленной базы данных
'	Начало/конец строки символов
:	Индикатор внешней переменной
&	Индикатор связанной переменной

4. Комментарии и метки

--	Комментарий в одной строке
/*	Начало многострочного комментария
*/	Конец многострочного комментария
>>	Начало метки
<<	Конец метки

Самое интересное, нафига, такое количество способов, сказать не равно! Хотя **!=** по моему вполне достаточно! Я например больше люблю вот так **<>!** :) Но вообще кому как нравится! Так же замечу, что вложенные комментарии не допускаются! Проще использовать **/* */** как в языке **C!** А теперь про самое интересное - блоки **PL/SQL**.

Блоки **PL/SQL**, могут быть, как я уже говорил "именованными" и "не именованными". Блок **PL/SQL** является фундаментальной программной конструкцией! Программирование модулями позволяет разрабатывать легко читаемый код и программировать сверху вниз. Неименованный блок **PL/SQL**, имеет три раздела - **Declaration** (объявления), **Body** (тело) и, как правило, **Exception** (исключения).

Стандартная конструкция неименованного блока:

```
DECLARE
-- объявления
```

```
BEGIN
-- выполняемый код

EXCEPTION
-- обработка исключений

END;

/ -- символ завершения для запуска блока на компиляцию
```

Итак, запускаем блокнот и **SQL*Plus**, пришло время написать и запустить вечную мантру программистов - правильно!!! **"Hello World!"**. Итак, код реализующий данную вечную и незыблемую мантру таков:

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line('Hello World!!!');
END;
/
```

Получаем после компиляции:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2  DBMS_OUTPUT.enable;
  3  DBMS_OUTPUT.put_line('Hello World!!!');
  4  END;
  5  /
Hello World!!!

PL/SQL procedure successfully completed
```

УРА!!! Получилось!!! А вот теперь немного, почесав затылок, попробуем разобраться, что же здесь для чего? Итак строка **SET SERVEROUTPUT ON** заставляет, сервер выводить сообщения как бы на "экран" или консоль. Что-то в этом духе (эту строку нужно вводить один раз на сеанс, при последующих запусках, ее можно не использовать!). Хотя, если правильно, то выводит пакет **DBMS_OUTPUT**, а первая строка просто заставляет сервер показывать то, что выводит пакет **DBMS_OUTPUT** с помощью метода **put_line()**, то есть показывать то, что возвращает сервер. **BEGIN END** - это обрамляющий программный блок. Пункт **DECLARE** отсутствует, в следствии того что, нам пока ничего не нужно декларировать! Но в дальнейшем он нам понадобится! **"/"** - символ запускающий код на исполнение. Строка **"PL/SQL procedure successfully completed"** просто говорит, что блок выполнен успешно. Вот собственно и все!!!

Можете написать, что ни будь свое и попробовать, что получится! Еще раз проштудируйте специальные символы, так как далее мы будем ими часто пользоваться!

Шаг 41 - PL/SQL - блоки, литералы, переменные

Итак, ваш первый "курсор" в области **SGA**, совершив работу, выдал **"Hello World!"**. Как он это сделал вы уже, наверное, успели немного разобраться! Давайте двигаться дальше. В [прошлый раз](#) мы просто вывели текст на экран и все проверив работу мантры! :) Но от таких программ мало толку и по этому давайте сделаем что-то полезное. Как правило, самой простой программой, которой обычно начинают изучать какой либо язык, является калькулятор! Конечно же писать калькулятор на языке **PL/SQL** - это излишество, но для начала вполне приемлемо! Итак, давайте разберем пару блоков **PL/SQL**:

```
SET SERVEROUTPUT ON -- Need first start

DECLARE

A INTEGER;
B INTEGER;

BEGIN

    A := 3;
    B := 5;

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(A+B);

END;
/
```

Получаем следующее:

```
SQL> SET SERVEROUTPUT ON -- Need first start

SQL> DECLARE
2
3 A INTEGER;
4 B INTEGER;
5
6 BEGIN
7
8     A := 3;
9     B := 5;
10
11     DBMS_OUTPUT.enable;
12     DBMS_OUTPUT.put_line(A+B);
13
14 END;
15 /
8
```

PL/SQL procedure successfully completed

Давайте разбирать. В разделе **DECLARE** мы объявили две переменных - **A** и **B**. Все переменные объявляются между оператором **DECLARE** и **BEGIN**! Затем в теле блока мы присвоили значениям переменных числа 3 и 5. Инициализация переменных может производиться либо при объявлении, либо в теле блока! Как вам больше нравится! Например, можно было сделать вот так:

```
DECLARE

A INTEGER := 3;
B INTEGER := 7;

BEGIN
.
```

Далее при выводе результата мы прямо в теле метода **put_line** объявили сумму двух переменных. Но вернее было бы провести явное преобразование типа с помощью функции **TO_CHAR()** вот так:

```
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(A+B));
```

В первом примере мы не сделали ошибки, а произошло неявное преобразование типа. Но мой совет делать именно так. Будет лучше читаться и меньше будете путаться! В данном случае можно записать более сложное выражение для наглядности и я думаю вам станет яснее почему я так делаю:

```
DECLARE

A INTEGER := 3;
B INTEGER;
K VARCHAR2(2) := '12';

BEGIN

    B := 5;

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(TO_CHAR(TO_NUMBER(K)*A+B));

END;
/
```

Получаем следующее:

```
SQL> DECLARE
2
3 A INTEGER := 3;
4 B INTEGER;
5 K VARCHAR2(2) := '12';
6
7 BEGIN
8
9 B := 5;
```

```
10
11      DBMS_OUTPUT.enable;
12      DBMS_OUTPUT.put_line(TO_CHAR(TO_NUMBER(K)*A+B));
13
14 END;
15 /
41
```

PL/SQL procedure successfully completed

Хорошо виден приоритет унарных операций $12*3+5 = 41$ - верно. Функция **TO_NUMBER()**, преобразует строковый литерал **K** в числовой литерал. С этими функциями преобразования мы еще не раз встретимся и, смею вас заверить, их очень много и будем разбирать их последовательно, так будет проще. Так же я разнообразно инициализировал переменные, что тоже не плохо, но лучше делать это одним способом, код будет более читаем. К стати можно записать и так:

```
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(K*A+B));
```

Но все же явное преобразование более понятно, и не теряйте чувство реальности! Давайте более ближе посмотрим на литералы. Например можно записать:

DECLARE

```
A VARCHAR2(1) := ' ';
B VARCHAR2(128) := 'Hello World!!!';
C VARCHAR2(128) := 'How "are" you?';
D VARCHAR2(128) := 'Hello Bob - "ok"!';
E VARCHAR2(128) := '12345';
F VARCHAR2(128) := '01/01/1989';
G VARCHAR2(128) := '!@#$%^&*()_";<,.?';
H VARCHAR2(128) := '" "';
```

```
I NUMBER := 12345;
J NUMBER := -12345;
K NUMBER := 12345.023745;
L NUMBER := 100.;
M NUMBER := 1.0237E2;
N NUMBER := 1.0237E-2;
O NUMBER := 0.34223;
P NUMBER := .321434;
```

BEGIN

```
DBMS_OUTPUT.enable;

DBMS_OUTPUT.put_line(A);
DBMS_OUTPUT.put_line(B);
DBMS_OUTPUT.put_line(C);
DBMS_OUTPUT.put_line(D);
DBMS_OUTPUT.put_line(E);
DBMS_OUTPUT.put_line(F);
```

```
DBMS_OUTPUT.put_line(G);
DBMS_OUTPUT.put_line(H);

DBMS_OUTPUT.put_line(TO_CHAR(I));
DBMS_OUTPUT.put_line(TO_CHAR(J));
DBMS_OUTPUT.put_line(TO_CHAR(K));
DBMS_OUTPUT.put_line(TO_CHAR(L));
DBMS_OUTPUT.put_line(TO_CHAR(M));
DBMS_OUTPUT.put_line(TO_CHAR(N));
DBMS_OUTPUT.put_line(TO_CHAR(O));
DBMS_OUTPUT.put_line(TO_CHAR(P));

END;
/
```

Получаем следующее:

```
SQL> DECLARE
2
3 A VARCHAR2(1) := ' ';
4 B VARCHAR2(128) := 'Hello World!!!';
5 C VARCHAR2(128) := 'How "are" you?';
6 D VARCHAR2(128) := 'Hello Bob - "ok"!';
7 E VARCHAR2(128) := '12345';
8 F VARCHAR2(128) := '01/01/1989';
9 G VARCHAR2(128) := '!@#$$%^&*()_":;<,>?';
10 H VARCHAR2(128) := '""';
11
12 I NUMBER := 12345;
13 J NUMBER := -12345;
14 K NUMBER := 12345.023745;
15 L NUMBER := 100.;
16 M NUMBER := 1.0237E2;
17 N NUMBER := 1.0237E-2;
18 O NUMBER := 0.34223;
19 P NUMBER := .321434;
20
21 BEGIN
22
23     DBMS_OUTPUT.enable;
24
25     DBMS_OUTPUT.put_line(A);
26     DBMS_OUTPUT.put_line(B);
27     DBMS_OUTPUT.put_line(C);
28     DBMS_OUTPUT.put_line(D);
29     DBMS_OUTPUT.put_line(E);
30     DBMS_OUTPUT.put_line(F);
31     DBMS_OUTPUT.put_line(G);
32     DBMS_OUTPUT.put_line(H);
33
34     DBMS_OUTPUT.put_line(TO_CHAR(I));
35     DBMS_OUTPUT.put_line(TO_CHAR(J));
36     DBMS_OUTPUT.put_line(TO_CHAR(K));
```

```

37      DBMS_OUTPUT.put_line(TO_CHAR(L));
38      DBMS_OUTPUT.put_line(TO_CHAR(M));
39      DBMS_OUTPUT.put_line(TO_CHAR(N));
40      DBMS_OUTPUT.put_line(TO_CHAR(O));
41      DBMS_OUTPUT.put_line(TO_CHAR(P));
42
43 END;
44 /

```

```

Hello World!!!
How 'are' you?
Hello Bob - "ok"!
12345
01/01/1989
!@#$%^&*()_";<,.?
','
12345
-12345
12345,023745
100
102,37
,010237
,34223
,321434

```

PL/SQL procedure successfully completed

Итак строковые литералы могут быть типов **CHAR**, **VARCHAR2**. В строковых литералах компилятор различает регистры символов. Давайте посмотрим, что здесь и как! Строковые литералы:

```

A VARCHAR2(1) := ' ';           -- пробел
B VARCHAR2(128) := 'Hello World!!!'; -- строка
C VARCHAR2(128) := 'How "are" you?'; -- строка с выводом символа '
D VARCHAR2(128) := 'Hello Bob - "ok"!'; -- строка с выводом символа "
E VARCHAR2(128) := '12345';      -- это не число, это строка
F VARCHAR2(128) := '01/01/1989'; -- это не дата, это строка. хотя похоже!
G VARCHAR2(128) := '!@#$%^&*()_";<,.?'; -- литералы могут содержать и специальные
символы!
H VARCHAR2(128) := '""';         -- строка содержит три символа "" ""
-- (одинарная кавычка, пробел одинарная кавычка)

```

Числовые литералы:

```

I NUMBER := 12345;              -- целое число
J NUMBER := -12345;             -- отрицательное число
K NUMBER := 12345.023745;       -- плавающая точка
L NUMBER := 100.;               -- с нулевой точностью
M NUMBER := 1.0237E2;           -- научная запись
N NUMBER := 1.0237E-2;          -- отрицательная степень
O NUMBER := 0.34223;            -- ведущий ноль не нужен!
P NUMBER := .321434;            -- можно и так!

```

Хорошим стилем программирования является объявление и инициализация переменных. Если инициализировать переменную, нет необходимости, то инициализацию можно опустить. Типы переменных мы уже рассматривали, все они применимы при их объявлении. Так что, можете с этим поэкспериментировать и проверить все ли так как я вам рассказал! :) Далее мы еще вернемся к созданию программы калькулятора.

Шаг 42 - PL/SQL - переменные, инициализация, правила

Кажется, жизнь налаживается. Продолжаем разбираться с переменными. Итак, еще раз внимательнее посмотрим, как их правильно записывают и, как и где инициализируют. Итак, давайте рассмотрим такую последовательность:

```
DECLARE          -- Открывается неименованный блок.
  Price NUMBER(5,2) := 12.43;      -- Переменная типа NUMBER (с плавающей точкой
                                   -- с точностью 5ть знаков, два после запятой.
  Sytki NUMBER := 123;             -- Переменная типа NUMBER простая.
  LGos INTEGER := 2;              -- Переменная типа INTEGER подкласс NUMBER типа.
  Max_Dist CONSTANT REAL := 0.45; -- Константа типа REAL.
                                   -- Тип CONSTANT требует обязательной инициализации!!!
  Dis_Tp VARCHAR2(1) := NULL;     -- Строковая переменная типа VARCHAR2 (присвоено
значение NULL)
  Ds_Nm CHAR(20) := NULL; -- Строковая переменная типа CHAR (присвоено значение NULL)
  Tk_Ir VARCHAR2(50) NOT NULL := 'Hello World!!!'; -- Строковая переменная
                                   -- типа VARCHAR2 не может иметь значение NULL!!!
  Tk_Sr BOOLEAN NOT NULL := TRUE; -- БУЛЕВА! переменная типа BOOLEAN не может
иметь значение NULL!!!
  It_Nm VARCHAR2(50) DEFAULT 'Hummer'; -- Строковая переменная типа VARCHAR2 имеет
значение по умолчанию
```

Теперь напишем неименованный блок и посмотрим все ли правильно мы написали:

```
SET SERVEROUTPUT ON
DECLARE
  Price NUMBER(5,2) := 12.43;
  Sytki NUMBER := 123;
  LGos INTEGER := 2;
  Max_Dist CONSTANT REAL := 0.45;
  Dis_Tp VARCHAR2(1) := NULL;
  Ds_Nm CHAR(20) := NULL;
  Tk_Ir VARCHAR2(50) NOT NULL := 'Hello World!!!';
  Tk_Sr BOOLEAN NOT NULL := TRUE;
  It_Nm VARCHAR2(50) DEFAULT 'Hummer';

BEGIN
  DBMS_OUTPUT.enable;
  DBMS_OUTPUT.put_line(TO_CHAR(Price));
  DBMS_OUTPUT.put_line(TO_CHAR(Sytki));
  DBMS_OUTPUT.put_line(TO_CHAR(LGos));
  DBMS_OUTPUT.put_line(TO_CHAR(Max_Dist));
  DBMS_OUTPUT.put_line(Dis_Tp);
  DBMS_OUTPUT.put_line(Ds_Nm);
  DBMS_OUTPUT.put_line(Tk_Ir);
  DBMS_OUTPUT.put_line(It_Nm);
END;
/
```

Получаем такой вывод **SQL*Plus**:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      Price NUMBER(5,2) := 12.43;
  3      Sytki NUMBER := 123;
  4      LGos INTEGER := 2;
  5      Max_Dist CONSTANT REAL := 0.45;
  6      Dis_Tp VARCHAR2(1) := NULL;
  7      Ds_Nm CHAR(20) := NULL;
  8      Tk_Ir VARCHAR2(50) NOT NULL := 'Hello World!!!';
  9      Tk_Sr BOOLEAN NOT NULL := TRUE;
 10      It_Nm VARCHAR2(50) DEFAULT 'Hummer';
 11
 12 BEGIN
 13      DBMS_OUTPUT.enable;
 14      DBMS_OUTPUT.put_line(TO_CHAR(Price));
 15      DBMS_OUTPUT.put_line(TO_CHAR(Sytki));
 16      DBMS_OUTPUT.put_line(TO_CHAR(LGos));
 17      DBMS_OUTPUT.put_line(TO_CHAR(Max_Dist));
 18      DBMS_OUTPUT.put_line(Dis_Tp);
 19      DBMS_OUTPUT.put_line(Ds_Nm);
 20      DBMS_OUTPUT.put_line(Tk_Ir);
 21      DBMS_OUTPUT.put_line(It_Nm);
 22 END;
 23 /
12,43
123
2
,45
Hello World!!!
Hummer
```

Процедура PL/SQL успешно завершена.

Действительно, первые три переменных просто вывелись на экран. Это числовые переменные разных типов и точности. А вот четвертую переменную стоит рассмотреть по ближе. Это **CONSTANT** переменная, то есть константа. Кроме того, она имеет тип **REAL**, являющимся разновидностью **NUMBER**. При таком типе объявления переменная должна сразу же получать значение, то есть инициализироваться! Все константы должны инициализироваться при объявлении! Запомните.

Пятая и шестая переменные - это просто строковые литералы разных типов, это я думаю понятно. Седьмая переменная - это строковый литерал, но с хитростью! Он не может иметь значение **NULL**! Так обычно определяют первичный ключ в таблице. Здесь тоже самое просто применяется, при объявлении переменной!

А, вот восьмая переменная - это, как говорят, отдельная песня! Это **BOOLEAN** переменная, да еще с грифом **NOT NULL**! Вообще в **PL/SQL** тип **BOOLEAN** привнесен, как мне кажется, немного искусственно для того, чтобы он был полноценным языком программирования. Почему? Как вам, наверное, известно, а если нет, то говорю, в **Oracle**, как и во многих **SQL** серверах в таблицах нельзя объявить тип поля (столбца) **BOOLEAN**! В этом смысле я обычно применяю поле **VARCHAR2(1)** и просто записываю туда либо **T**, либо **F**! А не иметь в самом **PL/SQL** типа **BOOLEAN**, было бы не очень красиво! По этому он есть. Что самое интересное, это я вывел

экспериментально, **TRUE** соответствует **1**, а **FALSE** соответствует **0**! Прямо как в языке **C++**! Собственно чего и следовало ожидать!

Девятая переменная, это тоже строковый литерал, но он кроме этого имеет значение по умолчанию! Тот же фокус проделывается, когда создается таблица БД! Вот так, если кратко объявляются и инициализируются переменные в **PL/SQL**! Практически все способы, которые вам могут понадобиться в дальнейшем, я вам показал! А, дальше это все нужно хорошенько запомнить или записать!!! :)

Кстати, все правила работы с **BOOLEAN** переменными и константами, те же что я приводил в ["Шаг 17 - Составные операторы в условии WHERE"](#)! Есть еще функция **XOR** она выглядит, для троичной логики так:

X	Y	X XOR Y
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
TRUE	NULL	NULL
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
FALSE	NULL	NULL
NULL	TRUE	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

Вот таким образом строится работа для функции **XOR**, для **AND**, **OR**, **NOT** смотрите [шаг 17](#). Пока мы с вами делали, линейные алгоритмы блоков. В следующем шаге будем рассматривать способы ветвления и правила работы со строками. Пока закрепляйте пройденное!!! :)

Шаг 43 - PL/SQL - оператор IF - THEN - ELSE

Итак, попробуем использовать операторы ветвления. Самым простым в использовании, в **PL/SQL** является оператор **IF**. Его основная логическая форма имеет вид:

```
IF(некоторое условие справедливо) THEN
-- условие справедливо, выполнять это.
ELSE      -- условие не выполняется
-- выполнять оператор в этой части.
END IF;   -- конец условного оператора.
```

Первый блок оператора после **IF-THEN** называется "ПРЕДЫДУЩИМ", а блок следующий за **ELSE** "ПОСЛЕДУЮЩИМ". Каждый оператор **IF-THEN** может содержать обрамляющие блоки **BEGIN - END**, если в этом есть необходимость, так как при записи типа:

```
IF(некоторое условие справедливо) THEN
-- оператор 1
-- оператор 2
-- оператор 3
-- оператор 4
END IF;   -- конец условного оператора.
```

Можно уверенно сказать, что все четыре оператора выполняются наверняка! Но, если применить такую конструкцию:

```
IF(некоторое условие справедливо) THEN
BEGIN
-- оператор 1
-- оператор 2
-- оператор 3
-- оператор 4
END;
END IF;   -- конец условного оператора.
```

Код будет более нагляден, хотя это вопрос относится к стилю программирования. Так же считается, хорошим стилем наличие комментария при использовании оператора **IF**. Особенно в случае большого количества вложений. Оператор **IF** можно вкладывать на любую глубину выражения. Таким образом, можно задать достаточно сложную логику выражения. Так же для экономии памяти, возможно использовать конструкцию, типа:

```
IF(некоторое условие справедливо) THEN  -- проверка условия
BEGIN
-- условие справедливо, выполнять это.
.
.
END;
ELSE  -- условие не выполняется
DECLARE
x NUMBER;
BEGIN
.
```

```
END;  
END IF;  -- конец условного оператора.
```

В этом случае переменная **x** будет создана, если условие предыдущего блока ложно. Хотя использовать, такие конструкции не всегда оправдано. Так же условные операторы в **PL/SQL** вычисляются с помощью так называемой "сокращенной" оценки. Допустим, есть такое условие:

```
IF( a AND b ) THEN -- проверка условия  
BEGIN  
.  
.  
END;  
END IF;  -- конец условного оператора.
```

Если выражение **a** равно **FALSE**, то дальнейшее выражение не вычисляется! Все выражения вычисляются слева направо, а выражения в скобках, имеют наивысший приоритет. Например, вот так:

```
IF( (a OR f) AND (c OR k) ) THEN  -- проверка условия  
BEGIN  
.  
.  
END;  
END IF;  -- конец условного оператора.
```

Выражения в скобках вычисляются первыми. Что ж, пришло время вспомнить старого знакомого! Работа с **NULL**. Оператор **IF** работает с **NULL** достаточно просто и эффективно. Если записать:

```
IF(X = NULL) THEN -- проверка условия  
.  
.  
END IF;  -- конец условного оператора.  
  
IF(X != NULL) THEN  -- проверка условия  
.  
.  
END IF;  -- конец условного оператора.
```

Такое условие будет иметь значение **FALSE** и ничего выполняться не будет. Правильная запись будет такая:

```
IF(X IS NULL) THEN -- проверка условия  
.  
.  
END IF;  -- конец условного оператора.  
  
IF(X IS NOT NULL) THEN  -- проверка условия  
.  
.  
END IF;  -- конец условного оператора.
```

От троичной логики никуда не деться, по этому запоминайте хорошенько! :) А, что делать, если необходимо проверить одно значение несколько раз? Например, вот так:

```
IF( val = 1 ) THEN -- проверка условия
.
.
END IF; -- конец условного оператора.

.
.
.

IF( val = 9 ) THEN -- проверка условия
.
.
END IF; -- конец условного оператора.
```

Такое количество операторов **IF** конечно можно использовать, но это слишком громоздко и не верно, так как существует конструкция **IF - THEN - ELSIF**. работает она достаточно просто и ее для таких целей по моему опыту вполне достаточно:

```
IF ( val = 1 ) THEN -- проверка условия
.
.
ELSIF ( val = 2 ) THEN
.
.
ELSIF ( val = 3 ) THEN
.
.
ELSIF ( val = 9 ) THEN
.
.
ELSE -- не сработало не одно из условий!
.
END IF; -- конец условного оператора.
```

Вот так это выглядит, достаточно просто и наглядно. Очень похоже на оператор **CASE** в языке **Pascal**. Но не совсем, хотя выполняет ту же функцию. Секция **ELSE** срабатывает, если не выполнилось ни одно из условий. Таким образом работает этот оператор. Как видите, на первый взгляд, все достаточно просто, хотя и реализует всю функциональность самого языка. Данный оператор является первым фундаментальным оператором языка **PL/SQL**. Что ж теперь, давайте изобразим небольшую программу для закрепления. Запускайте **SQL*Plus** и в любом редакторе наберите следующее:

```
-- OPER VARCHAR2(5) := '&TODO';

SET SERVEROUTPUT ON

DECLARE

A INTEGER := 7;
```

```
B INTEGER := 4;
OPER VARCHAR2(2) := '+';

BEGIN

    DBMS_OUTPUT.enable;
    IF (OPER = '+') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'sum = '||TO_CHAR(A+B));
    ELSIF (OPER = '-') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'res = '||TO_CHAR(A-B));
    ELSIF (OPER = '*') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'mul = '||TO_CHAR(A*B));
    ELSIF (OPER = '/') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'div = '||TO_CHAR(A/B));
    END IF;

END;
/
```

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3 A INTEGER := 7;
4 B INTEGER := 4;
5 OPER VARCHAR2(2) := '+';
6
7 BEGIN
8
9 DBMS_OUTPUT.enable;
10 IF (OPER = '+') THEN
11 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'sum = '||TO_CHAR(A+B));
12 ELSIF (OPER = '-') THEN
13 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'res = '||TO_CHAR(A-B));
14 ELSIF (OPER = '*') THEN
15 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'mul = '||TO_CHAR(A*B));
16 ELSIF (OPER = '/') THEN
17 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'div = '||TO_CHAR(A/B));
18 END IF;
19
20 END;
21 /
Operation is + sum = 11
```

Процедура PL/SQL успешно завершена.

В данном случае я применил конструкцию **IF - THEN - ELSIF** для создания чего-то похожего на калькулятор. В операторе **DBMS_OUTPUT.put_line** - для сцепления строк применяем конструкцию конкатенации строк вида "||" в результате вот такого выражения **('Operation is '||OPER||' '||'sum = '||TO_CHAR(A+B))** мы получаем строку **"Operation is + sum = 11"**. Таким образом, происходит сцепление отдельных строковых литералов. Ее мы еще не раз применим в дальнейшем. В проверках условий в операторах **IF - THEN - ELSIF** хорошо видно, что производится сравнение

переменной со строковым литералом, такие конструкции так же применимы при осуществлении проверок. Давайте немного усложним код программы, заменив выражение **OPER VARCHAR2(2) := '+'**; на **OPER VARCHAR2(5) := '&TODO'**; вот так:

```
SET SERVEROUTPUT ON

DECLARE

A INTEGER := 7;
B INTEGER := 4;
OPER VARCHAR2(5) := '&TODO';

BEGIN

    DBMS_OUTPUT.enable;
    IF (OPER = '+') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'sum = '||TO_CHAR(A+B));
    ELSIF (OPER = '-') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'res = '||TO_CHAR(A-B));
    ELSIF (OPER = '*') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'mul = '||TO_CHAR(A*B));
    ELSIF (OPER = '/') THEN
        DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'div = '||TO_CHAR(A/B));
    END IF;

END;
/
```

После запуска увидите следующее:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3 A INTEGER := 7;
4 B INTEGER := 4;
5 OPER VARCHAR2(5) := '&TODO';
6
7 BEGIN
8
9 DBMS_OUTPUT.enable;
10 IF (OPER = '+') THEN
11 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'sum = '||TO_CHAR(A+B));
12 ELSIF (OPER = '-') THEN
13 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'res = '||TO_CHAR(A-B));
14 ELSIF (OPER = '*') THEN
15 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'mul = '||TO_CHAR(A*B));
16 ELSIF (OPER = '/') THEN
17 DBMS_OUTPUT.put_line('Operation is '||OPER||' '||'div = '||TO_CHAR(A/B));
18 END IF;
19
20 END;
21 /
```


Введите значение для todo: *

После запуска введите одну из типов операции +, -, *, / !!! А, затем увидите следующее, если введете скажем *:

```
прежний  5: OPER VARCHAR2(5) := '&TODO';  
новый    5: OPER VARCHAR2(5) := '*';
```

```
Operation is * mul = 28  
Процедура PL/SQL успешно завершена.
```

Вот теперь наш калькулятор стал задавать нам вопросы, а виноват в этом литерал **&**, он трактуется **SQL*Plus**, как ждать ввода связанной переменной с именем **"todo"**! В данном случае **"todo"**, есть так называемая "связанная" переменная. Вот так, просто и без затей. А вам домашнее задание переписать код для ввода, сначала чисел, а затем операции над ними! И запомнить, как работает оператор ветвлений в **PL/SQL**. Дерзайте!!!

Шаг 44 - PL/SQL - GOTO и его определения, типы данных...

Давайте, для полноты картины познакомимся с оператором **PL/SQL - GOTO**. В обычных условиях, применение этого оператора, как наверное и во многих языках программирования не приветствуется. В следствии того, что код вашей программы теряет свою блочную структуру и такой код трудно сопровождать. Тем не менее, знать как работает этот оператор вам пригодится. Давайте посмотрим на следующий фрагмент кода:

```
BEGIN
  BEGIN
    GOTO MID;
  END;
  <<MID>>
  NULL;
END;
/
```

Собственно **<<MID>>** есть метка для указания места перехода оператора **GOTO**. Так она объявляется. Двойные кавычки после метки не ставятся. Оператор **NULL** является заполнителем, вместо него вы могли бы вставить свой код. Таким образом данный код легко выполняется, но ничего не производит:

```
SQL> BEGIN
2   BEGIN
3     GOTO MID;
4
5   END;
6   <<MID>>
7   NULL;
8 END;
9 /
```

Процедура PL/SQL успешно завершена.

Рассмотрим несколько другой случай:

```
BEGIN
  GOTO MID;
  BEGIN
    <<MID>>
    NULL;
  END;
END;
/
```

В результате получаем:

```
SQL> BEGIN
2   GOTO MID;
3   BEGIN
4     <<MID>>
```

```
5          NULL;
6      END;
7  END;
8  /
      GOTO MID;
      *
```

ошибка в строке 2:

ORA-06550: Строка 2, столбец 10:

PLS-00201: идентификатор 'MID' должен быть объявлен

ORA-06550: Строка 2, столбец 5:

PL/SQL: Statement ignored

Здесь на лицо нарушение области видимости для оператора **GOTO**. Внутри блока переменная **<<MID>>** для него не доступна, что и приводит к ошибке компиляции. То есть внешний блок не имеет информации о содержимом вложенного блока. Так же нельзя переходить из одного блока в другой:

```
BEGIN
BEGIN
      GOTO OTHER;
END;

BEGIN
      <<OTHER>>
      NULL;
END;
END;
/
```

Получаем:

```
SQL> BEGIN
2      BEGIN
3          GOTO OTHER;
4      END;
5
6      BEGIN
7          <<OTHER>>
8          NULL;
9      END;
10 END;
11 /
      GOTO OTHER;
      *
```

ошибка в строке 3:

ORA-06550: Строка 3, столбец 11:

PLS-00201: идентификатор 'OTHER' должен быть объявлен

ORA-06550: Строка 3, столбец 6:

PL/SQL: Statement ignored

Что собственно и требовалось доказать. Так же нельзя производить переход из предшествующих блоков, условных операторов в последующий блок. Вот так:

```
BEGIN
IF (TRUE) THEN
    GOTO OTHER;
ELSE
    <<OTHER>>
    NULL;
END IF;
END;
/
```

Получаем:

```
SQL> BEGIN
2      IF (TRUE) THEN
3          GOTO OTHER;
4      ELSE
5          <<OTHER>>
6          NULL;
7      END IF;
8  END;
9  /
    GOTO OTHER;
    *
```

ошибка в строке 3:

ORA-06550: Строка 3, столбец 6:

PLS-00375: недопустимый оператор GOTO; этот GOTO не может выполнять переход к метке 'OTHER'

ORA-06550: Строка 4, столбец 3:

PL/SQL: Statement ignored

Хорошо видно, что блок выполнен с ошибкой компиляции. Так же остерегайтесь вот таких конструкций:

```
BEGIN
<<INFINITE_LOOP>>
GOTO INFINITE_LOOP
END;
/
```

PL/SQL не распознает бесконечные циклы и по этому не пытайтесь выполнить данный код! Иначе ваш администратор БД, может открутить вам голову, за такие фокусы! :) Придется вручную удалять сеанс **PL/SQL** и вам придется немного попотеть!!! Вот так! Еще один не маловажный аспект, для того, чтобы писать читаемый код, как правило необходимо использовать комментарии. В **PL/SQL** есть два способа определения комментариев в коде. Первое это:

-- Это строка комментария!

И второй это как в языке **ANSI C**:

/* Это комментарий */

Как правило, такого типа определений вполне достаточно. Остальное остается на ваше усмотрение. Теперь давайте рассмотрим такой вопрос, ранее мы определяли в наших пробных программах переменные различных типов. Давайте не спеша рассмотрим основные из них для того, чтобы ясно представлять ху из ... :) Основные типы, которые вы будете использовать такие:

- **CHAR**
- **VARCHAR2**
- **NUMBER**
- **DATE**

А вот их основные определения:

Тип	Подтип	Описание	Ограничение значений	Ограничения для БД
CHAR	CHARACTER STRING ROWID NCHAR	Символьные строки фиксированной длины	Возможный размер 0-32767 байт; по умолчанию 1	255 байт
VARCHAR2	VARCHAR STRING+ NVARCHAR	Символьные строки переменной длины	Возможный размер 0-32767 байт; по умолчанию 1	4000 байт
NUMBER(p,s)	NUMERIC DEC DECIMAL INT INTEGER FLOAT REAL DOUBLE PRECISION SMALLINT	Упакованные десятичные значения. p - общее число цифр. s - масштаб.	Диапазон: 1.0E-129 - 9.99E125 Точность: 1-38 (по умолчанию максимальное значение поддерживаемое системой) Масштаб: -84-127 (по умолчанию 0)	То же что и для PL/SQL
DATE		Внутреннее представление даты.	1 января 4712 года до н.э. время в секундах начиная с полуночи (по умолчанию 0:00)	То же что и для PL/SQL

Вот собственно описания основных типов данных, которые понадобятся вам в ближайшем будущем. Зная их описание и внутреннее устройство вы легко справитесь с любой заданной задачей. Это, конечно же, не все типы данных и данных БД, но остальные мы рассмотрим по ходу нашего повествования. А пока запомните эти описания.

Шаг 45 - PL/SQL - DDL, DML и еще кое что

Замечательно, мы знаем уже почти все типы данных, умеем делать исполняемые блоки и много уже чего. Здорово!

Пришло наконец время поближе познакомиться с такими понятиями как язык манипулирования данными - **DML (Data manipulation language)** и языком определения данных - **DDL (Data definition language)**. Все эти понятия лежат в контексте **SQL** внутри **PL/SQL**. Так как **PL/SQL** является расширением стандартного **SQL**, то собственно все становится ясно. Из всех операторов **SQL** в программах на **PL/SQL** можно использовать только **DML** операторы, сразу скажу, что к ним относятся **SELECT, INSERT, UPDATE, DELETE**. Операторы **DDL** за редким исключением использовать нельзя! Для того, чтобы объяснить смысл этих ограничений рассмотрим принцип создания программ **PL/SQL**.

В любом языке программирования, есть такое понятие как привязка переменных. Она может быть ранней или поздней. Привязка (**binding**) - переменной, это процесс указания области памяти, соответствующий идентификатору этой переменной. То есть, определен - указана область памяти, где находится определение. Так же **PL/SQL**, в процесс привязки входит (вот здесь внимательно!) - так же проверка БД на наличие полномочий, позволяющих обращаться к объектам схем (пользователей)! Так как это не просто программирование, а работа в сложном и взаимосвязанном комплексе хранилища данных! :) В языке где используется ранняя привязка (**early binding**) этот процесс осуществляется на этапе компиляции программы, а в языке, который использует позднюю привязку (**late binding**) она откладывается до этапа выполнения программы. Ранняя привязка означает, что компиляция программы будет занимать большее количество времени. (так как при этом нужно привязывать переменные), однако выполнятся такая программа будет быстрее! Действительно, привязка завершена. Поздняя привязка сокращает время компиляции, но увеличивает время выполнения программы. При создании **PL/SQL** было определено, что в нем будет использоваться ранняя привязка переменных, так как все блоки хранятся в памяти (**SGA**) и вызываются и выполняются максимально быстро! Именно по этому операторы **DDL** использовать нельзя!!! Оператор **DDL** модифицирует объект базы данных, следовательно полномочия на объект должны быть подтверждены вновь. Процесс подтверждения полномочий требует привязки идентификаторов, а это уже было сделано во время компиляции! Вот таким образом строится этот процесс. Наверное, вы уже устали от чтения, давайте приведем пример:

```
BEGIN
```

```
CREATE TABLE t_tbl(fld1 VARCHAR2(128));
```

```
INSERT INTO t_tbl(fld1)
VALUES('Hello World!!!');
```

```
END;
/
```

```
SQL> BEGIN
2
3 CREATE TABLE t_tbl(fld1 VARCHAR2(128));
4
5 INSERT INTO t_tbl(fld1)
6 VALUES('Hello World!!!');
7
8 END;
```

```
9 /
CREATE TABLE t_tbl(fld1 VARCHAR2(128));
*
ошибка в строке 3:
ORA-06550: Строка 3, столбец 2:
PLS-00103: Встретился символ "CREATE" в то время как ожидалось одно из
следующих:
begin case declare exit for goto if loop mod null pragma
raise return select update while with
<<
close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge
pipe
```

Наша программа не отработала должным образом, так как такие операторы не применимы в данном случае. Итак, давайте чуть подробнее рассмотрим основные операторы **DDL (Data definition language)**. Как вы уже наверное поняли, к ним прежде всего относятся оператор создания таблиц - **CREATE TABLE**, так же оператор **DROP TABLE**, **CREATE VIEW**, **DROP VIEW**, **CREATE PROCEDURE**, **DROP PROCEDURE** и т.д. Все это операторы создания объектов БД. Их еще много и рассмотрим мы их с вами по мере изучения материала. Главное, чтобы вы имели представление какая группа операторов к ним относится.

Для начала давайте разберемся с оператором **DDL - CREATE TABLE**. Собственно он, как не трудно догадаться, используется для создания самого основного объекта БД - таблиц. Общий синтаксис оператора таков:

```
----- CREATE TABLE -- СХЕМА -- . -- ИМЯ ТАБЛИЦЫ ---- (
----- FIELD1 TYPE FIELD,
----- FIELD2 TYPE FIELD,
      :
      :
      :
----- FIELDn TYPE FIELD
----- );
```

Сразу смею вас заверить, что это очень упрощенная схема, например, полное описание оператора **CREATE TABLE**, в руководстве администратора сервера **Oracle**, занимает три с половиной листа убоистого текста. Плюс подробное описание каждой секции, но сейчас это не столь важно. В дальнейшем вы еще успеете познакомиться с оператором **DDL**, создания таблицы в полном виде. Но для начала, этого достаточно. В [ware 24](#) мы уже создавали таблицу **PEOPLE**, если мне не изменяет память, вот такого вида:

```
CREATE TABLE PEOPLE
( ID NUMBER,
  NM VARCHAR2(50),
  FAMIL VARCHAR2(50),
  OTCH VARCHAR2(50),
  DROG DATE
)
/
```

И даже заполняли ее тремя записями, по моему вот так:

```
INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
              VALUES(1, 'John', 'Godwin', 'Petrovich', TO_DATE('03-12-1967','DD-MM-YYYY'))
/

INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
              VALUES(2, 'Bob', 'Doris', 'Martovich', TO_DATE('01-02-1960','DD-MM-YYYY'))
/

INSERT INTO PEOPLE(ID, NM, FAMIL, OTCH, DROG)
              VALUES(3, 'Frank', 'Black', 'Milleniumich', TO_DATE('03-07-1953','DD-MM-YYYY'))
/

COMMIT
/
```

Здесь, я применил внутреннюю функцию **TO_DATE**, которая преобразует строку в дату по шаблону. Но там я не сказал, как эту таблицу можно не только создать, но и удалить. Оператор **DML** - **INSERT**, мы подробнее рассмотрим в следующий раз, а пока давайте поучимся создавать и удалять таблицы БД. Сначала создадим новую таблицу. Введите в **SQL*Plus** оператор создания таблицы **NEWPEOPLE**:

```
CREATE TABLE NEWPEOPLE
(
  NM VARCHAR2(50),
  FAMIL VARCHAR2(50),
  OTCH VARCHAR2(50)
)
/

COMMIT
/
```

В результате получаем:

```
SQL> CREATE TABLE NEWPEOPLE
2 (
3  NM VARCHAR2(50),
4  FAMIL VARCHAR2(50),
5  OTCH VARCHAR2(50)
6 )
7 /
```

Таблица создана.

```
SQL>
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Итак, новая табличка создана с применением оператора **DDL** - **CREATE TABLE**. Давайте убедимся, что она действительно создана.

Дадим вот такой запрос:

```
SELECT a.OBJECT_NAME, a.OBJECT_TYPE, a.STATUS FROM USER_OBJECTS a
WHERE a.OBJECT_TYPE = 'TABLE'
/
```

Получаем:

```
SQL> SELECT a.OBJECT_NAME, a.OBJECT_TYPE, a.STATUS FROM USER_OBJECTS a
2  WHERE a.OBJECT_TYPE = 'TABLE'
3  /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
CUSTOMERS	TABLE	VALID
NEWPEOPLE	TABLE	VALID
OFFICES	TABLE	VALID
ORDERS	TABLE	VALID
PEOPLE	TABLE	VALID
PRODUCTS	TABLE	VALID
SALESREPS	TABLE	VALID

7 строк выбрано.

Хорошо видно, что обе таблицы **PEOPLE** и **NEWPEOPLE** находятся в схеме **MILLER** и в исправном состоянии, о чем свидетельствует запись **VALID** в поле **STATUS**. Что ж, теперь давайте удалим обе эти таблицы, так как пока они нам больше не нужны. Для удаления таблиц воспользуемся оператором языка **DDL** - **DROP TABLE**:

```
DROP TABLE PEOPLE
/
```

```
DROP TABLE NEWPEOPLE
/
```

```
COMMIT
/
```

После ввода в **SQL*Plus** получаем:

```
SQL> DROP TABLE PEOPLE
2  /
```

Таблица удалена.

```
SQL> DROP TABLE NEWPEOPLE
2  /
```

Таблица удалена.

```
SQL>
SQL> COMMIT
```

2 /

Фиксация обновлений завершена.

Вот и все! Таблицы удалены! Но с оператором **DROP TABLE** советую обращаться осторожнее, не удалите по запарке какую-нибудь важную таблицу! В крайнем случае, воспользуйтесь откатом транзакции, оператор **ROLLBACK**! Вот, теперь мы умеем создавать и удалять таблицы в схеме БД. Можете пробовать, только пока не удаляйте наши учебные таблички, он нам еще понадобятся!!! :)

Шаг 46 - PL/SQL - DML, оператор INSERT

Итак, приступим к более детальному разбору операторов **DML**. Без применения этих операторов было бы вообще бессмысленна вся эта затея с серверами БД. Начать лучше всего с оператора **DML** - **INSERT**. Оператор **INSERT**, служит для заполнения таблиц БД данными и является достаточно простым в использовании. Давайте посмотрим, на синтаксис оператора **INSERT**:

```
----- INSERT INTO --- таблица -----  
----- (имя столбца таблицы) -----  
----- , -----  
----- VALUES --- (выражение); -----  
----- оператор выбора -----
```

Я думаю, понятно, что "таблица" - это имя таблицы, куда вводятся данные. "имя столбца" - это список столбцов, в которые вводятся данные. "Выражение" - это собственно сами данные. "Оператор выбора" - это предложение **SELECT**, для заполнения таблицы. Используется без части **VALUES**. Давайте удалим все таблицы, которые вы создавали в прошлый раз, так как сейчас мы будем рассматривать их новый вариант. Удалять таблицы вы уже умеете. Если вы не сделали этого в прошлый раз то просто введите:

```
DROP TABLE PEOPLE  
/  
  
DROP TABLE NEWPEOPLE  
/  
  
COMMIT  
/
```

Затем создаем таблицу **PEOPLE** с новыми параметрами:

```
CREATE TABLE PEOPLE  
(  
  ID NUMBER PRIMARY KEY,  
  NM VARCHAR2(50),  
  FM VARCHAR2(50),  
  OT VARCHAR2(50)  
)  
/  
  
COMMIT  
/
```

Получаем:

```
SQL> CREATE TABLE PEOPLE  
2 (  
3   ID NUMBER PRIMARY KEY,  
4   NM VARCHAR2(50),  
5   FM VARCHAR2(50),  
6   OT VARCHAR2(50)
```

```
7 )  
8 /
```

Таблица создана.

```
SQL> COMMIT  
2 /
```

Фиксация обновлений завершена.

Обратите внимание на поле **ID** таблицы **PEOPLE**! Оно имеет атрибут **PRIMARY KEY**, то есть является первичным ключом таблицы. В него можно записывать только отличные друг от друга значения. И это поле не принимает значения типа **NULL**! Теперь, применив оператор **INSERT**, согласно его синтаксического разбора введем в таблицу шесть значений:

```
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(1, 'John', 'Godwin', 'Petrovich')  
/  
  
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(2, 'Bob', 'Doris', 'Martovich')  
/  
  
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(3, 'Frank', 'Black', 'Milleniumich')  
/  
  
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(4, 'Pupkin', 'Misha', 'Semenovich')  
/  
  
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(5, 'Pistoletov', 'Makar', 'Patronovich')  
/  
  
INSERT INTO PEOPLE(ID, NM, FM, OT)  
VALUES(6, 'Avtomatov', 'Kolya', 'Pricelovich')  
/  
  
COMMIT  
/
```

Получаем:

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)  
2 VALUES(1, 'John', 'Godwin', 'Petrovich')  
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
```

```
2 VALUES(2, 'Bob', 'Doris', 'Martovich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2 VALUES(3, 'Frank', 'Black', 'Milleniumich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2 VALUES(4, 'Pupkin', 'Misha', 'Semenovich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2 VALUES(5, 'Pistoletov', 'Makar', 'Patronovich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2 VALUES(6, 'Avtomatov', 'Kolya', 'Pricelovich')
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

В части **INTO PEOPLE(ID, NM, FM, OT)** оператора **INSERT** указаны поля в порядке ввода данных, если есть необходимость порядок следования полей можно изменить или вообще исключить некоторые из них. В части **VALUES(6, 'Avtomatov', 'Kolya', 'Pricelovich')** оператора **INSERT** указаны собственно данные для ввода. Вот так работает оператор **INSERT**.

Я думаю, ничего сложного в этом нет. В результате таблица **PEOPLE** получила шесть записей. А что, если у вас есть данные, которые быстро нужно загрузить не путая с тем, что уже было в таблице. Для этого можно применить оператор **INSERT** с выражением **SELECT**. Создадим промежуточную таблицу и перебросим содержимое основной в нее:

```
CREATE TABLE OLD_PEOPLE
(
  ID NUMBER PRIMARY KEY,
  NM VARCHAR2(50),
  FM VARCHAR2(50),
  OT VARCHAR2(50)
)
```

```
COMMIT  
/
```

Получаем: SQL> CREATE TABLE OLD_PEOPLE 2 (3 ID NUMBER PRIMARY KEY, 4 NM VARCHAR2(50), 5 FM VARCHAR2(50), 6 OT VARCHAR2(50) 7) 8 / Таблица создана. SQL> COMMIT 2 / Фиксация обновлений завершена.

Теперь применяя **INSERT** с выражением, перегрузим данные в новую таблицу:

```
INSERT INTO OLD_PEOPLE  
  SELECT * FROM PEOPLE  
/
```

Получаем:

```
SQL> INSERT INTO OLD_PEOPLE  
  2  SELECT * FROM PEOPLE  
  3  /
```

6 строк создано.

```
SQL> COMMIT  
  2  /
```

Фиксация обновлений завершена.

Теперь в обеих таблицах одинаковые данные. Вот таким образом можно, применяя оператор **SELECT** со всей его мощностью, получать результаты разнообразных запросов. Но при этом количество, имена и типы данных обеих таблиц должны быть одинаковы!!! Например, таким оператором я часто пользуюсь при сливании таблиц телефонных номеров. Очень удобно. Таким образом, мы рассмотрели первый из трех основных операторов, **DML** - **INSERT**. Пока не удаляйте таблицы **PEOPLE** и **OLD_PEOPLE**, в следующем шаге они нам еще понадобятся. Пока можете поработать с тем материалом, который рассмотрели сейчас.

Шаг 47 - PL/SQL - DML, оператор DELETE

Наполнять таблицу данными задача для нас не такая сложная. А вот теперь, если какие-либо данные нужно, скажем, удалить из таблицы? Вопрос достаточно тривиален. Для этого в языке **DML** - есть оператор **DELETE**. Вот так выглядит его синтаксис:

```
----- DELETE FROM --- таблица --- * псевдоним * -----  
----- (имя столбца таблицы) -----  
----- WHERE ----- условие -----
```

Как мы видим в операторе **DELETE** так же есть часть определения условия **WHERE**. То есть все, что мы использовали ранее для описания работы **WHERE** в операторе **SELECT**, применимо и здесь. Допустим нам нужно удалить запись с номером три. Записываем:

```
DELETE FROM PEOPLE  
  WHERE ID = 3  
/  
  
COMMIT  
/
```

Получаем:

```
SQL> DELETE FROM PEOPLE  
  2  WHERE ID = 3  
  3  /
```

1 строка удалена.

```
SQL> COMMIT  
  2  /
```

Фиксация обновлений завершена.

Оператор **DML** - **DELETE** как видим совершенно безукоризненно выполнил свою работу. Теперь давайте удалим все из таблицы **PEOPLE**. Записываем:

```
DELETE FROM PEOPLE  
/  
  
COMMIT  
/
```

Получаем:

```
SQL> DELETE FROM PEOPLE  
  2  /
```

5 строк удалено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Теперь из таблицы **PEOPLE** удалены все записи, которые мы вводили ранее. Да! Давайте еще раз вернемся к **NULL** для того, чтобы посмотреть как он ведет себя здесь. Введем уже знакомым нам с вам способом, три записи в таблицу **PEOPLE** со значениями **NULL**:

```
INSERT INTO PEOPLE(ID, NM, FM, OT)
      VALUES(1, NULL, 'Godwin', 'Petrovich')
/

INSERT INTO PEOPLE(ID, NM, FM, OT)
      VALUES(2, 'Bob', NULL, 'Martovich')
/

INSERT INTO PEOPLE(ID, NM, FM, OT)
      VALUES(3, 'Frank', 'Black', NULL)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2   VALUES(1, NULL, 'Godwin', 'Petrovich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2   VALUES(2, 'Bob', NULL, 'Martovich')
3 /
```

1 строка создана.

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2   VALUES(3, 'Frank', 'Black', NULL)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Теперь в таблице **PEOPLE** в трех разных записях присутствует значение **NULL**. Допустим вы хотите удалить записи со значением **NULL** в поле **NM** таблицы **PEOPLE**. Тогда записываем:

```
DELETE FROM PEOPLE  
  WHERE NM IS NULL  
/
```

```
COMMIT  
/
```

Получаем:

```
SQL> DELETE FROM PEOPLE  
  2  WHERE NM IS NULL  
  3  /
```

1 строка удалена.

```
SQL> COMMIT  
  2  /
```

Фиксация обновлений завершена.

Запись такого типа одна и она удалена оператором **DELETE**. Для закрепления пройденного давайте сделаем следующее. Очистим таблицу **OLD_PEOPLE** и заполним ее остатками записей в таблице **PEOPLE**. Записываем:

```
DELETE FROM OLD_PEOPLE  
/
```

```
COMMIT  
/
```

Получаем:

```
SQL> DELETE FROM OLD_PEOPLE  
  2  /
```

6 строк удалено.

```
SQL> COMMIT  
  2  /
```

Фиксация обновлений завершена.

Таблица **OLD_PEOPLE** очищена и пуста. Вводим далее такое выражение:

```
INSERT INTO OLD_PEOPLE  
  SELECT * FROM PEOPLE  
/
```

```
COMMIT  
/
```

Получаем:

```
SQL> INSERT INTO OLD_PEOPLE  
2  SELECT * FROM PEOPLE  
3  /
```

2 строк создано.

```
SQL> COMMIT  
2  /
```

Фиксация обновлений завершена.

Вот таким образом в обоих наших тренировочных табличках на текущий момент по две одинаковых записи. Вот так достаточно просто работает оператор **DELETE**. Можете попробовать еще раз все с самого начала для закрепления! :)

Шаг 48 - PL/SQL - DML, оператор UPDATE

Что ж! Осталась одна операция с таблицами БД, которую мы с вами еще не рассматривали. Конечно же, изменение данных. Данные в таблицах БД изменяются с помощью оператора **DML - UPDATE**. Синтаксис этого оператора таков:

```
----- UPDATE ----- таблица ----- SET -----  
----- имя столбца -- = -- выражение -----  
----- WHERE ----- условие -----
```

Синтаксически все довольно просто, как видим, в операторе **UPDATE** так же присутствует предложение **WHERE**, а следовательно, применимы все правила получения результирующего набора. Давайте посмотрим на практике, что можно сделать с таблицей с помощью оператора **DML - UPDATE**. Например, введите примерно следующее:

```
UPDATE PEOPLE  
  SET NM = 'IVAN'  
/  
  
COMMIT  
/
```

Получаем:

```
SQL> UPDATE PEOPLE  
  2  SET NM = 'IVAN'  
  3  /
```

2 строк обновлено.

```
SQL> COMMIT  
  2  /
```

Фиксация обновлений завершена.

А, вот теперь давайте уже не вслепую как в прошлый раз, а уже на деле посмотрим, что же произошло с таблицей:

```
SELECT * FROM PEOPLE  
/
```

Вывод:

```
SQL> SELECT * FROM PEOPLE  
  2  /
```

	ID NM	FM	OT
2	IVAN	NULL	Martovich
3	IVAN	Black	NULL

Ух ты! Неожиданный ход, не так ли? Вроде бы мы ввели одно значение в операторе. А изменились все записи, хоть их всего то и две! Да, именно так и должно быть. Ведь мы не указали **UPDATE** в условии какие записи менять, а какие нет! Вот и получите результат, он прошелся по всей таблице от первой до последней записи и изменил их значение на **IVAN**! Теперь давайте исправим ситуацию и применим трюк с **NULL** в предложении **WHERE**! Вот так:

```
UPDATE PEOPLE
  SET FM = 'Jason'
WHERE FM IS NULL
/

COMMIT
/

SELECT * FROM PEOPLE
/
```

Получаем следующее:

```
SQL> UPDATE PEOPLE
2  SET FM = 'Jason'
3  WHERE FM IS NULL
4  /
```

1 строка обновлена.

```
SQL> COMMIT
2  /
```

Фиксация обновлений завершена.

```
SQL> SELECT * FROM PEOPLE
2  /
```

ID	NM	FM	OT
2	IVAN	Jason	Martovich
3	IVAN	Black	NULL

Изменилась тоько одна запись, а теперь давайте вернем **Jason**-у его старое имя:

```
UPDATE PEOPLE
  SET NM = 'Bob'
WHERE FM = 'Jason'
/

COMMIT
/

SELECT * FROM PEOPLE
/
```

Смотрим, что получилось:

```
SQL> UPDATE PEOPLE
2  SET NM = 'Bob'
3  WHERE FM = 'Jason'
4  /
```

1 строка обновлена.

```
SQL> COMMIT
2  /
```

Фиксация обновлений завершена.

```
SQL> SELECT * FROM PEOPLE
2  /
```

ID	NM	FM	OT
2	Bob	Jason	Martovich
3	IVAN	Black	NULL

Вот теперь **Bob Jason** стал похож на человека! Всего-то мы применили предложение **WHERE**! Теперь давайте добавим запись с **NULL** значениями, то есть пустую:

```
INSERT INTO PEOPLE(ID, NM, FM, OT)
VALUES(5, NULL, NULL, NULL)
/

COMMIT
/

SELECT * FROM PEOPLE
/
```

Вывод:

```
SQL> INSERT INTO PEOPLE(ID, NM, FM, OT)
2  VALUES(5, NULL, NULL, NULL)
3  /
```

1 строка создана.

```
SQL> COMMIT
2  /
```

Фиксация обновлений завершена.

```
SQL> SELECT * FROM PEOPLE
2  /
```

ID	NM	FM	OT
----	----	----	----

```
-----
5 NULL  NULL  NULL
2 Bob   Jason  Martovich
3 IVAN  Black  NULL
```

Есть пустая запись. А вот теперь давайте изменим ее по всем правилам с применением значения ключевого (уникального) поля **ID** таблицы **PEOPLE**:

```
UPDATE PEOPLE
  SET NM = 'Irvin',
      FM = 'Show',
      OT = 'Brefovich'
WHERE ID = 5
/

COMMIT
/

SELECT * FROM PEOPLE
/
```

Получаем результат работы:

```
SQL> UPDATE PEOPLE
2  SET NM = 'Irvin',
3  FM = 'Show',
4  OT = 'Brefovich'
5  WHERE ID = 5
6  /
```

1 строка обновлена.

```
SQL> COMMIT
2  /
```

Фиксация обновлений завершена.

```
SQL> SELECT * FROM PEOPLE
2  /
```

```
-----
ID NM   FM      OT
-----
5 Irvin Show  Brefovich
2 Bob   Jason  Martovich
3 IVAN  Black  NULL
```

Вот смотрите внимательно, так работает внутренняя структура почти всех клиентских приложений, берется значение ключевого поля и изменяется именно эта конкретная запись! Теперь давайте изменим значение самого ключа поля применив математическое выражение в части **SET** оператора **UPDATE**:

```
UPDATE PEOPLE
```

```
    SET ID = 5+2
WHERE ID = 5
/

COMMIT
/

SELECT * FROM PEOPLE
/
```

Получаем:

```
SQL> UPDATE PEOPLE
2  SET ID = 5+2
3  WHERE ID = 5
4  /
```

1 строка обновлена.

```
SQL> COMMIT
2  /
```

Фиксация обновлений завершена.

```
SQL> SELECT * FROM PEOPLE
2  /
```

ID	NM	FM	OT
7	Irvin	Show	Brefovich
2	Bob	Jason	Martovich
3	IVAN	Black	NULL

Теперь, как мы видим, значение ключевого поля изменилось с 5 на 7. А, остальное не изменялось. Главное, чтобы это значение не совпало ни с одним из уже существующих в таблице **PEOPLE**! Вот собственно так работает оператор **UPDATE**. В итоге мы с вами рассмотрели три основных рабочих лошадки, обработки и изменения таблиц БД! :)

Шаг 49 - PL/SQL - виды и типы циклов

На сегодня вы уже многое знаете о **PL/SQL**. Но очень многое еще впереди. Как и во всех языках программирования, в **PL/SQL** имеются операторы циклов. Их три основных типа:

1. Безусловные циклы (выполняемые бесконечно)
2. Интерактивные циклы (**FOR**)
3. Условные циклы (**WHILE**)

Самый простой тип цикла в языке **PL/SQL** таков:

```
LOOP
NULL;
END LOOP
/
```

Но использовать такой цикл нет смысла, да и для экземпляра БД это не безопасно. Для этого необходимо использовать определенные пути выхода из циклов. Их то же три:

1. **EXIT** - Безусловный выход из цикла. Используется посредством применения оператора **IF**.
2. **EXIT WHEN** - Выход при выполнении условия.
3. **GOTO** - Выход из цикла во внешний контекст.

Давайте рассмотрим пример с применением цикла **LOOP EXIT WHEN**. Запишем следующее:

```
DECLARE
i NUMBER := 0;

BEGIN

LOOP                                -- start loop 1
i := i + 1;
IF (i >= 100) THEN
    i := 0;
    EXIT;                            -- exit when i >= 0
END IF;
END LOOP;                            -- end loop 1

LOOP                                -- start loop 2
i := i + 1;
EXIT WHEN (i >= 100);               -- exit when i >= 0
END LOOP;                            -- end loop 2-----

END;
/
```

Получаем после исполнения:

```
SQL> DECLARE
2  i NUMBER := 0;
3
```



```
4 BEGIN
5
6 LOOP
7   i := i + 1;
8   IF (i >= 100) THEN
9     i := 0;
10    EXIT;
11  END IF;
12 END LOOP;
13
14 LOOP
15   i := i + 1;
16   EXIT WHEN (i >= 100);
17 END LOOP;
18
19 END;
20 /
```

Процедура PL/SQL успешно завершена.

Видимых действий не было, но и ошибок то же! Первый цикл закончился после того как **i** стало равно 10. При этом оно получило значение 0 и произошел выход из цикла. Второй цикл применил вложенное предложение **EXIT WHEN**, что является более верным его использованием синтаксически. Тем не менее, применение условных операторов предполагает перед выходом из цикла проделать некоторые действия. Цикл **LOOP EXIT WHEN END LOOP** в последующем будет самым, часто используемым при построении конструкций курсоров. Рассмотрим еще одну разновидность вышеприведенного цикла:

```
DECLARE
  k NUMBER := 0;

BEGIN

  WHILE (k < 10) LOOP
    k := k + 1;
  END LOOP;

END;
/
```

Получаем:

```
SQL> DECLARE
2   k NUMBER := 0;
3
4 BEGIN
5
6   WHILE (k < 10) LOOP
7     k := k + 1;
8   END LOOP;
9
10  END;
```

11 /

Процедура PL/SQL успешно завершена.

Здесь в отличие от предыдущего цикла, действия выполняются до тех пор пока условие истинно. Если условие ложно, то цикл прекращается. Что хорошо видно из приведенного примера. В **PL/SQL** в конструкциях циклов нет такого, иногда полезного, оператора как **CONTINUE**, вследствие того, что выражение **CONTINUE** зарезервировано языком **PL/SQL** и используется для других целей. Но такую конструкцию как **CONTINUE** можно эмулировать, применив цикл вида **LOOP EXIT WHEN END LOOP** и используя весьма не популярный, но в данном случае очень полезный оператор **GOTO**!

Запишем следующее, выведем все нечетные числа до 20:

```
SET SERVEROUTPUT ON

DECLARE
s NUMBER := 0;

BEGIN

DBMS_OUTPUT.enable;

LOOP
IF(MOD(s, 2) = 1) THEN
GOTO LESS;
END IF;
DBMS_OUTPUT.put_line(TO_CHAR(s)||' is even!');
<<LESS>>
EXIT WHEN (s = 20);
s := s + 1;
END LOOP;

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2  s NUMBER := 0;
3
4  BEGIN
5
6  DBMS_OUTPUT.enable;
7
8  LOOP
9  IF(MOD(s, 2) = 1) THEN
10   GOTO LESS;
11  END IF;
12  DBMS_OUTPUT.put_line(TO_CHAR(s)||' is even!');
13  <<LESS>>
```

```
14 EXIT WHEN (s = 20);
15   s := s + 1;
16 END LOOP;
17
18 END;
19 /
0 is even!
2 is even!
4 is even!
6 is even!
8 is even!
10 is even!
12 is even!
14 is even!
16 is even!
18 is even!
20 is even!
0 is even!
2 is even!
4 is even!
6 is even!
8 is even!
10 is even!
12 is even!
14 is even!
16 is even!
18 is even!
20 is even!
```

Процедура PL/SQL успешно завершена.

В результате применения **GOTO** в теле цикла получаем не сложную и понятную логику работы. Теперь давайте рассмотрим, не менее полезный и очень популярный в **PL/SQL** цикл **FOR**. Он к стати очень удобен при работе с курсорами, но об этом чуть позднее. Запишем следующее:

```
BEGIN
FOR i IN 1..100 LOOP
NULL;
END LOOP;
END;
/

SQL> BEGIN
2  FOR i IN 1..100 LOOP
3  NULL;
4  END LOOP;
5  END;
6  /
```

Процедура PL/SQL успешно завершена.

В данный момент **FOR** успешно ничего не делал аж сто раз! Итак, давайте рассмотрим его чуть ближе, **IN** как и в операторе **SELECT** задает диапазон значений итерации цикла, а ".." это как

вы помните так называемый "оператор диапазона"! Вот так просто и ясно. Остальное уже знакомо. Замечу так же, что переменная цикла **i** является переменной только для чтения. По этому шаг цикла **FOR** изменить принудительно нельзя! Если это необходимо, то лучше применять цикл вида **LOOP EXIT WHEN END LOOP!** :) Теперь давайте поработаем с числами:

```
DECLARE
s NUMBER := 0;

BEGIN
DBMS_OUTPUT.enable;
FOR i IN 1..20 LOOP
IF(MOD(i, 2) = 1) THEN
    DBMS_OUTPUT.put_line(TO_CHAR(i)||' is even!');
    s := i;
END IF;
END LOOP;
DBMS_OUTPUT.put_line('last odd number was '||TO_CHAR(s));
END;
/
```

Получаем:

```
SQL> DECLARE
2  s NUMBER := 0;
3
4  BEGIN
5  DBMS_OUTPUT.enable;
6  FOR i IN 1..20 LOOP
7  IF(MOD(i, 2) = 1) THEN
8  DBMS_OUTPUT.put_line(TO_CHAR(i)||' is even!');
9  s := i;
10 END IF;
11 END LOOP;
12 DBMS_OUTPUT.put_line('last odd number was '||TO_CHAR(s));
13 END;
14 /
1 is even!
3 is even!
5 is even!
7 is even!
9 is even!
11 is even!
13 is even!
15 is even!
17 is even!
19 is even!
last odd number was 19
```

Процедура PL/SQL успешно завершена.

Та же задачка, только с циклом **FOR**. Да, функция **MOD** возвращает остаток от деления чисел, как вы, наверное, уже догадались. Так же в операторе **FOR** есть возможность задавать обратный отсчет, ну, например, перед стартом или взрывом! :) Вот так:

```
BEGIN
  DBMS_OUTPUT.enable;
  FOR i IN REVERSE 1..10 LOOP
    DBMS_OUTPUT.put_line(TO_CHAR(i)||'-');
  END LOOP;
  DBMS_OUTPUT.put_line('Blastoff!');
END;
/
```

Получаем:

```
SQL> BEGIN
  2  DBMS_OUTPUT.enable;
  3  FOR i IN REVERSE 1..10 LOOP
  4  DBMS_OUTPUT.put_line(TO_CHAR(i)||'-');
  5  END LOOP;
  6  DBMS_OUTPUT.put_line('Blastoff!');
  7  END;
  8  /
10-
9-
8-
7-
6-
5-
4-
3-
2-
1-
Blastoff!
```

Процедура PL/SQL успешно завершена.

Нолика не было, но бабахнуло! Вот такой достаточно богатый набор операторов циклов в языке **PL/SQL**! Надеюсь, вы научитесь с легкостью их применять при построении серверных приложений ваших БД! :)

Шаг 50 - PL/SQL - переменные их типы и кое что еще!

Вот, наконец, мы достаточно много, как мне кажется, успели изучить, а я добрался таки до 50-го Шага! Юбилей! Ну да ладно. Давайте разберемся вот с чем. Наверное, как вы уже знаете основная единица, хранения информации в БД это таблица. При работе с таблицами в **PL/SQL**, обычно создается множество переменных, разных типов. В них вы храните какие-либо данные, как правило, из полей таблиц. Например, учебная таблица **OFFICES**, определенная с помощью оператора **CREATE TABLE** выглядит следующим образом:

```
CREATE TABLE OFFICES
(
  OFFICE INTEGER PRIMARY KEY,
  CITY VARCHAR2(30) NOT NULL,
  REGION VARCHAR2(30) NOT NULL,
  MGR INTEGER,
  TARGET NUMBER,
  SALES NUMBER NOT NULL
)
```

Допустим, вы будете хранить содержимое поля **REGION**, в переменной **REGION_MY** и поле **MGR** в переменной **MGR_MY**. Соответственно вы объявите эти переменные примерно следующим образом:

```
DECLARE

  REGION_MY VARCHAR2(30);
  MGR_MY INTEGER;
  .
  .
  .
```

Но допустим, в какой-то момент, вам нужно переопределить поля таблицы **OFFICES**, **MGR** и **REGION** таким образом:

```
.
.
REGION VARCHAR2(128) NOT NULL,
MGR NUMBER(10,5),
.
.
```

Следовательно, вы уже не сможете в дальнейшем хранить в этих переменных значения переопределенных полей таблицы **OFFICES**, так как их типы не совпадают и **PL/SQL** выдаст сообщение об ошибке! А, если таких переменных объявлено достаточно много, то разобраться и изменить все типы будет достаточно сложно и, как правило, вы наделаете кучу ошибок! Для того, чтобы избежать таких неприятностей в дальнейшем необходимо использовать конструкцию **%TYPE**, следующим образом:

```
----- VARIABLE TABLE.FIELD%TYPE -----
```

Или, если более понятно, то вышеуказанные переменные нужно объявить вот так:

```
DECLARE

    REGION_MY OFFICES.REGION%TYPE;
    MGR_MY OFFICES.MGR%TYPE;

BEGIN
    .
    .
    .
```

Конструкция получается более громоздкая но, за то точно и надежно! Теперь, если тип поля изменится, то изменится и тип всех переменных связанных с данным полем! Можно привести еще несколько примеров применения конструкции **%TYPE**:

```
DECLARE

    v_MYCITY OFFICES.CITY%TYPE;
    v_MYSALES OFFICES.SALES%TYPE;
    v_TEMPOLD NUMBER(7,4) NOT NULL;
    v_DOPTEMP v_TEMPOLD%TYPE;
```

Замечу так же, что для переменной **v_TEMPOLD** есть ограничение **NOT NULL**, а вот для переменной **v_DOPTEMP** это ограничение не переносится, так как применяется конструкция **%TYPE**, от типа другой переменной! Применение конструкции **%TYPE**, является хорошим стилем программирования, так как делает код более понятным и проще адаптируемым. Вот собственно, что касается, конструкции **%TYPE**. В **PL/SQL** имеется так же возможность объявлять так называемы подтипы (**subtype**). То есть типы на основе других, стандартных типов. Такое извращение, как правило применяется для того, чтобы получить более понятное описание типа. Ряд типов в **PL/SQL**, такие например как **INTEGER**, **DECIMAL** - являются подтипом, типа **NUMBER**! Делается это таким образом:

```
---- SUBTYPE --- новый тип -- IS -- исходный тип -----
```

В части исходный тип, можно применить любой стандартный тип или конструкцию **%TYPE**. Например:

```
DECLARE

    SUBTYPE t_LoopCn IS NUMBER;      -- Определяем новый тип
    v_LpCounter t_LoopCn;            -- Объявляем переменную с эти подтипом
```

Сразу замечу, что подтип нельзя ограничивать непосредственно в определении **SUBTYPE**. Это будет ошибкой:

```
DECLARE

    SUBTYPE t_LoopCn IS NUMBER(4);    -- Error !!!
    v_LpCounter t_LoopCn;            -- Объявляем переменную с эти подтипом
```

Но данное ограничение можно обойти, применив фиктивную переменную нужного типа (с ограничением) и в определении **SUBTYPE** воспользоваться конструкцией **%TYPE**:

```
DECLARE
```

```
    v_NullVar NUMBER(7,3);
```

```
    SUBTYPE t_LoopCn IS v_NullVar%TYPE;    -- Определяем новый тип
```

```
    v_LpCounter t_LoopCn;                  -- Объявляем переменную с эти подтипом
```

При объявлении переменной с неограниченным подтипом ее тип может быть ограничен, вот так:

```
DECLARE
```

```
    SUBTYPE t_NumVar IS NUMBER            -- Определяем неограниченный подтип,
```

```
    v_Temp t_NumVar(5);                  -- но ограничиваем переменную с этим подтипом!
```

При этом не забывайте, что подтип принадлежит к тому же семейству типов, что и базовый тип! Это важно! Вот такие тонкости в работе с типами данных, есть в **PL/SQL**! :) Закрепляйте знания и можете переписать пару блоков, из [прошлого шага](#) в свете работы с типами и подтипами данных! :) Удачи!

Шаг 51 - PL/SQL - КУРСОР, теория

Основной базис **PL/SQL**, как вы уже наверное заметили, мы с вами, можно сказать, получили. Пришло время начинать разбираться с краеугольным, так сказать, камнем языка **PL/SQL** - а, именно с таким понятием как "КУРСОР". Но, для начала давайте, рассмотрим несколько понятий, которые предваряют этот материал. Ранее я часто упоминал, такое слово как "транзакция". Мы еще не раз к ней будем возвращаться, так как это довольно объемная тема. Но, чтобы было проще для понимания, начнем вот с чего. Допустим, многие из Вас ходили в Сбербанк платить за квартиру, за талоны техосмотра, ну и еще много чего. По своей сути вы выполняли банковскую операцию платежа. В принципе это и есть не что иное, как транзакция (банковская в данном случае). Вы отдаете квитанцию девушке операционистке, то есть вы инициируете начало транзакции, девушка проделывает пляшущее движение пальчиками по клавиатуре то есть вводит в машину ваш код платежа, сумму и т.д. тем самым выполняет операцию отъема у вас энной суммы денег! :) Затем, мило улыбаясь, говорит сколько вы должны выложить денежек и так же мило улыбаясь, берет их у вас и отсчитывает сдачу, если таковая должна иметь место. То есть происходит процесс обмена, данными (в нашем случае квитанция - деньги). Затем она быстренько разбрасывает отполовиненные квитанции по ячейкам, а вторые половинки отдает вам. Все, транзакция завершена, оплата прошла. Примерно тоже самое происходит внутри сервера **Oracle** когда вы подключаетесь к нему скажем при помощи **SQL*Plus**. К чему я все это тут вам рассказываю, а к тому что - понятие механизма обмена данными между клиентом и сервером БД, достаточно хорошо просматривается на примере работы курсоров. По мере их изучения, мы понемногу подберемся к теме, определения экземпляров БД и еще многим, достаточно сложным понятиям. Пока это описание банковского платежа, пусть отложится у вас в памяти, оно еще нам потребуется! :) Проводя аналогии с реальными ситуациями, иногда достаточно просто разобраться со сложными процессами, которые происходят внутри БД, что я вам еще не раз продемонстрирую. Итак, в прошлый раз мы изучили оператор **%TYPE**. Надеюсь, вы поняли, что это за оператор и что он позволяет выполнять. Так же в **PL/SQL** для удобства работы с данными имеется еще один очень полезный и наиболее часто применимый оператор **%ROWTYPE**! Он способен проделывать, очень интересный фокус. Посмотрим, как он работает:

----- переменная - структура данных%ROWTYPE -----

Запишем такой пример:

```
DECLARE
```

```
    v_RecOffices OFFICES%ROWTYPE;
```

```
    .  
    .  
    .
```

При этом переменная **v_RecOffices** будет иметь вот такую внутреннюю структуру:

```
v_RecOffices(  
  OFFICE INTEGER  
  CITY VARCHAR2(30),  
  REGION VARCHAR2(30),  
  MGR INTEGER,  
  TARGET NUMBER,  
  SALES NUMBER
```

)

Так как мы объявили переменную **v_RecOffices** на базе таблицы **OFFICES**, то в результате получаем "запись" с внутренней структурой идентичной полям таблицы **OFFICES**. То есть в своей сути **v_RecOffices** это не просто переменная, а переменная типа "запись" и она содержит в нашем случае как бы шесть переменных, типы которых, совпадают с типами полей таблицы **OFFICES**! Теперь надеюсь ясно, что за один раз в такую, переменную, можно будет поместить целую строку из таблицы **OFFICES** и, если какое либо поле в таблице будет изменено в процессе сопровождения кода, то менять что-то в коде нет необходимости, так как тип **%ROWTYPE**, будет сам реагировать на эти изменения. Так же есть возможность обращаться к любому полю записи **v_RecOffices**. Такое обращение производится через точечную нотацию (как в языке **Pascal**):

DECLARE

```
a INTEGER := v_RecOffices.OFFICE;  
b VARCHAR2(50) := v_RecOffices.CITY;  
c INTEGER := v_RecOffices.MGR;
```

То есть, после выборки строки из таблицы очень легко получить отдельные значения полей в промежуточные переменные. За один раз можно выбрать только одну строку в переменную данного типа! Так же как и для **%TYPE** ограничение **NOT NULL** для полей записи отсутствует. В дальнейшем вы увидите как **%ROWTYPE** значительно облегчает последовательную выборку данных при работе с курсорами.

Шаг 52 - PL/SQL - КУРСОР, теория - продолжаем

В прошлый раз, я заморочил вам голову с аналогиями! Но надеюсь, что понятие транзакция для вас не пустой звук! :) Что ж, давайте определим, что же такое "курсор". Если говорить по большому счету понятие курсор в серверах SQL является фундаментальным и вот почему. Все базовые операторы **DML** - это не что иное, как - да! да! "курсоры". При выполнении **INSERT**, **UPDATE** и т.д. в "глобальной системной области", (применительно к **Oracle**) - всегда, открывается "курсор"! Курсоры имеют ряд атрибутов для облегчения работы с ними. Так же курсоры могут быть явными и не явными. Кстати все операторы **DML** (благо их не так много) - это не что иное, как неявные курсоры. Так же курсоры могут быть параметризованные или нет. Что ж, много всего навалилось, давайте разбирать это все последовательно. Итак, немного анатомии курсора сервера **Oracle**. При обработке **SQL** - операторов **Oracle** выделяет область памяти называемую "контекстной областью" (context area). Она содержит информацию необходимую для начала и завершения обработки **SQL** оператора. В том, числе - число строк обрабатываемых оператором, указатель на представление этого оператора после синтаксического анализа (еще это называют парсинг) и активный набор (active set) - например набор строк возвращаемых запросом.

Итак: КУРСОР - это указатель (хотя как такового, понятия "указатель" в **PL/SQL** нет!) на контекстную область памяти, с помощью которого программа на языке **PL/SQL** может управлять контекстной областью и ее состоянием во время обработки оператора.

Объявление курсора определяет какое выражение языка **SQL** - будет передано программе **SQL Statement Executor** (системе исполнителю выражения **SQL**). Курсор может представлять собой любое допустимое предложение языка **SQL**! Так же, курсор является основным базовым "кирпичиком" для построения блоков **PL/SQL**. Курсоры обеспечивают циклический механизм оперирования наборами данных в БД. Курсор может возвращать одну или несколько строк данных или вообще ни одной.

Типичная последовательность, при операциях в данном случае с явными (определенными курсорами) будет такая:

1. Объявление курсора и структуры данных, в которую, будут помещены найденные строки.
2. Открытие курсора.
3. Последовательная выборка данных.
4. Закрытие курсора.

Полный синтаксис определения явного курсора таков:

```
----- CURSOR -- имя (передаваемые параметры) --- IS -----  
----- SELECT список полей FROM таблица выбора  
----- WHERE условия выбора в курсор -----
```

Вот так определяется явный курсор. Давайте запишем несколько определений курсоров, и вы посмотрите как это делается. Если пока что-то не понятно можете не расстраиваться, я все поясню по ходу.

Итак:

```
DECLARE
```

```
-- 1.
-- Выбрать все заказы:
CURSOR get_orders IS
    SELECT * FROM ORDERS;

-- 2.
-- Выбрать несколько столбцов
-- для определенного номера заказа
CURSOR get_orders(Pord_num ORDERS.order_num%TYPE) IS
    SELECT ORDER_DATE, MFR, AMOUNT FROM ORDERS
    WHERE order_num = Pord_num;

-- 3.
-- Получить полную запись
-- для определенного номера заказа
CURSOR get_orders(Pord_num ORDERS.order_num%TYPE) IS
    SELECT * FROM ORDERS
    WHERE order_num = Pord_num
    RETURN ORDERS%ROWTYPE;

-- 4.
-- Получение имени сотрудника
-- по его номеру
CURSOR get_name(empl_nm SALESREPS.empl_num%TYPE)
    RETURN SALESREPS.name%TYPE IS
    SELECT name FROM SALESREPS WHERE empl_num = empl_nm;
```

Надеюсь, вы заметили, что курсоры не просто так определены, а выполняют отбор вполне осмысленной информации. Вспомните наши мучения с **SELECT**, я не даром там распинался. Но те кто его хорошо знают, то же пусть освежат знания. Итак в определенных (явных) курсорах ключевым звеном является оператор **SELECT** (ДА ОН ВООБЩЕ КЛЮЧЕВОЙ ЗВЕНО ВСЕХ БД!!!). Именно он определяет, что будет возвращать курсор после того как вы его откроете. Итак курсор под номером 1. Это простой не параметризованный курсор возвращающий все содержимое таблицы запроса в данном случае **ORDERS**. Это самый легкий из курсоров! :) Курсор под номером 2. Это более сложный параметризованный курсор с одним параметром, но их может быть и больше. Он возвращает как видно из условия **WHERE** одну строку, так как поле **order_num** ключевое! Обратите внимание при передаче параметра использована конструкция **%TYPE**. Курсор под номером 3. Чуть более расширенная версия предыдущего, кроме того используется конструкция **RETURN** - с применением переменной записи на базе таблицы **ORDERS**. Такая конструкция просто возвратит одну полную запись из таблицы **ORDERS** при открытии курсора. Такие конструкции курсоров мы рассмотрим после того, как разберемся с процедурами и функциями. Курсор 4. Это то же самое, что и курсор 3, но он применяет конструкцию **RETURN** для того, чтобы вернуть значение одного поля таблицы **SALESREPS**, обратите внимание на местоположение оператора **RETURN**, хотя я в своей практике такие "фортели" с курсорами не использовал! :) Но, тем не менее, в документации такое есть и вам будет полезно посмотреть как это выглядит! Вот собственно, так (пока отвлеченно) определяются курсоры. Можете пока все это переварить, так как дальше на вас обрушится еще большее количество информации!!! :)

Шаг 53 - PL/SQL - КУРСОР - идем дальше

В прошлый раз мы с вами выполнили первую часть правил работы с курсором, а именно первый пункт (Объявление курсора и структуры данных, в которую, будут помещены найденные строки.)

Собственно определение структуры данных в которую, будут помещены найденные строки мы пока опустим, к ней мы еще вернемся, а пока поговорим о втором пункте (Открытие курсора).

Курсор открывается с помощью оператора **OPEN**. Его синтаксис следующий:

----- OPEN - имя курсора(передаваемые параметры) -----

Давайте, определим вот такой курсор, следующим образом:

```
DECLARE

v_Office OFFICES.OFFICE%TYPE;

-- определяем курсор
CURSOR get_offices IS
    SELECT * FROM OFFICES
    WHERE OFFICE = v_Office;

BEGIN
    -- присваиваем значение
    v_Office := 11;

    OPEN get_offices; -- открываем курсор

    -- меняем значение
    v_Office := 12;

END;
/
```

Здесь я определил простой не параметризованный курсор, который тем не менее, получает условие через объявленную переменную, скажу сразу это не очень хороший стиль работы, но пока пусть будет так. Что же происходит когда курсор открывается? А вот, что:

1. Анализируется значение переменных привязки.
2. На основе значений переменных привязки определяется активный набор.
3. Указатель активного набора устанавливается на первую строку.

Вот таким образом происходит открытие курсора. Переменные привязки анализируются только во время открытия курсора, а у нас переменная **v_Office** перед открытием курсора равна 11. Соответственно результирующий набор отвечает данному условию. Дальнейшее изменение переменной **v_Office** с 11 на 12, уже не влияет на активный набор запроса. Такой алгоритм открытия называется согласованностью чтения (**read - consistency**). Он разработан специально для обеспечения целостности базы данных. Чтобы получить, новый набор нужно закрыть курсор, а затем снова его открыть, вот так:

```
.  
. .  
.  
OPEN get_offices; -- открываем курсор  
  
CLOSE get_offices;  
  
-- меняем значение  
v_Office := 12;  
  
OPEN get_offices; -- открываем курсор  
  
END;  
/
```

Вот теперь курсор получил данные соответствующие новому значению переменной **v_Office**! А, можно сделать и так:

```
.  
. .  
.  
OPEN get_offices; -- открываем курсор  
  
-- меняем значение значение  
v_Office := 12;  
  
OPEN get_offices; -- открываем курсор  
  
END;  
/
```

Так как повторное открытие курсора приводит к неявному вызову оператора **CLOSE**, то результат будет такой же, но использовать **CLOSE**, по моему мнению, более правильно, так как код более читаем и это является хорошим стилем программирования. Кроме того, совершенно допустимо открытие одновременно нескольких курсоров. Параметризованные курсоры открываются точно так же, но лишь с той разницей, что им передается заранее определенный параметр. Перепишем наш предыдущий курсор, как параметризованный, что так же является более правильным стилем программирования, вследствие того, что курсор с условием **WHERE** в части **SELECT** должен принимать условие отбора как параметр, а не как "размытую" переменную! :) Запишем:

```
DECLARE  
  
-- определяем переменную отбора  
v_Office OFFICES.OFFICE%TYPE;  
  
-- определяем параметризованный курсор  
CURSOR get_offices(v_Off OFFICES.office%TYPE) IS  
    SELECT * FROM OFFICES  
    WHERE OFFICE = v_Off;  
  
BEGIN
```

```
-- присваиваем значение
v_Office := 11;

OPEN get_offices(v_Office);      -- открываем курсор и передаем параметр отбора!

-- меняем значение
v_Office := 12;

END;
/
```

Вот так будет выглядеть параметризованный курсор. Ну, а в остальном, все вышесказанное справедливо и для него. Как я понимаю, вы уже поняли, что оператор **CLOSE** закрывает курсор. Давайте немного подробнее остановимся на нем. Его синтаксис таков:

----- CLOSE - имя курсора -----

Я уже упоминал, что хорошим стилем программирования является закрытие курсора, когда работа с ним окончена и вот почему. Закрытие курсора говорит о том, что программа закончила с ним работу и что все ресурсы, которые занимает данный курсор можно освободить. Если в вашей программе или процедуре один курсор, то если вы его не закроете, то система сама при завершении вызовет оператор **CLOSE**, но если у вас достаточно большой блок обработки с десятком курсоров, то их не закрытие может существенно снизить производительность вашего блока **PL/SQL**! После закрытия курсора, выбирать из него строки уже нельзя! Если попытаться это сделать, то получите сообщение об ошибке:

ORA-1001 Invalid Cursor
(неверный курсор)

или:

ORA-1002 Fetch out of Sequence
(непоследовательное считывание)

ORA-1001 - так же срабатывает, если вы попытаетесь, закрыть уже закрытый курсор. Так что с определением открытием и закрытием курсора будем считать мы разобрались! Осталось, совсем не много ... может быть! :)

Шаг 54 - PL/SQL - КУРСОР - извлекаем данные, оператор FETCH

С определениями, открытиями и закрытиями мы разобрались. Давайте, наконец, получим какие-нибудь результаты, т.е. данные. Собственно для чего и нужны курсоры. Выборка данных из курсора производится с помощью оператора **FETCH**. Пожалуй, он является ключевой фигурой в данном случае. Так как именно с его помощью происходит выборка из активного набора, сформированного при открытии курсора. После того как курсор открыт, данные в контекстной области **SGA** уже готовы, их остается только извлечь. Оператор **FETCH** именно это и производит. Синтаксис его таков:

```
----- FETCH - имя курсора - INTO - список переменных -----  
и  
----- FETCH - имя курсора - INTO - запись PL/SQL (%ROWTYPE) -----
```

Итак, переходим к практическим действиям. Запускаем наш старый добрый **SQL*Plus**. Запишем вот такой блок, используя пройденный материал:

```
DECLARE  
  
  v_Office OFFICES.OFFICE%TYPE;  
  v_City OFFICES.CITY%TYPE;  
  
  CURSOR get_offices IS  
    SELECT OFFICE, CITY  
    FROM OFFICES;  
  
BEGIN  
  
  OPEN get_offices;  
    -- Use operator FETCH to get variables!  
    FETCH get_offices INTO v_Office, v_City;  
  
  CLOSE get_offices;  
  
END;  
/
```

Получаем следующее:

```
SQL> DECLARE  
2  
3  v_Office OFFICES.OFFICE%TYPE;  
4  v_City OFFICES.CITY%TYPE;  
5  
6  CURSOR get_offices IS  
7    SELECT OFFICE, CITY  
8    FROM OFFICES;  
9  
10 BEGIN  
11  
12  OPEN get_offices;
```



```
13  -- Use operator FETCH to get variables!
14  FETCH get_offices INTO v_Office, v_City;
15
16  CLOSE get_offices;
17
18  END;
19  /
```

Процедура PL/SQL успешно завершена.

Что ж, налицо правильная выборка, в две переменные первой строки активного набора. Но, как то не наглядно, не видно результата. Давайте немного перепишем наш предыдущий блок вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    v_Office OFFICES.OFFICE%TYPE;
    v_City OFFICES.CITY%TYPE;
```

```
    CURSOR get_offices IS
        SELECT OFFICE, CITY
        FROM OFFICES;
```

```
BEGIN
```

```
    OPEN get_offices;
    -- Use operator FETCH to get variables!
    FETCH get_offices INTO v_Office, v_City;

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
    CLOSE get_offices;
```

```
END;
/
```

Получаем следующее:

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> DECLARE
```

```
2
3      v_Office OFFICES.OFFICE%TYPE;
4      v_City OFFICES.CITY%TYPE;
5
6      CURSOR get_offices IS
7          SELECT OFFICE, CITY
8          FROM OFFICES;
9
10     BEGIN
11
```

```
12      OPEN get_offices;
13      -- Use operator FETCH to get variables!
14      FETCH get_offices INTO v_Office, v_City;
15
16      DBMS_OUTPUT.enable;
17      DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
18
19      CLOSE get_offices;
20
21      END;
22      /
OutPutString is: 22 Запиндрицинск
```

Процедура PL/SQL успешно завершена.

Ура! Наконец-то мы видим результат! Давайте подробнее рассмотрим, что же произошло. Поле открытия курсора (не параметризованного) происходит выборка данных, согласно выражения **SELECT** курсора. Кстати легко проверить, что должен вернуть курсор просто выполнив его оператор **SELECT**, вот так:

```
SELECT OFFICE, CITY
FROM OFFICES
/
```

Получаем:

```
SQL> SELECT OFFICE, CITY
2     FROM OFFICES
3  /
```

```
OFFICE CITY
-----
```

```
22 Запиндрицинск
11 Красный Мотоцикл
12 Чугуевск
13 Бубурино
21 Котрогайка
```

5 строк выбрано

Это и есть ваш результирующий набор! Но здесь пять(!) строк скажете вы! Да! В таблице **OFFICES** пять записей, как вы помните из наших прошлых занятий. Но пример с **FETCH** вернул одну строку, потому что этот оператор производит выбор одной строки и смещает указатель в контекстной области на единицу. И ждет следующей команды на выборку. До тех пор, пока не будет достигнута последняя запись. Когда будет достигнута последняя запись, сработает атрибут курсора **%NOTFOUND** (его мы рассмотрим чуть позднее), он станет **TRUE**. Это значит, что все записи из результирующего набора выбраны! Но с помощью нашего примера можно выбрать и все пять записей, сделав вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
v_Office OFFICES.OFFICE%TYPE;
v_City OFFICES.CITY%TYPE;

CURSOR get_offices IS
    SELECT OFFICE, CITY
    FROM OFFICES;

BEGIN

    OPEN get_offices;

        DBMS_OUTPUT.enable;
        -- Use operator FETCH to get variables!
        FETCH get_offices INTO v_Office, v_City;    -- 1
        DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
        FETCH get_offices INTO v_Office, v_City;    -- 2
        DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
        FETCH get_offices INTO v_Office, v_City;    -- 3
        DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
        FETCH get_offices INTO v_Office, v_City;    -- 4
        DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
        FETCH get_offices INTO v_Office, v_City;    -- 5
        DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);

    CLOSE get_offices;

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3  v_Office OFFICES.OFFICE%TYPE;
4  v_City OFFICES.CITY%TYPE;
5
6  CURSOR get_offices IS
7      SELECT OFFICE, CITY
8      FROM OFFICES;
9
10 BEGIN
11
12  OPEN get_offices;
13
14  DBMS_OUTPUT.enable;
15  -- Use operator FETCH to get variables!
16  FETCH get_offices INTO v_Office, v_City; -- 1
17  DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
18  FETCH get_offices INTO v_Office, v_City; -- 2
19  DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
```

```
20  FETCH get_offices INTO v_Office, v_City; -- 3
21  DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
22  FETCH get_offices INTO v_Office, v_City; -- 4
23  DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
24  FETCH get_offices INTO v_Office, v_City; -- 5
25  DBMS_OUTPUT.put_line('OutPutString is: '||TO_CHAR(v_Office)||' '||v_City);
26
27  CLOSE get_offices;
28
29  END;
30  /
```

```
OutPutString is: 22 Запиндрицинск
OutPutString is: 11 Красный Мотоцикл
OutPutString is: 12 Чугуевск
OutPutString is: 13 Бубурино
OutPutString is: 21 Котрогайка
```

Процедура PL/SQL успешно завершена.

Вот все пять записей, каждый последующий **FETCH** вернул по одной записи! Теперь понятно? Думаю, да! И если вы уже, наверное, догадались, что здесь просто запрашивается оператор цикла! Давайте для начала (просто для примера) применим, оператор **LOOP EXIT WHEN** и при этом немного изменим наш курсор, применив атрибут **%ROWTYPE** вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    CURSOR get_offices IS
        SELECT * FROM OFFICES;
```

```
    v_gt get_offices%ROWTYPE;
```

```
BEGIN
```

```
    OPEN get_offices;
```

```
    LOOP
```

```
        EXIT WHEN get_offices%NOTFOUND;
```

```
        DBMS_OUTPUT.enable;
```

```
        -- Use operator FETCH to get variables!
```

```
        FETCH get_offices INTO v_gt;
```

```
        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
```

```
        ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
```

```
        '||TO_CHAR(v_gt.SALES));
```

```
    END LOOP;
```

```
    CLOSE get_offices;
```

```
END;
```

/

И наконец получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3  CURSOR get_offices IS
4      SELECT * FROM OFFICES;
5
6  v_gt get_offices%ROWTYPE;
7
8  BEGIN
9
10  OPEN get_offices;
11
12  LOOP
13
14  EXIT WHEN get_offices%NOTFOUND;
15
16  DBMS_OUTPUT.enable;
17  -- Use operator FETCH to get variables!
18  FETCH get_offices INTO v_gt;
19  DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
20  ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
'||TO_CHAR(v_gt.SALES));
21
22  END LOOP;
23
24  CLOSE get_offices;
25
26  END;
27 /
Get Data: 22 Запиндрищинск Запад 108 300 186,042
Get Data: 11 Красный Мотоцикл Восток 106 575 692,637
Get Data: 12 Чугуевск Восток 104 800 735,044
Get Data: 13 Бубурино Восток 105 350 367,911
Get Data: 21 Котрогайка Запад 108 725 835,915
Get Data: 21 Котрогайка Запад 108 725 835,915
```

Процедура PL/SQL успешно завершена.

Ура! Я вас поздравляю, мы наконец чуть забежав вперед, с применением оператора цикла и курсорного атрибута **%NOTFOUND**, наконец написали наш первый полноценный курсор! Который выбирает все данные из таблицы и выводит их на экран! Но, не думайте, что это все что касается курсоров! Это далеко не так! :)

Шаг 55 - PL/SQL - КУРСОР и его атрибуты

Настало время рассмотреть немаловажный аспект, а именно курсорные атрибуты. В **PL/SQL** имеется четыре основных курсорных атрибута **%FOUND**, **%NOTFOUND**, **%ISOPEN** и **%ROWCOUNT**. Атрибуты курсора объявляются подобно операторам **%TYPE** и **%ROWTYPE**, справа от имени курсора, вот так:

```
----- имя курсора%атрибут -----
```

Для того, чтобы понять что это за операторы и для чего они предназначены, давайте предположим, что наша учебная таблица **OFFICES**, содержит два столбца **OFFICE** и **CITY**, а так же имеет только две записи. Вот таким образом: (на самом деле в вашей учебной базе таблица **OFFICES** содержит 5ть записей и больше полей, но пока просто включите свою фантазию!) :)

Таблица OFFICES

OFFICE CITY

```
-----  
22 Запндрищинск  
11 Красный Мотоцикл
```

Теперь к этой таблице запишем вот такой блок **PL/SQL** и объявим в нем курсор **get_offices** следующим образом:

```
DECLARE
```

```
-- Cursor declaration --  
CURSOR get_offices IS  
    SELECT * FROM OFFICES;  
-- Record to store the fetched data --  
v_gt get_offices%ROWTYPE;
```

```
BEGIN
```

```
-- location (1) -- cursor is open --  
OPEN get_offices;  
  
-- location (2)  
FETCH get_offices INTO v_gt;           -- fetch first row --  
-- location (3)  
    FETCH get_offices INTO v_gt;       -- fetch second row --  
-- location (4)  
        FETCH get_offices INTO v_gt;   -- Third first --  
  
-- location (5)  
CLOSE get_offices;                     -- cursor is close --  
-- location (6)  
  
END;  
/
```

Теперь давайте рассмотрим по очереди все атрибуты курсоров. Начнем с атрибута **%FOUND**. Данный атрибут является логическим объектом и возвращает следующие значения согласно точкам, указанным в нашем курсоре. Надеюсь, что дополнительно пояснять не нужно, так как все достаточно хорошо видно в таблице!

%FOUND

Точка	Значение get_offices%FOUND	Пояснение
1.	Ошибка ORA-1001	Курсор еще не открыт и активного набора для него не существует!
2.	NULL	Хотя курсор открыт, не было произведено ни одного считывания строк. Значение атрибута не определено.
3.	TRUE	С помощью предшествующего оператора FETCH выбрана первая строк таблицы OFFICES.
4.	TRUE	С помощью предшествующего оператора FETCH выбрана вторая строк таблицы OFFICES.
5.	FALSE	Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны.
6.	Ошибка ORA-1001	Курсор закрыт, и вся хранившаяся информация об активном наборе удалена.

Атрибут **%NOTFOUND**, как вы уже догадались, является полной противоположностью **%FOUND** и так же хорошо понятен из приведенной ниже таблицы:

%NOTFOUND

Точка	Значение get_offices%NOTFOUND	Пояснение
1.	Ошибка ORA-1001	Курсор еще не открыт и активного набора для него не существует!
2.	NULL	Хотя курсор открыт, не было произведено ни одного считывания строк. Значение атрибута не определено.
3.	FALSE	С помощью предшествующего оператора FETCH выбрана первая строк таблицы OFFICES.
4.	FALSE	С помощью предшествующего оператора FETCH выбрана вторая строк таблицы OFFICES.
5.	TRUE	Предшествующий оператор FETCH не вернул никаких данных, так как все строки активного набора выбраны.
6.	Ошибка ORA-1001	Курсор закрыт, и вся хранившаяся информация об активном наборе удалена.

Атрибут **%ISOPEN** так же логический объект и указывает только на то, открыт ли курсор или нет. Возвращаемые значение приведены ниже:

%ISOPEN

Точка	Значение	Пояснение
-------	----------	-----------

	get_offices%ISOPEN	
1.	FALSE	Курсор get_offices еще не открыт.
2.	TRUE	Курсор get_offices был открыт.
3.	TRUE	Курсор get_offices еще открыт.
4.	TRUE	Курсор get_offices еще открыт.
5.	TRUE	Курсор get_offices еще открыт.
6.	FALSE	Курсор get_offices закрыт.

Атрибут **%ROWCOUNT** является числовым атрибутом и возвращает число строк считанных курсором на определенный момент времени. Его значение приведены ниже:

%ROWCOUNT

Точка	Значение get_offices%ROWCOUNT	Пояснение
1.	Ошибка ORA-1001	Курсор еще не открыт и активного набора для него не существует!
2.	0	Хотя курсор открыт, не было произведено ни одного считывания строк.
3.	1	Считана первая строка таблицы OFFICES
4.	2	Считана вторая строка таблицы OFFICES
5.	2	К данному моменту считаны две строки таблицы OFFICES
6.	Ошибка ORA-1001	Курсор закрыт, и вся хранившаяся информация об активном наборе удалена.

Надеюсь теперь вам стало окончательно ясно, как работает последний курсор из прошлого шага, где мы применили атрибут **%NOTFOUND** в теле цикла, дабы оповестить одного, что пора заканчивать чтение строк! :) Что ж, применим наши знания на практике, так как сухая теория это еще не все. Перепишем, наш учебный курсор вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
-- Cursor declaration --
CURSOR get_offices IS
    SELECT * FROM OFFICES;
-- Record to store the fetched data --
v_gt get_offices%ROWTYPE;
```

```
BEGIN
```

```
DBMS_OUTPUT.enable;
```

```
IF(get_offices%ISOPEN) THEN
    DBMS_OUTPUT.put_line('Cursor get_offices is open now!');
ELSE
    DBMS_OUTPUT.put_line('Cursor get_offices is close now!');
```



```
END IF;

-- location (1) -- cursor is open --
OPEN get_offices;

IF(get_offices%ISOPEN) THEN
DBMS_OUTPUT.put_line('Cursor get_offices is open now!');
ELSE
DBMS_OUTPUT.put_line('Cursor get_offices is close now!');
END IF;

-- location (2)
FETCH get_offices INTO v_gt;          -- fetch first row --
IF(get_offices%FOUND) THEN
DBMS_OUTPUT.put_line('Row is Found!');
ELSE
DBMS_OUTPUT.put_line('Row is not Found!');
END IF;
DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));

-- location (3)
FETCH get_offices INTO v_gt;          -- fetch second row --

IF(get_offices%NOTFOUND) THEN
DBMS_OUTPUT.put_line('Row is not Found!');
ELSE
DBMS_OUTPUT.put_line('Row is Found!');
END IF;
DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));

-- location (4)
FETCH get_offices INTO v_gt;          -- Third first --
IF(get_offices%FOUND) THEN
DBMS_OUTPUT.put_line('Row is Found!');
ELSE
DBMS_OUTPUT.put_line('Row is not Found!');
END IF;
DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));

-- location (5)
CLOSE get_offices;                    -- cursor is close --
-- location (6)

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
```

```
SQL> DECLARE
2
3     -- Cursor declaration --
4     CURSOR get_offices IS
5         SELECT * FROM OFFICES;
6     -- Record to store the fetched data --
7     v_gt get_offices%ROWTYPE;
8
9 BEGIN
10
11     DBMS_OUTPUT.enable;
12
13     IF(get_offices%ISOPEN) THEN
14         DBMS_OUTPUT.put_line('Cursor get_offices is open now!');
15     ELSE
16         DBMS_OUTPUT.put_line('Cursor get_offices is close now!');
17     END IF;
18
19     -- location (1) -- cursor is open --
20     OPEN get_offices;
21
22     IF(get_offices%ISOPEN) THEN
23         DBMS_OUTPUT.put_line('Cursor get_offices is open now!');
24     ELSE
25         DBMS_OUTPUT.put_line('Cursor get_offices is close now!');
26     END IF;
27
28     -- location (2)
29     FETCH get_offices INTO v_gt;          -- fetch first row --
30     IF(get_offices%FOUND) THEN
31         DBMS_OUTPUT.put_line('Row is Found!');
32     ELSE
33         DBMS_OUTPUT.put_line('Row is not Found!');
34     END IF;
35     DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
36         ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
37
38     -- location (3)
39     FETCH get_offices INTO v_gt;          -- fetch second row --
40
41     IF(get_offices%NOTFOUND) THEN
42         DBMS_OUTPUT.put_line('Row is not Found!');
43     ELSE
44         DBMS_OUTPUT.put_line('Row is Found!');
45     END IF;
46     DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
47         ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
48
49     -- location (4)
50     FETCH get_offices INTO v_gt;          -- Third first --
51     IF(get_offices%FOUND) THEN
52         DBMS_OUTPUT.put_line('Row is Found!');
53     ELSE
```

```
54      DBMS_OUTPUT.put_line('Row is not Found!');
55      END IF;
56      DBMS_OUTPUT.put_line(TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
57                          ||'The count row is '||TO_CHAR(get_offices%ROWCOUNT));
58
59      -- location (5)
60      CLOSE get_offices;          -- cursor is close --
61      -- location (6)
62
63 END;
64 /
Cursor get_offices is close now!
Cursor get_offices is open now!
Row is Found!
22 Запиндрищинск The count row is 1
Row is Found!
11 Красный Мотоцикл The count row is 2
Row is Found!
12 Чугуевск The count row is 3
```

Процедура PL/SQL успешно завершена.

Не пугайтесь этих 64-х строк кода, в дальнейшем наши программы будут все больше и больше, привыкайте! :) Здесь, я постарался применить все атрибуты для того, чтобы на практике показать, как они работают и устроены, по этому еще раз самостоятельно посмотрите весь код и обратите внимание на то, как срабатывают разные атрибуты при разных условиях, а на дом вам задание написать тот же курсор, применив цикл и выбрав все записи из него! Дерзайте!!! :)

Шаг 56 - PL/SQL - составные типы

Познакомившись, достаточно близко с курсорами, я думаю, у вас хватило терпения разобраться со всем этим! :) Далее давайте разберем один довольно интересный тип данных, который очень напоминает структуры в языке **C**. Обычные скалярные типы, такие как **VARCHAR2**, **NUMBER** и т.д. предварительно определены в модуле **STANDARD**. По этому для их использования программе требуется лишь объявить переменную, данного типа, например вот так:

```
DECLARE
```

```
m_COMPANY VARCHAR2(30);  
m_CUST_REP INTEGER;  
m_CREDIT_LIMIT NUMBER;
```

Но иногда удобнее использовать, так называемый составной тип. Одним из таких типов, в языке **PL/SQL** является запись **RECORD**. Как можно было догадаться, предыдущий пример, очень напоминает своей структурой, нашу учебную таблицу **CUSTOMERS**. Посмотрим на синтаксис объявления записи:

```
-----  TYPE тип записи IS RECORD  ( -----  
-----  поле1 тип1 [NOT NULL] [:= выражение1] -----  
-----  поле2 тип2 [NOT NULL] [:= выражение2] -----  
-----  поле-n тип-n [NOT NULL] [:= выражение-n] ); -----
```

Вот таким образом объявляется составной тип **RECORD**. Как видите для полей записи, можно указывать ограничение **NOT NULL**, но подобно описанию, переменной вне записи исходное значение и ограничение **NOT NULL** не обязательны! Давайте запишем для примера вот такой блок:

```
SET SERVEROUTPUT ON  
DECLARE
```

```
TYPE is_SmplRec IS RECORD  
(  
  m_Fld1 VARCHAR2(10),  
  m_Fld2 VARCHAR2(30) := 'Buber',  
  m_DtFld DATE,  
  m_Fld3 INTEGER := 1000,  
  m_Fld4 VARCHAR2(100) NOT NULL := 'System'  
);
```

```
MY_SMPL is_SmplRec;
```

```
BEGIN
```

```
DBMS_OUTPUT.enable;  
DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
```

```
END;  
/
```

После его прогона в **SQL*Plus** получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3 TYPE is_SmplRec IS RECORD
  4   (
  5     m_Fld1 VARCHAR2(10),
  6     m_Fld2 VARCHAR2(30) := 'Buber',
  7     m_DtFld DATE,
  8     m_Fld3 INTEGER := 1000,
  9     m_Fld4 VARCHAR2(100) NOT NULL := 'System'
 10   );
 11
 12
 13 MY_SMPL is_SmplRec;
 14
 15 BEGIN
 16
 17 DBMS_OUTPUT.enable;
 18 DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
 19
 20 END;
 21 /
Buber System
```

Процедура PL/SQL успешно завершена.

Как видно процедура успешно выполнялась и вернула значения. Здесь хорошо видно, как мы проинициализировали переменные записи внутри объявления. Так же одно из полей объявленное как **NOT NULL** сразу получило значение, вследствие того, что при этих условиях это необходимо! Давайте так же рассмотрим блок, где показано, как с использованием правил обращения к полям записи, а именно через точечную нотацию (как в языке **Pascal**), так же можно переопределить исходные значения полей записи:

```
SET SERVEROUTPUT ON
DECLARE

TYPE is_SmplRec IS RECORD
(
m_Fld1 VARCHAR2(10),
m_Fld2 VARCHAR2(30) := 'Buber',
m_DtFld DATE,
m_Fld3 INTEGER := 1000,
m_Fld4 VARCHAR2(100) NOT NULL := 'System'
);

MY_SMPL is_SmplRec;

BEGIN

MY_SMPL.m_DtFld := SYSDATE;
```

```
MY_SMPL.m_Fld4 := 'Unknown';

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);

END;
/
```

После обработки получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3 TYPE is_SmplRec IS RECORD
  4   (
  5     m_Fld1 VARCHAR2(10),
  6     m_Fld2 VARCHAR2(30) := 'Buber',
  7     m_DtFld DATE,
  8     m_Fld3 INTEGER := 1000,
  9     m_Fld4 VARCHAR2(100) NOT NULL := 'System'
 10   );
 11
 12
 13 MY_SMPL is_SmplRec;
 14
 15 BEGIN
 16
 17 MY_SMPL.m_DtFld := SYSDATE;
 18 MY_SMPL.m_Fld4 := 'Unknown';
 19
 20 DBMS_OUTPUT.enable;
 21 DBMS_OUTPUT.put_line(MY_SMPL.m_Fld2||' '||MY_SMPL.m_Fld4);
 22
 23 END;
 24 /
Buber Unknown
```

Процедура PL/SQL успешно завершена.

Как видно переменная **MY_SMPL.m_DtFld** получила значение текущей даты, с применением функции **SYSDATE**, которая возвращает текущую дату и время системы, а переменная **MY_SMPL.m_Fld4**, сменила строковый литерал. Вот таким образом, записи как составной тип данных способны к изменению своего внутреннего содержимого. Теперь давайте рассмотрим, как производится присваивание записей. В языке **PL/SQL** при присвоении значения одной записи другой используется, так называемая семантика копирования. Хотя присвоить одну запись непосредственно другой не допускается, даже если поля в обеих записях одинаковы. Для **PL/SQL** это разные типы и по этому происходит следующее:

```
SET SERVEROUTPUT ON
DECLARE
```

```
TYPE is_SmplRecOne IS RECORD
(
  m_Fld1 VARCHAR2(10),
  m_Fld2 VARCHAR2(30),
  m_DtFld DATE,
  m_Fld3 INTEGER,
  m_Fld4 VARCHAR2(100)
);
```

```
TYPE is_SmplRecTwo IS RECORD
(
  m_Fld1 VARCHAR2(10),
  m_Fld2 VARCHAR2(30),
  m_DtFld DATE,
  m_Fld3 INTEGER,
  m_Fld4 VARCHAR2(100)
);
```

```
MY_SMPLONE is_SmplRecOne;
MY_SMPLTWO is_SmplRecTwo;
```

```
BEGIN
```

```
MY_SMPLONE := MY_SMPLTWO;
```

```
END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3 TYPE is_SmplRecOne IS RECORD
4   (
5     m_Fld1 VARCHAR2(10),
6     m_Fld2 VARCHAR2(30),
7     m_DtFld DATE,
8     m_Fld3 INTEGER,
9     m_Fld4 VARCHAR2(100)
10    );
11
12 TYPE is_SmplRecTwo IS RECORD
13   (
14     m_Fld1 VARCHAR2(10),
15     m_Fld2 VARCHAR2(30),
16     m_DtFld DATE,
17     m_Fld3 INTEGER,
18     m_Fld4 VARCHAR2(100)
19    );
20
```

```
21
22 MY_SMPLONE is_SmplRecOne;
23 MY_SMPLTWO is_SmplRecTwo;
24
25 BEGIN
26
27 MY_SMPLONE := MY_SMPLTWO;
28
29 END;
30 /
MY_SMPLONE := MY_SMPLTWO;
*
```

ошибка в строке 27:

ORA-06550: Строка 27, столбец 15:

PLS-00382: выражение неправильного типа

ORA-06550: Строка 27, столбец 1:

PL/SQL: Statement ignored

Что и следовало ожидать, получаем "PLS-00382: выражение неправильного типа"! Все верно, типы то разные! И такое представление не проходит! Но вот присвоение полей этих записей между собой вполне приемливо! Вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
TYPE is_SmplRecOne IS RECORD
```

```
(
  m_Fld1 VARCHAR2(10),
  m_Fld2 VARCHAR2(30),
  m_DtFld DATE,
  m_Fld3 INTEGER,
  m_Fld4 VARCHAR2(100)
);
```

```
TYPE is_SmplRecTwo IS RECORD
```

```
(
  m_Fld1 VARCHAR2(10),
  m_Fld2 VARCHAR2(30),
  m_DtFld DATE,
  m_Fld3 INTEGER,
  m_Fld4 VARCHAR2(100)
);
```

```
MY_SMPLONE is_SmplRecOne;
```

```
MY_SMPLTWO is_SmplRecTwo;
```

```
BEGIN
```

```
MY_SMPLONE.m_Fld3 := 100;
```

```
MY_SMPLONE.m_Fld4 := 'Buber';
```

```
MY_SMPLTWO.m_Fld3 := MY_SMPLONE.m_Fld3;
MY_SMPLTWO.m_Fld4 := MY_SMPLONE.m_Fld4;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_SMPLTWO.m_Fld3));
DBMS_OUTPUT.put_line(MY_SMPLTWO.m_Fld4);

END;
/
```

Получаем после обработки:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3 TYPE is_SmplRecOne IS RECORD
  4   (
  5     m_Fld1 VARCHAR2(10),
  6     m_Fld2 VARCHAR2(30),
  7     m_DtFld DATE,
  8     m_Fld3 INTEGER,
  9     m_Fld4 VARCHAR2(100)
 10   );
 11
 12 TYPE is_SmplRecTwo IS RECORD
 13   (
 14     m_Fld1 VARCHAR2(10),
 15     m_Fld2 VARCHAR2(30),
 16     m_DtFld DATE,
 17     m_Fld3 INTEGER,
 18     m_Fld4 VARCHAR2(100)
 19   );
 20
 21
 22 MY_SMPLONE is_SmplRecOne;
 23 MY_SMPLTWO is_SmplRecTwo;
 24
 25 BEGIN
 26
 27 MY_SMPLONE.m_Fld3 := 100;
 28 MY_SMPLONE.m_Fld4 := 'Buber';
 29
 30 MY_SMPLTWO.m_Fld3 := MY_SMPLONE.m_Fld3;
 31 MY_SMPLTWO.m_Fld4 := MY_SMPLONE.m_Fld4;
 32
 33 DBMS_OUTPUT.enable;
 34 DBMS_OUTPUT.put_line(TO_CHAR(MY_SMPLTWO.m_Fld3));
 35 DBMS_OUTPUT.put_line(MY_SMPLTWO.m_Fld4);
 36
 37 END;
 38 /
100
```

Buber

Процедура PL/SQL успешно завершена.

Вот теперь все правильно! Типы полей совпадают и присвоение проходит нормально! Присвоить значения полям, базы можно и с помощью оператора **SELECT**, выбрав одну запись из таблицы. Например, давайте запишем такой блок:

```
DECLARE

TYPE is_Customers IS RECORD
(
  m_COMPANY CUSTOMERS.COMPANY%TYPE,
  m_CUST_REP CUSTOMERS.CUST_REP%TYPE,
  m_CREDIT_LIMIT CUSTOMERS.CREDIT_LIMIT%TYPE
);

MY_CUST is_Customers;

BEGIN

SELECT COMPANY, CUST_REP, CREDIT_LIMIT INTO MY_CUST
FROM CUSTOMERS WHERE CUST_NUM = 2108;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(MY_CUST.m_COMPANY||' '||
TO_CHAR(MY_CUST.m_CUST_REP)||' '||TO_CHAR(MY_CUST.m_CREDIT_LIMIT));

END;
/
```

Здесь объявлена запись на основе типов таблицы **CUSTOMERS**, и выбрана одна запись из этой таблицы с помощью оператора **INTO**, все это отправлено в переменные записи:

```
SQL> DECLARE
2
3 TYPE is_Customers IS RECORD
4   (
5     m_COMPANY CUSTOMERS.COMPANY%TYPE,
6     m_CUST_REP CUSTOMERS.CUST_REP%TYPE,
7     m_CREDIT_LIMIT CUSTOMERS.CREDIT_LIMIT%TYPE
8   );
9
10 MY_CUST is_Customers;
11
12 BEGIN
13
14 SELECT COMPANY, CUST_REP, CREDIT_LIMIT INTO MY_CUST
15 FROM CUSTOMERS WHERE CUST_NUM = 2108;
16
17 DBMS_OUTPUT.enable;
18 DBMS_OUTPUT.put_line(MY_CUST.m_COMPANY||' '||
19 TO_CHAR(MY_CUST.m_CUST_REP)||' '||TO_CHAR(MY_CUST.m_CREDIT_LIMIT));
20
21 END;
```

[22 /](#)
[Унесенные ветром 109 55,323](#)

[Процедура PL/SQL успешно завершена.](#)

Вот таким образом, работает составной тип **RECORD**. Да, кстати я тут нашел, кое-какие ошибочки и недочеты в прошлых уроках. Я за них извиняюсь, так как материал, очень объемный и сложный! Вы сами можете мне на них указать, я буду рад! При этом сразу оговорюсь ДЕЛАЮ КАК МОГУ И СТАРАЮСЬ ПОМОЧЬ ВСЕМ В ИЗУЧЕНИИ ORACLE! А, отпускать в мой адрес ехидные заковырки, может каждый, а вот реально что-то сделать в этой жизни способны не многие! По этому ошибки в частности в [шаге 17](#) исправлены и ничего страшного я в этом не вижу!!! :) Что ж, удачи всем и жду конструктивных советов и замечаний!

Шаг 57 - PL/SQL - составные типы данных

Пришло, я думаю, время разобрать составной тип, который, по моему мнению, является наиболее мощным и простым средством скоростной обработки больших массивов данных. Такой составной тип имеет название **TABLE**! Рассмотрим его синтаксис: (он подобен синтаксису **RECORD**)

----- TYPE - тип таблицы - IS TABLE OF тип - INDEX BY BINARY_INTEGER -----

Вот так просто объявить составной тип **TABLE**! В своей сути это одномерный массив скалярного типа. Он не может содержать тип **RECORD** или другой тип **TABLE**. Но может быть объявлен от любого другого стандартного типа. И это еще не все! **TABLE** может быть объявлен от атрибута **%ROWTYPE**! Вот где скрывается, по истине огромная мощь, этого типа данных! Но пока, все по порядку. Тип **TABLE**, можно представить как одну из разновидностей коллекции. Хотя в более глубоком понимании это не совсем так. При первом рассмотрении он похож на массив языка **C**. Массив **TABLE** индексируется типом **BINARY_INTEGER** и может содержать, что-то вроде - 2,147,483,647 - 0 - + 2,147,483,647 как в положительную, так и в отрицательную часть. Начинать индексацию можно с 0 или 1 или любого другого числа. Если массив будет иметь разрывы, то это не окажет какой-либо дополнительной нагрузки на память. Так же следует помнить, что для каждого элемента, например, типа **VARCHAR2** будет резервироваться столько памяти, сколько вы укажете, по этому определяйте размерность элементов по необходимости. Скажем не стоит объявлять **VARCHAR2(255)**, если хотите хранить строки менее 100 символов! :) Итак, давайте объявим массив и посмотрим, как он работает. Запишем такой блок:

```
SET SERVEROUTPUT ON
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

TYPE m_SmplTblData IS TABLE OF DATE
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;
MY_TBL_DT m_SmplTblData;

BEGIN

MY_TBL(1) := 'Buber';
MY_TBL_DT(2) := SYSDATE - 1;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(1)));
DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL_DT(2)));

END;
/
```

Получаем после обработки в **SQL*Plus**:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
5
6     TYPE m_SmplTblData IS TABLE OF DATE
7     INDEX BY BINARY_INTEGER;
8
9
10    MY_TBL m_SmplTable;
11    MY_TBL_DT m_SmplTblData;
12
13 BEGIN
14
15 MY_TBL(1) := 'Buber';
16 MY_TBL_DT(2) := SYSDATE - 1;
17
18 DBMS_OUTPUT.enable;
19 DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(1)));
20 DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL_DT(2)));
21
22 END;
23 /
Buber
01.11.03
```

Процедура PL/SQL успешно завершена.

Рассмотрим, что же здесь происходит. Мы объявили две одномерные коллекции **m_SmplTable**, **m_SmplTblData**, одна из них содержит элементы размерностью **VARCHAR2(128)**, другая **DATE**. Затем объявили две переменные данного типа - **MY_TBL**, **MY_TBL_DT** и присвоили первому элементу строкового массива значение "Buber", а второму элементу массива дат, значение текущей даты минус 1. Результат вывели на консоль! Вот собственно и все. При этом хорошо видно, что тип **TABLE** очень схож с таблицей БД и содержит обычно два ключа **KEY** и **VALUE**, ключ и значение. Ключ имеет тип **BINARY_INTEGER**:

-- переменная(KEY)VALUE --

В нашем случае имеет место:

Ключ (KEY)	Значение (VALUE)
MY_TBL	
1	Buber
MY_TBL_DT	
2	01.11.03

При этом можете обратить внимание на следующее:

1. Число строк таблицы ни чем не ограничено. Единственное ограничение это значения, которые могут быть представлены типом **BINARY_INTEGER**.

2. Порядок элементов таблицы **PL/SQL** не обязательно должен быть строго определен. Эти элементы хранятся в памяти не подряд как массивы и по этому могут вводиться с произвольными ключами.
3. Ключи, используемые в таблице **PL/SQL**, не обязательно должны быть последовательными. В качестве индекса таблицы может быть использовано любое значение или выражение имеющее тип **BINARY_INTEGER**.

Вот такие правила действуют при определении и работе с таблицами **PL/SQL** (не путать с таблицами БД!!!). Так же при обращении к несуществующему элементу получите сообщение об ошибке! Давайте запишем вот такой блок:

```
SET SERVEROUTPUT ON
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;

BEGIN

FOR i IN 1..10 LOOP
MY_TBL(i) := TO_CHAR(i+5);
END LOOP;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(5)));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
5
6     MY_TBL m_SmplTable;
7
8 BEGIN
9
10 FOR i IN 1..10 LOOP
11 MY_TBL(i) := TO_CHAR(i+5);
12 END LOOP;
13
14 DBMS_OUTPUT.enable;
15 DBMS_OUTPUT.put_line(TO_CHAR(MY_TBL(5)));
16
17 END;
18 /
```

10

Процедура PL/SQL успешно завершена.

Здесь мы объявили коллекцию из строковых переменных, и с помощью известного вам цикла **FOR** записали с 1-го по 10-й элемент значениями исходный плюс пять. Затем мы вывели на экран пятый элемент, как и следовало ожидать его значение равно 10 (5+5). Вот один из способов наполнения массива значениями. Но этот способ бесполезен и показан в качестве примера. А, вот следующий пример более осмыслен. Перепишем блок из [предыдущего шага](#) вот так:

```
DECLARE

TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE
INDEX BY BINARY_INTEGER;

MY_CUST is_Customers;

BEGIN

SELECT * INTO MY_CUST(100)
FROM CUSTOMERS WHERE CUST_NUM = 2108;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(MY_CUST(100).CUST_NUM)||' '||MY_CUST(100).COMPANY||' '||
TO_CHAR(MY_CUST(100).CUST_REP)||' '||TO_CHAR(MY_CUST(100).CREDIT_LIMIT));

END;
/
```

Получаем:

```
SQL> DECLARE
2
3 TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE
4     INDEX BY BINARY_INTEGER;
5
6 MY_CUST is_Customers;
7
8 BEGIN
9
10 SELECT * INTO MY_CUST(100)
11 FROM CUSTOMERS WHERE CUST_NUM = 2108;
12
13 DBMS_OUTPUT.enable;
14 DBMS_OUTPUT.put_line(TO_CHAR(MY_CUST(100).CUST_NUM)||'
'||MY_CUST(100).COMPANY||' '||
15 TO_CHAR(MY_CUST(100).CUST_REP)||' '||TO_CHAR(MY_CUST(100).CREDIT_LIMIT));
16
17 END;
18 /
2108 Унесенные ветром 109 55,323
```

Процедура PL/SQL успешно завершена.

Здесь мы объявили коллекцию, с типом запись из таблицы **CUSTOMERS**! И получили тот же результат как в прошлый раз, но более элегантно! Итак, объявляем тип:

```
TYPE is_Customers IS TABLE OF CUSTOMERS%ROWTYPE  
INDEX BY BINARY_INTEGER;
```

Это значит, что каждая строка коллекции содержит полную запись из таблицы БД **CUSTOMERS**. Замечательно. Далее 100 элементу массива присваиваем значение записи таблицы с индексом 2108. Вот так это выглядит изнутри:

```
MY_CUST(98)  
MY_CUST(99)  
.  
MY_CUST(100) ->  
MY_CUST(100).CUST_NUM MY_CUST(100).COMPANY MY_CUST(100).CUST_REP  
MY_CUST(100).CREDIT_LIMIT  
2108          Унесенные ветром          109          55,323  
.  
.  
.  
MY_CUST(n)
```

Обращаться к отдельному элементу (полю) записи можно через точечную нотацию. Что мы и проделали с вами в примере! Так же скажу, что используя привязку таблиц **PL/SQL** к массивам интерфейса **OCI** можно достичь очень высокой скорости передачи данных в клиентских приложениях БД. В следующий раз рассмотрим атрибуты таблиц **PL/SQL**.

Шаг 58 - PL/SQL - PL/SQL таблицы и их атрибуты

С таблицами **PL/SQL** мы познакомились, теперь разберем их атрибуты. Атрибуты таблиц имеют следующий синтаксис:

----- таблица.атрибут -----

Где таблица - это ссылка на таблицу **PL/SQL**, а атрибут - собственно сам атрибут. Основные атрибуты таблиц **PL/SQL** следующие:

Атрибут	Возвращаемый тип	Описание
COUNT	NUMBER	Возвращает число строк таблицы.
DELETE	-	Удаляет строки таблицы.
EXISTS	BOOLEAN	Возвращает TRUE если указанный элемент находится в таблице иначе FALSE.
FIRST	BINARY_INTEGER	Возвращает индекс первой строки таблицы.
LAST	BINARY_INTEGER	Возвращает индекс последней строки таблицы.
NEXT	BINARY_INTEGER	Возвращает индекс строки таблицы, которая следует за указанной строкой.
PRIOR	BINARY_INTEGER	Возвращает индекс строки таблицы, которая предшествует указанной строке.

Вот основные атрибуты таблиц **PL/SQL**. Теперь давайте разберем их все на примерах. Начнем по порядку. Атрибут **COUNT**. Запишем вот такой блок:

```
SET SERVEROUTPUT ON
-- COUNT
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;

BEGIN

FOR i IN 1..10 LOOP
MY_TBL(i) := TO_CHAR(i+5);
END LOOP;

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2
  3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
  4     INDEX BY BINARY_INTEGER;
  5
  6     MY_TBL m_SmplTable;
  7
  8 BEGIN
  9
 10 FOR i IN 1..10 LOOP
 11 MY_TBL(i) := TO_CHAR(i+5);
 12 END LOOP;
 13
 14 DBMS_OUTPUT.enable;
 15 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
 16
 17 END;
 18 /
Count table is: 10
```

Процедура PL/SQL успешно завершена.

С помощью цикла **FOR** мы ввели в таблицу 10 значений и применив атрибут - **COUNT** получили количество записей в таблице. Вот собственно и все. Следующий атрибут **DELETE**. Тут чуть сложнее. Во-первых, он имеет аргументы:

- **DELETE** - удаляет все строки таблицы.
- **DELETE(n)** - удаляет **n**-ю строку коллекции.
- **DELETE(n,m)** - удаляет строки коллекции с **n** по **m**.

Рассмотрим для примера такой блок:

```
SET SERVEROUTPUT ON
-- DELETE
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;

BEGIN

MY_TBL(1) := 'One';
MY_TBL(3) := 'Three';
MY_TBL(-2) := 'Minus Two';
MY_TBL(0) := 'Zero';
MY_TBL(100) := 'Hundred';

DBMS_OUTPUT.enable;
```

```
DBMS_OUTPUT.put_line(MY_TBL(1));
DBMS_OUTPUT.put_line(MY_TBL(3));
DBMS_OUTPUT.put_line(MY_TBL(-2));
DBMS_OUTPUT.put_line(MY_TBL(0));
DBMS_OUTPUT.put_line(MY_TBL(100));
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

MY_TBL.DELETE(100);
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
MY_TBL.DELETE(1,3);
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
MY_TBL.DELETE;
DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> -- DELETE
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
5
6     MY_TBL m_SmplTable;
7
8 BEGIN
9
10 MY_TBL(1) := 'One';
11 MY_TBL(3) := 'Three';
12 MY_TBL(-2) := 'Minus Two';
13 MY_TBL(0) := 'Zero';
14 MY_TBL(100) := 'Hundred';
15
16 DBMS_OUTPUT.enable;
17
18 DBMS_OUTPUT.put_line(MY_TBL(1));
19 DBMS_OUTPUT.put_line(MY_TBL(3));
20 DBMS_OUTPUT.put_line(MY_TBL(-2));
21 DBMS_OUTPUT.put_line(MY_TBL(0));
22 DBMS_OUTPUT.put_line(MY_TBL(100));
23 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
24
25 MY_TBL.DELETE(100);
26 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
27 MY_TBL.DELETE(1,3);
28 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
29 MY_TBL.DELETE;
30 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
31
32 END;
```

```
33 /
One
Three
Minus Two
Zero
Hundred
Count table is: 5
Count table is: 4
Count table is: 2
Count table is: 0
```

Процедура PL/SQL успешно завершена.

В данном случае сначала мы удалили 100-ю запись коллекции, затем с 1 по 3 (при этом 2-й записи не существовало, но это не столь важно) и затем очистили всю коллекцию. Вот так работает атрибут **DELETE**. И не путайте его с оператором **DML DELETE**! К стати очистить всю коллекцию целиком, можно и так:

```
SET SERVEROUTPUT ON
-- DELETE
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;
MY_TBL_Empty m_SmplTable;

BEGIN

MY_TBL(1) := 'One';
MY_TBL(3) := 'Three';
MY_TBL(-2) := 'Minus Two';
MY_TBL(0) := 'Zero';
MY_TBL(100) := 'Hundred';

DBMS_OUTPUT.enable;

MY_TBL := MY_TBL_Empty;

DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> -- DELETE
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
```

```
4      INDEX BY BINARY_INTEGER;
5
6      MY_TBL m_SmplTable;
7      MY_TBL_Empty m_SmplTable;
8
9 BEGIN
10
11 MY_TBL(1) := 'One';
12 MY_TBL(3) := 'Three';
13 MY_TBL(-2) := 'Minus Two';
14 MY_TBL(0) := 'Zero';
15 MY_TBL(100) := 'Hundred';
16
17 DBMS_OUTPUT.enable;
18
19      MY_TBL := MY_TBL_Empty;
20
21      DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
22
23 END;
24 /
Count table is: 0
```

Процедура PL/SQL успешно завершена.

Как видно, присвоение пустой коллекции, а при объявлении коллекции это именно так, вся таблица очистилась, за один, заход! Но как это делать на практике судить Вам! :) И последний на этот раз атрибут **EXISTS**. Рассмотрим такой блок:

```
SET SERVEROUTPUT ON
-- EXISTS
DECLARE

TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;

BEGIN

MY_TBL(1) := 'Miller';
MY_TBL(3) := 'Kolobok';

DBMS_OUTPUT.enable;

DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

IF (MY_TBL.EXISTS(1)) THEN
DBMS_OUTPUT.put_line(MY_TBL(1));
ELSE
DBMS_OUTPUT.put_line('MY_TBL(1) is not exist!');
END IF;
```

```
IF (MY_TBL.EXISTS(3)) THEN
DBMS_OUTPUT.put_line(MY_TBL(3));
ELSE
DBMS_OUTPUT.put_line('MY_TBL(3) is not exist!');
END IF;

IF (MY_TBL.EXISTS(2)) THEN
DBMS_OUTPUT.put_line(MY_TBL(2));
ELSE
DBMS_OUTPUT.put_line('MY_TBL(2) is not exist!');
END IF;

END;
/
```

Далее получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> -- EXISTS
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
5
6     MY_TBL m_SmplTable;
7
8 BEGIN
9
10 MY_TBL(1) := 'Miller';
11 MY_TBL(3) := 'Kolobok';
12
13 DBMS_OUTPUT.enable;
14
15 DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
16
17 IF (MY_TBL.EXISTS(1)) THEN
18 DBMS_OUTPUT.put_line(MY_TBL(1));
19 ELSE
20 DBMS_OUTPUT.put_line('MY_TBL(1) is not exist!');
21 END IF;
22
23 IF (MY_TBL.EXISTS(3)) THEN
24 DBMS_OUTPUT.put_line(MY_TBL(3));
25 ELSE
26 DBMS_OUTPUT.put_line('MY_TBL(3) is not exist!');
27 END IF;
28
29 IF (MY_TBL.EXISTS(2)) THEN
30 DBMS_OUTPUT.put_line(MY_TBL(2));
31 ELSE
32 DBMS_OUTPUT.put_line('MY_TBL(2) is not exist!');
33 END IF;
34
```

```
35 END;  
36 /  
Count table is: 2  
Miller  
Kolobok  
MY_TBL(2) is not exist!
```

Процедура PL/SQL успешно завершена.

Здесь хорошо видно, что в объявленной коллекции существуют только две строки с номером 1 и 3! А строка с номером 2 не существует. Что и продемонстрировал наш пример. Следующие атрибуты разберем далее... :) А, пока запоминайте и анализируйте.

Шаг 59 - PL/SQL - PL/SQL атрибуты PL/SQL таблиц, примеры

Что ж, продолжаем рассматривать на примерах работу с **PL/SQL** - таблицами. Следующий пример показывает как работают атрибуты **FIRST** и **LAST**. Запишем такой блок:

```
SET SERVEROUTPUT ON
-- FIRST LAST
DECLARE

    TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
    INDEX BY BINARY_INTEGER;

    MY_TBL m_SmplTable;

    FRST BINARY_INTEGER;
    LST BINARY_INTEGER;

BEGIN

    MY_TBL(1) := 'One';
    MY_TBL(3) := 'Three';
    MY_TBL(-2) := 'Minus Two';
    MY_TBL(0) := 'Zero';
    MY_TBL(100) := 'Hundred';

    DBMS_OUTPUT.enable;

    DBMS_OUTPUT.put_line(MY_TBL(1));
    DBMS_OUTPUT.put_line(MY_TBL(3));
    DBMS_OUTPUT.put_line(MY_TBL(-2));
    DBMS_OUTPUT.put_line(MY_TBL(0));
    DBMS_OUTPUT.put_line(MY_TBL(100));
    DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));

    FRST := MY_TBL.FIRST;
    LST := MY_TBL.LAST;

    DBMS_OUTPUT.put_line('Count table first: '||TO_CHAR(FRST));

    DBMS_OUTPUT.put_line('Count table last: '||TO_CHAR(LST));

END;
/
```

После запуска в **SQL*Plus** - получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> -- FIRST LAST
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
```



```
5
6     MY_TBL m_SmplTable;
7
8     FRST BINARY_INTEGER;
9     LST BINARY_INTEGER;
10
11 BEGIN
12
13     MY_TBL(1) := 'One';
14     MY_TBL(3) := 'Three';
15     MY_TBL(-2) := 'Minus Two';
16     MY_TBL(0) := 'Zero';
17     MY_TBL(100) := 'Hundred';
18
19     DBMS_OUTPUT.enable;
20
21     DBMS_OUTPUT.put_line(MY_TBL(1));
22     DBMS_OUTPUT.put_line(MY_TBL(3));
23     DBMS_OUTPUT.put_line(MY_TBL(-2));
24     DBMS_OUTPUT.put_line(MY_TBL(0));
25     DBMS_OUTPUT.put_line(MY_TBL(100));
26     DBMS_OUTPUT.put_line('Count table is: '||TO_CHAR(MY_TBL.COUNT));
27
28     FRST := MY_TBL.FIRST;
29     LST := MY_TBL.LAST;
30
31     DBMS_OUTPUT.put_line('Count table first: '||TO_CHAR(FRST));
32
33     DBMS_OUTPUT.put_line('Count table last: '||TO_CHAR(LST));
34
35 END;
36 /
One
Three
Minus Two
Zero
Hundred
Count table is: 5
Count table first: -2
Count table last: 100
```

Процедура PL/SQL успешно завершена.

Хорошо видно, что количество записей в коллекции пять, и первая запись имеет индекс -2(!), а последняя 100. Вот такая необычная ситуация, хотя здесь собственно все верно. Далее, рассмотрим на примере работу атрибутов **NEXT** и **PRIOR**. С их помощью можно перемещаться по коллекции. Рассмотрим такой блок:

```
SET SERVEROUTPUT ON
-- NEXT PRIOR
DECLARE

    TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
```

```
INDEX BY BINARY_INTEGER;

MY_TBL m_SmplTable;
m_Cnt BINARY_INTEGER;
a BINARY_INTEGER;

BEGIN

FOR i IN 1..10 LOOP
MY_TBL(i) := TO_CHAR(i+5);
END LOOP;

DBMS_OUTPUT.enable;

-- FOR i IN MY_TBL.FIRST..MY_TBL.LAST LOOP
-- DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(i)));
-- END LOOP;

m_Cnt := MY_TBL.FIRST;

LOOP

    DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(m_Cnt)));
    EXIT WHEN m_Cnt = MY_TBL.LAST;
    m_Cnt := MY_TBL.NEXT(m_Cnt);

END LOOP;

m_Cnt := MY_TBL.LAST;

LOOP

    DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(m_Cnt)));
    EXIT WHEN m_Cnt = MY_TBL.FIRST;
    m_Cnt := MY_TBL.PRIOR(m_Cnt);

END LOOP;

END;
/
```

После запуска в **SQL*Plus** - получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> -- NEXT PRIOR
SQL> DECLARE
2
3     TYPE m_SmplTable IS TABLE OF VARCHAR2(128)
4     INDEX BY BINARY_INTEGER;
5
6     MY_TBL m_SmplTable;
7     m_Cnt BINARY_INTEGER;
8     a BINARY_INTEGER;
```

```
9
10 BEGIN
11
12     FOR i IN 1..10 LOOP
13         MY_TBL(i) := TO_CHAR(i+5);
14     END LOOP;
15
16     DBMS_OUTPUT.enable;
17
18     -- FOR i IN MY_TBL.FIRST..MY_TBL.LAST LOOP
19     -- DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(i)));
20     -- END LOOP;
21
22     m_Cnt := MY_TBL.FIRST;
23
24     LOOP
25
26         DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(m_Cnt)));
27         EXIT WHEN m_Cnt = MY_TBL.LAST;
28         m_Cnt := MY_TBL.NEXT(m_Cnt);
29
30     END LOOP;
31
32     m_Cnt := MY_TBL.LAST;
33
34     LOOP
35
36         DBMS_OUTPUT.put_line('Table is: '||TO_CHAR(MY_TBL(m_Cnt)));
37         EXIT WHEN m_Cnt = MY_TBL.FIRST;
38         m_Cnt := MY_TBL.PRIOR(m_Cnt);
39
40     END LOOP;
41
42 END;
43 /
```

```
Table is: 6
Table is: 7
Table is: 8
Table is: 9
Table is: 10
Table is: 11
Table is: 12
Table is: 13
Table is: 14
Table is: 15
Table is: 15
Table is: 14
Table is: 13
Table is: 12
Table is: 11
Table is: 10
Table is: 9
Table is: 8
```

Table is: 7

Table is: 6

Процедура PL/SQL успешно завершена.

Давайте детальнее рассмотрим, что здесь происходит. Первый цикл наполняет коллекцию с 1-го по 10-й элементы, затем первый цикл **LOOP** выводит элементы коллекции по возрастанию, а второй цикл **LOOP** выводит элементы коллекции по убыванию. Хорошо видно, как при этом срабатывают атрибуты **NEXT**, **PRIOR**, а так же **FIRST**, **LAST**. Кстати закомментированный второй цикл **FOR** делает более компактный код, можете его раскомментировать и посмотреть, а на дом, сделать третий цикл **FOR**, который будет выводить как последний **LOOP**! Вот мы и рассмотрели на примерах работу всех атрибутов коллекций **Oracle PL/SQL**. Запоминайте! :)

Шаг 60 - PL/SQL - КУРСОРЫ - Неявные Курсоры

Мы с вами, уже рассмотрели основные концепции определения и работы с курсорами. Но один аспект я упустил. А именно - неявные курсоры. Как вы уже, наверное, догадались, сам оператор **SELECT** представляет собой в прямом определении "КУРСОР", или еще это называют **SQL**-курсор! Откуда следует вывод, что любой оператор **DML** объемлит собой курсор. Так как, каждый оператор **DML** выполняется в пределах контекстной области и по этому имеет курсор указывающий на контекстную область. В отличии от явных, **SQL**-курсор не открывается и не закрывается. **PL/SQL** - сам неявно открывает **SQL**-курсор, обрабатывает **SQL**-оператор и закрывает **SQL**-курсор. Для **SQL**-курсора операторы **FETCH**, **OPEN**, **CLOSE** не нужны, но с ними можно использовать курсорные атрибуты, вот таким образом:

----- SQL%АТРИБУТ КУРСОРА -----

Давайте рассмотрим конкретный пример. В [ware 48](#) мы закончили работу с табличками **PEOPLE** и **OLD_PEOPLE**, если вы их удалили, то ничего страшного, просто вернитесь к [wary 46](#) и снова их создайте. Если они у вас остались, еще лучше! Итак, рассмотрим такой пример на основе таблички **PEOPLE**. Пусть нам нужно изменить поле **NM** в таблице **PEOPLE**, где поле **ID** содержит значение 555. Введем такой запрос, перед тем как:

```
SELECT * FROM PEOPLE  
/
```

Получаем (у меня так, у вас может отличаться, если таблица удалялась или вы делали что-то еще):

```
SQL> SELECT * FROM PEOPLE  
2 /
```

ID	NM	FM	OT
7	Irvin	Show	Brefovich
2	Bob	Jason	Martovich
3	IVAN	Black	NULL

Но строки, где бы поле **ID** содержало 555, нет! Это нам и нужно, для того, чтобы продемонстрировать работу атрибута. Итак, запишем блок:

```
BEGIN  
  
UPDATE PEOPLE  
  SET NM = 'Pupkin'  
 WHERE ID = 555;  
  
  IF(SQL%NOTFOUND) THEN  
    INSERT INTO PEOPLE(ID, NM, FM, OT)  
      VALUES(555, 'Pupkin', 'Axlamon', 'Feodosovich');  
  END IF;  
  
COMMIT;  
  
END;
```

/

Получаем:

```
SQL> BEGIN
2
3 UPDATE PEOPLE
4     SET NM = 'Pupkin'
5 WHERE ID = 555;
6
7     IF(SQL%NOTFOUND) THEN
8     INSERT INTO PEOPLE(ID, NM, FM, OT)
9         VALUES(555, 'Pupkin', 'Axlamon', 'Feodosovich');
10    END IF;
11
12 COMMIT;
13
14 END;
15 /
```

Процедура PL/SQL успешно завершена.

Посмотрим, что вышло:

```
SELECT * FROM PEOPLE
/
```

Получаем:

```
SQL> SELECT * FROM PEOPLE
2 /
```

ID	NM	FM	OT
7	Irvin	Show	Brefovich
2	Bob	Jason	Martovich
3	IVAN	Black	NULL
555	Pupkin	Axlamon	Feodosovich

Как видите, сработал атрибут **SQL**-курсора, **%NOTFOUND**. И, так как записи с таким значением поля **ID** не было, то с помощью оператора **INSERT** мы его добавили в таблицу **PEOPLE**! Все получилось верно! То же можно было сделать применив атрибут **SQL**-курсора, **%ROWCOUNT**. Вот так:

```
BEGIN

UPDATE PEOPLE
  SET NM = 'Mirkin'
WHERE ID = 888;

IF(SQL%ROWCOUNT = 0) THEN
  INSERT INTO PEOPLE(ID, NM, FM, OT)
```

```
VALUES(888, 'Mirkin', 'Lupoglaz', 'Kotletovich');
END IF;

COMMIT;

END;
/
```

Получаем:

```
SQL> BEGIN
2
3 UPDATE PEOPLE
4     SET NM = 'Mirkin'
5 WHERE ID = 888;
6
7     IF(SQL%ROWCOUNT = 0) THEN
8     INSERT INTO PEOPLE(ID, NM, FM, OT)
9         VALUES(888, 'Mirkin', 'Lupoglaz', 'Kotletovich');
10    END IF;
11
12 COMMIT;
13
14 END;
15 /
```

Процедура PL/SQL успешно завершена.

Смотрим содержимое таблицы **PEOPLE**:

```
SELECT * FROM PEOPLE
/
```

Получаем:

```
SQL> SELECT * FROM PEOPLE
2 /
```

ID	NM	FM	OT
7	Irvin	Show	Brefovich
2	Bob	Jason	Martovich
3	IVAN	Black	NULL
555	Pupkin	Axlamon	Feodosovich
888	Mirkin	Lupoglaz	Kotletovich

И здесь все сработало верно, с той разницей, что мы применили атрибут **SQL**-курсора **%ROWCOUNT**. Теперь, давайте рассмотрим более сложную ситуацию, с применением атрибута **SQL**-курсора **%NOTFOUND**. А начнем, вот с чего. Я уже показывал вам оператор **SELECT** - формы **SELECT ... INTO**. Но, не заострял внимание. Давайте немного отвлечемся и я все постараюсь объяснить. Итак, **SELECT ... INTO**, это как бы некий эквивалент явного курсора с применением

оператора выборки **FETCH**. Но сам по себе он является неявным курсором с выборкой в переменную. Понятно? Если нет, идем дальше. Если явный курсор вида:

```
CURSOR get_people IS
  SELECT * FROM PEOPLE;

-- Record to store the fetched data --
v_gt get_people%ROWTYPE;
.
.
.
FETCH get_people INTO v_gt;
```

Выбегает данные с помощью оператора **FETCH**, имеющего конструкцию, **INTO** в переменную **v_gt**. В свою очередь, являющейся одномерной коллекцией на основе курсора **get_people**. То запись вида:

```
-- Record to store the fetched data --
v_gt PEOPLE%ROWTYPE;

SELECT * INTO v_gt FROM PEOPLE;
```

Абсолютно ей эквивалентна. **SELECT ... INTO** и есть неявный курсор с выборкой данных в одномерную коллекцию **v_gt**, так как они обе имеют один и тот же тип определения **ROWTYPE** и содержат четыре переменных вида:

```
v_gt.ID, v_gt.NM, v_gt.FM, v_gt.OT
```

С той лишь разницей, что первая объявлена на основе курсора (поля которого определены собственно выражением **SELECT**), а вторая на основе полей таблицы! Теперь, я надеюсь, что все неясности по поводу курсоров как явных, так и не явных у вас отпали сами собой. Безусловно, явные курсоры наиболее предпочтительны, так как более наглядны и применимы. Но если вам необходимо выполнить что-то очень простое, то можно использовать **SELECT ... INTO**. В остальных случаях только явные курсоры! :) Итак, собственно переходим к делу, если применить атрибут **%NOTFOUND** совместно с конструкцией **SELECT ... INTO**, то он может не сработать, так как если **SELECT ... INTO** не получит запись, то возникнет ошибка **"ORA-1403 no data found"**. Для этого мы запишем вот такой блок и разберем его:

```
SET SERVEROUTPUT ON

DECLARE

  m_PLP PEOPLE%ROWTYPE;

BEGIN

  DBMS_OUTPUT.enable;

  SELECT * INTO m_PLP FROM PEOPLE
  WHERE ID = 999;
```



```
IF (SQL%NOTFOUND) THEN
INSERT INTO PEOPLE(ID, NM, FM, OT)
      VALUES(999, 'Volopasov', 'Lobotryas', 'Oslovich');
END IF;

COMMIT;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.put_line('No Data Found! Execute exception handler');

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3     m_PLP PEOPLE%ROWTYPE;
4
5 BEGIN
6
7     DBMS_OUTPUT.enable;
8
9     SELECT * INTO m_PLP FROM PEOPLE
10    WHERE ID = 999;
11
12    IF (SQL%NOTFOUND) THEN
13    INSERT INTO PEOPLE(ID, NM, FM, OT)
14      VALUES(999, 'Volopasov', 'Lobotryas', 'Oslovich');
15    END IF;
16
17    COMMIT;
18
19 EXCEPTION
20    WHEN NO_DATA_FOUND THEN
21    DBMS_OUTPUT.put_line('No Data Found! Execute exception handler');
22
23 END;
24 /
No Data Found! Execute exception handler
```

Процедура PL/SQL успешно завершена.

Как видите, сработал блок **EXCEPTION** (как с ним управляться - это чуть позже). Что и должно было произойти в нашем случае. Так как блок "**SELECT * INTO...**" не получил ни одной записи, то сработал блок **EXCEPTION** с условием **NO_DATA_FOUND**. Вот так производится обработка атрибутов при использовании неявных курсоров.

Шаг 61 - PL/SQL - КУРСОРЫ - циклы и выборка данных

Итак, немного разобравшись с неявными курсорами приступим вплотную к изучению явных курсоров. Если кто из вас был внимателен, и в [шаге 54](#) заметил, что одна из строк в последнем курсоре с циклом выборки выдалась дважды! А произошло это по той причине, что оператор **FETCH** с одними данными сработал дважды. Кстати на досуге можете подумать над тем как устранить этот недочет. А, пока давайте поучимся писать курсоры с циклами. Для работы будем использовать таблицу **OFFICES**. Для начала запишем вот такой блок:

```
SET SERVEROUTPUT ON

DECLARE

    CURSOR get_offices IS
        SELECT * FROM OFFICES;

    v_gt get_offices%ROWTYPE;

BEGIN

    DBMS_OUTPUT.enable;

    OPEN get_offices;

    LOOP

        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
            ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
            '||TO_CHAR(v_gt.SALES));

        FETCH get_offices INTO v_gt;
        EXIT WHEN get_offices%NOTFOUND;

    END LOOP;

    CLOSE get_offices;

END;
/
```

После прохода в **SQL*Plus** получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3     CURSOR get_offices IS
4         SELECT * FROM OFFICES;
5
6     v_gt get_offices%ROWTYPE;
7
8 BEGIN
```

```
9
10      DBMS_OUTPUT.enable;
11
12      OPEN get_offices;
13
14      LOOP
15
16          DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
17                                ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
18                                '||TO_CHAR(v_gt.SALES));
19
20          FETCH get_offices INTO v_gt;
21          EXIT WHEN get_offices%NOTFOUND;
22
23      END LOOP;
24
25      CLOSE get_offices;
26
27 END;
```

Get Data:
Get Data: 22 Запиндрищинск Запад 108 300 186,042
Get Data: 11 Красный Мотоцикл Восток 106 575 692,637
Get Data: 12 Чугуевск Восток 104 800 735,044
Get Data: 13 Бубурино Восток 105 350 367,911
Get Data: 21 Котрогайка Запад 108 725 835,915

Процедура PL/SQL успешно завершена.

Опять хорошо видно, что первая строка пустая. Но все по порядку. Использовали стандартный цикл **LOOP**, как правило это самый распространенный подход при написании курсора. А первая строка, пустая вследствие того, что **put_line** отработал с пустыми переменными. Этот недочет можно исправить, вынеся первый оператор **FETCH** за тело цикла вот так:

```
DECLARE

    CURSOR get_offices IS
        SELECT * FROM OFFICES;

    v_gt get_offices%ROWTYPE;

BEGIN

    DBMS_OUTPUT.enable;

    OPEN get_offices;

    FETCH get_offices INTO v_gt;

    LOOP

        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||' '||TO_CHAR(v_gt.SALES));

    END LOOP;
```

```
        ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
'||TO_CHAR(v_gt.SALES));

        FETCH get_offices INTO v_gt;
        EXIT WHEN get_offices%NOTFOUND;

    END LOOP;

    CLOSE get_offices;

END;
/
```

Получаем:

```
SQL> DECLARE
2
3     CURSOR get_offices IS
4         SELECT * FROM OFFICES;
5
6     v_gt get_offices%ROWTYPE;
7
8 BEGIN
9
10    DBMS_OUTPUT.enable;
11
12    OPEN get_offices;
13
14    FETCH get_offices INTO v_gt;
15
16    LOOP
17
18        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
19        ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
'||TO_CHAR(v_gt.SALES));
20
21        FETCH get_offices INTO v_gt;
22        EXIT WHEN get_offices%NOTFOUND;
23
24    END LOOP;
25
26    CLOSE get_offices;
27
28 END;
29 /
Get Data: 22 Запиндришинск Запад 108 300 186,042
Get Data: 11 Красный Мотоцикл Восток 106 575 692,637
Get Data: 12 Чугуевск Восток 104 800 735,044
Get Data: 13 Бубурино Восток 105 350 367,911
Get Data: 21 Котрогайка Запад 108 725 835,915
```

Процедура PL/SQL успешно завершена.

Вот теперь количество строк точно соответствует количеству записей в таблице. Если не устраивает, можете придумать более оригинальный способ! :) Вот так строится классический курсор с циклом **LOOP**. Давайте сделаем выборку циклом **WHILE** и посмотрим, что получится. Запишем следующий блок:

```
SET SERVEROUTPUT ON

DECLARE

    CURSOR get_offices IS
        SELECT * FROM OFFICES;

    v_gt get_offices%ROWTYPE;

BEGIN

    DBMS_OUTPUT.enable;

    OPEN get_offices;

    FETCH get_offices INTO v_gt;

    WHILE (get_offices%FOUND) LOOP

        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
            ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
            '||TO_CHAR(v_gt.SALES));

        FETCH get_offices INTO v_gt;

    END LOOP;

    CLOSE get_offices;

END;
/
```

После прохода в **SQL*Plus** получаем:

```
SQL> DECLARE
2
3     CURSOR get_offices IS
4         SELECT * FROM OFFICES;
5
6     v_gt get_offices%ROWTYPE;
7
8 BEGIN
9
10    DBMS_OUTPUT.enable;
11
12    OPEN get_offices;
13
14    FETCH get_offices INTO v_gt;
```

```
15
16     WHILE (get_offices%FOUND) LOOP
17
18         DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
19         ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
'||TO_CHAR(v_gt.SALES));
20
21         FETCH get_offices INTO v_gt;
22
23     END LOOP;
24
25     CLOSE get_offices;
26
27 END;
28 /
Get Data: 22 Запиндрищинск Запад 108 300 186,042
Get Data: 11 Красный Мотоцикл Восток 106 575 692,637
Get Data: 12 Чугуевск Восток 104 800 735,044
Get Data: 13 Бубурино Восток 105 350 367,911
Get Data: 21 Котрогайка Запад 108 725 835,915
```

Процедура PL/SQL успешно завершена.

А, вот здесь выноска первого **FETCH** за тело цикла, просто необходима по определению, иначе сам цикл **WHILE** будет трудно заставить работать. В дальнейшем я бы рекомендовал не использовать данный цикл для работы с курсором вследствие его неудобства. Далее, вы поймете, что цикл **LOOP** наиболее применим и еще цикл **FOR**, но это уже отдельная тема. Пока можете поработать с этими типами и попробовать устранить мои ошибки! Удачи!

Шаг 62 - PL/SQL - КУРСОРЫ - цикл выборки FOR

В [прошлый раз](#) мы пробовали работать с циклами **LOOP** и **WHILE**. Но есть один очень интересный момент, касающийся выборки данных из курсоров, а именно цикл **FOR**! Давайте посмотрим на классический цикл выборки **LOOP**:

```
DECLARE
  CURSOR get_offices IS
    SELECT * FROM OFFICES;
  -- Объявляется курсор.
  v_gt get_offices%ROWTYPE;
  -- Объявляется переменная запись, где будут храниться данные.
BEGIN
  OPEN get_offices;
  -- Открывается курсор.

  -- Запускается цикл чтения.
  LOOP

    -- Выбираются данные.
    FETCH get_offices INTO v_gt;
    EXIT WHEN get_offices%NOTFOUND;

  END LOOP;

  -- Закрывается курсор.
  CLOSE get_offices;

END;
/
```

Вроде ничего не забыли. А вот, теперь посмотрите на вот такой блок:

```
DECLARE

  CURSOR get_offices IS
    SELECT * FROM OFFICES;

BEGIN

  DBMS_OUTPUT.enable;

  FOR v_gt IN get_offices LOOP

    DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
      ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
      '||TO_CHAR(v_gt.SALES));

  END LOOP;

END;
/
```

После отработки в **SQL*Plus** получаем:

```
SQL> DECLARE
2
3     CURSOR get_offices IS
4         SELECT * FROM OFFICES;
5
6 BEGIN
7
8     DBMS_OUTPUT.enable;
9
10    FOR v_gt IN get_offices LOOP
11
12        DBMS_OUTPUT.put_line('Get Data: '||TO_CHAR(v_gt.OFFICE)||' '||v_gt.CITY||' '
13            ||v_gt.REGION||' '||TO_CHAR(v_gt.MGR)||' '||TO_CHAR(v_gt.TARGET)||'
14            '||TO_CHAR(v_gt.SALES));
15    END LOOP;
16
17 END;
18 /
Get Data: 22 Запиндрищинск Запад 108 300 186,042
Get Data: 11 Красный Мотоцикл Восток 106 575 692,637
Get Data: 12 Чугуевск Восток 104 800 735,044
Get Data: 13 Бубурино Восток 105 350 367,911
Get Data: 21 Котрогайка Запад 108 725 835,915
```

Процедура PL/SQL успешно завершена.

Действия все те же самые, но кода гораздо меньше! И вот почему. При работе с циклом **FOR** все становится на порядок проще, во-первых, открывать и закрывать курсор не нужно, он делает это сам не явно. Во-вторых, объявлять переменную запись не нужно, то есть нужно, но она определяется гораздо проще, в-третьих, меньше строчек кода. Теперь по порядку, в классическом **FOR**:

```
FOR v_gt ( переменная запись, то же что и определение v_gt get_offices%ROWTYPE )
IN get_offices ( это и есть диапазон выборки сам курсор от и до как оператор .. ) LOOP начало
цикла!
```

Вот и все, вот так **FOR** упрощает выборку данных из курсора! К слову хорошего стиля программирования рекомендую, если нет необходимости использовать LOOP при выборке данных из курсора, то всегда используйте **FOR**! :)

Шаг 63 - PL/SQL - КУРСОРЫ - курсоры с параметрами

Раньше мы с вами рассматривали оператор **SELECT**, который получал все данные из таблиц. Но как известно, чаще вы будете использовать условие **WHERE** в курсорном операторе **SELECT**. И в дальнейшем вам будет необходимо определять условия выборки. Как это сделать наиболее правильно, самый простой способ вот так:

```
-- Пишем курсор с условием
CURSOR get_sls IS
  SELECT * FROM SALESREPS
  WHERE AGE = 37;
```

Замечательно, а если нужно другое значение отличное от 37? Что тогда? Напрашивается вывод о том, что в этом случае необходим курсор с параметрами или "параметризованный курсор"! Он определяется вот так:

```
CURSOR get_sls(INAGE NUMBER) IS
  SELECT * FROM SALESREPS
  WHERE AGE = INAGE;
```

Но еще более правильно, будет сделать вот так:

```
CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
  SELECT * FROM SALESREPS
  WHERE AGE = INAGE;
```

Применив оператор **TYPE** мы сделали код более мобильным, так как если тип поля изменится, то не нужно будет лопатить весь код в поисках ошибки! Это так же считается наиболее правильным стилем программирования! :) И теперь при открытии такого курсора, его оператор **OPEN** будет принимать передаваемый параметр вот так:

```
OPEN get_sls(37);
```

Подытожим наши заключения следующим блоком:

```
SET SERVEROUTPUT ON

DECLARE

  CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
    SELECT * FROM SALESREPS
    WHERE AGE = INAGE;

  in_sls get_sls%ROWTYPE;

BEGIN

  DBMS_OUTPUT.enable;

  OPEN get_sls(37);
```

```
    FETCH get_sls INTO in_sls;

    LOOP

        DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'
'||in_sls.HIRE_DATE);

        FETCH get_sls INTO in_sls;
        EXIT WHEN get_sls%NOTFOUND;

    END LOOP;

    CLOSE get_sls;

END;
/
```

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3     CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
4         SELECT * FROM SALESREPS
5         WHERE AGE = INAGE;
6
7     in_sls get_sls%ROWTYPE;
8
9 BEGIN
10
11     DBMS_OUTPUT.enable;
12
13     OPEN get_sls(37);
14
15     FETCH get_sls INTO in_sls;
16
17     LOOP
18
19         DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'
'||in_sls.HIRE_DATE);
20
21         FETCH get_sls INTO in_sls;
22         EXIT WHEN get_sls%NOTFOUND;
23
24     END LOOP;
25
26     CLOSE get_sls;
27
28 END;
29 /
Record is: Вася Пупкин Рапорт продажа 12.02.88
Record is: Максим Галкин Продано все 12.10.89
```

Процедура PL/SQL успешно завершена.

Вот так отработал наш с вами первый параметризованный курсор, надеюсь, все понятно, если нет - спрашивайте! Для закрепления, давайте запишем тот же курсор, с циклом **FOR**:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS  
        SELECT * FROM SALESREPS  
        WHERE AGE = INAGE;
```

```
BEGIN
```

```
    DBMS_OUTPUT.enable;
```

```
    FOR in_sls IN get_sls(37) LOOP
```

```
        DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'  
'||in_sls.HIRE_DATE);
```

```
    END LOOP;
```

```
END;
```

```
/
```

Получаем:

```
SQL> DECLARE
```

```
2
```

```
3     CURSOR get_sls(INAGE SALESREPS.AGE%TYPE) IS
```

```
4         SELECT * FROM SALESREPS
```

```
5         WHERE AGE = INAGE;
```

```
6
```

```
7 BEGIN
```

```
8
```

```
9     DBMS_OUTPUT.enable;
```

```
10
```

```
11     FOR in_sls IN get_sls(37) LOOP
```

```
12
```

```
13         DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'  
'||in_sls.HIRE_DATE);
```

```
14
```

```
15     END LOOP;
```

```
16
```

```
17 END;
```

```
18 /
```

```
Record is: Вася Пупкин Рапорт продажа 12.02.88
```

```
Record is: Максим Галкин Продано все 12.10.89
```

Процедура PL/SQL успешно завершена.

Параметров у курсора может быть несколько, например вот так:

```
SET SERVEROUTPUT ON

DECLARE

    CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE, INSL SALESREPS.SALES%TYPE) IS
        SELECT * FROM SALESREPS
        WHERE MANAGER = INMG AND SALES > INSL;

BEGIN

    DBMS_OUTPUT.enable;

    FOR in_sls IN get_sls(104, 300) LOOP

        DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'
'||TO_CHAR(in_sls.HIRE_DATE,'DD-MM-YYYY')||
        ' '||TO_CHAR(in_sls.SALES));

    END LOOP;

END;
/
```

Получаем после запуска:

```
SQL> DECLARE
2
3     CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE, INSL SALESREPS.SALES%TYPE) IS
4         SELECT * FROM SALESREPS
5         WHERE MANAGER = INMG AND SALES > INSL;
6
7 BEGIN
8
9     DBMS_OUTPUT.enable;
10
11     FOR in_sls IN get_sls(104, 300) LOOP
12
13         DBMS_OUTPUT.put_line('Record is: '||in_sls.NAME||' '||in_sls.TITLE||'
'||TO_CHAR(in_sls.HIRE_DATE,'DD-MM-YYYY')||
14         ' '||TO_CHAR(in_sls.SALES));
15
16     END LOOP;
17
18 END;
19 /
Record is: Вася Пупкин Рапорт продажа 12-02-1988 367,911
Record is: Игорь Николаев Рапорт продажа 20-10-1986 305,673
```

Процедура PL/SQL успешно завершена.

Что собственно и требовалось доказать. Вот так работают курсоры с параметрами и без таковых. Надеюсь, теперь вы научились писать курсоры, и выбирать из них данные! :)

Шаг 64 - PL/SQL - КУРСОРЫ - курсоры с обновлением

Иногда, при выборке из курсора бывает ситуация, что какой-либо столбец или строки результирующего набора необходимо обновить. То есть, изменить их содержимое. Для того, чтобы это осуществить, непосредственно при объявлении курсора необходимо использовать конструкцию - **FOR UPDATE** (для обновления ..). А, так же конструкцию, **WHERE CURRENT OF** (где текущая строка ..) в операторах **UPDATE**, **DELETE**. Собственно конструкция **FOR UPDATE**, является частью оператора **SELECT** и объявляется последней:

```
----- SELECT .... FROM .... FOR UPDATE [OF ссылка на столбец][NOWAIT] -----
```

Где, собственно, "ссылка на столбец" это столбец таблицы, для которой выполнен запрос. Можно так же использовать список столбцов. Например, вот так:

```
DECLARE
```

```
    CURSOR get_sls IS
        SELECT * FROM SALESREPS
        FOR UPDATE OF SALESREPS.QUOTA, SALESREPS.SALES;
```

```
    .
    .
    .
    .
```

```
-- Для столбцов QUOTA и SALES, таблицы SALESREPS.
```

```
DECLARE
```

```
    CURSOR get_sls(INMG SALESREPS.MANAGER%TYPE) IS
        SELECT * FROM SALESREPS
        WHERE MANAGER = INMG
        FOR UPDATE;
```

```
    .
    .
    .
    .
```

```
-- Для всех столбцов таблицы SALESREPS.
```

Теперь немного теории. Так как обычный запрос с помощью оператора **SELECT**, при выполнении получает строки таблицы и при этом сама таблица выборки не блокируется, то есть любой другой пользователь может выполнить запрос к той же таблице, получив при этом данные. В **Oracle** при выполнении запроса, т.е. при извлечении активного набора **SELECT**, производится моментальный снимок таблицы (**snapshot**), при этом все изменения сделанные до этого момента кем-либо еще отражаются в данном наборе. А, после того как **snapshot** получен все изменения, произведенные в данной таблице выборке, даже если они зафиксированы оператором **COMMIT**, отражаться не будут!!! Для того, чтобы их отразить нужно закрыть и снова открыть курсор, загрузив данные заново! Это и есть алгоритм согласованного чтения данных, о котором я уже упоминал ранее. А вот когда мы объявляем **FOR UPDATE** - строки активного набора данных блокируются до момента выполнения **COMMIT**. Таким образом мы запрещаем изменение данных другим сеансам. Если какой-либо сеанс уже блокировал строки, то следующий **SELECT FOR UPDATE**, будет ждать снятия

блокировки. В этом случае можно применить **NOWAIT** (без ожидания). Если обратиться к заблокированным строкам получим сообщение об ошибке **ORA-54**. Вот таким образом это работает. А вот конструкция **WHERE CURRENT OF** используется уже непосредственно при изменении данных:

----- WHERE CURRENT OF курсор -----

Где "курсор" - это курсор, открытый на обновление. Давайте рассмотрим практический пример такого курсора:

```
DECLARE

CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
        SELECT * FROM SALESREPS
               WHERE MANAGER IN (
                           SELECT O.MGR FROM OFFICES O
                           WHERE TARGET > INTG)
        FOR UPDATE OF SALESREPS.QUOTA;

BEGIN

FOR get_cur_upd IN cur_upd(700) LOOP

UPDATE SALESREPS
    SET SALESREPS.QUOTA = SALESREPS.QUOTA + 50
    WHERE CURRENT OF cur_upd;

END LOOP;

COMMIT;

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> DECLARE
2
3  CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
4    SELECT * FROM SALESREPS
5      WHERE MANAGER IN (
6        SELECT O.MGR FROM OFFICES O
7        WHERE TARGET > INTG)
8    FOR UPDATE OF SALESREPS.QUOTA;
9
10 BEGIN
11
12 FOR get_cur_upd IN cur_upd(700) LOOP
13
14 UPDATE SALESREPS
15   SET SALESREPS.QUOTA = SALESREPS.QUOTA + 50
16   WHERE CURRENT OF cur_upd;
```

```
17
18 END LOOP;
19
20 COMMIT;
21
22 END;
23 /
```

Процедура PL/SQL успешно завершена.

Да, а что произошло? Просто строки столбца **QUOTA**, таблицы **SALESREPS**, соответствующие условию **TARGET > 700** увеличились, на 50! Что, можно проверить, выполнив простой запрос:

```
SELECT * FROM SALESREPS
/
```

Его посмотрите сами. Обратите внимание на то, что курсор выполнен с передачей параметра и использует цикл **LOOP**. Конструкция **FOR UPDATE OF SALESREPS.QUOTA** определяет обновляемый столбец, конструкция **WHERE CURRENT OF cur_upd** в операторе **UPDATE SALESREPS** определяет какие строки обновить. Кстати для закрепления, используя пакет **DBMS_OUTPUT** можете добавить код для того, чтобы было видно, что происходит. Оператор **COMMIT**, расположенный вне тела цикла, снимает блокировку с таблицы и фиксирует изменения. Давайте, с помощью другого блока, применив более компактный код, вернем все назад:

```
DECLARE

CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
    SELECT * FROM SALESREPS
        WHERE MANAGER IN (
            SELECT O.MGR FROM OFFICES O
            WHERE TARGET > INTG)
    FOR UPDATE NOWAIT;

BEGIN

FOR get_cur_upd IN cur_upd(700) LOOP

UPDATE SALESREPS
    SET SALESREPS.QUOTA = SALESREPS.QUOTA - 50
    WHERE CURRENT OF cur_upd;

END LOOP;

COMMIT;

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> DECLARE
2
```



```
3      CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
4          SELECT * FROM SALESREPS
5              WHERE MANAGER IN (
6                  SELECT O.MGR FROM OFFICES O
7                  WHERE TARGET > INTG)
8          FOR UPDATE NOWAIT;
9
10 BEGIN
11
12     FOR get_cur_upd IN cur_upd(700) LOOP
13
14         UPDATE SALESREPS
15             SET SALESREPS.QUOTA = SALESREPS.QUOTA - 50
16             WHERE CURRENT OF cur_upd;
17
18     END LOOP;
19
20 COMMIT;
21
22 END;
23 /
```

Процедура PL/SQL успешно завершена.

Здесь хорошо видно, что мы применили конструкцию **FOR UPDATE NOWAIT**, хотя в результате работы мы получили то, что нужно. Так же, применив курсорный цикл **FOR**, мы сделали более компактный код. Теперь, давайте побеседуем на тему оператора **COMMIT**. Посмотрим вот такой блок:

```
DECLARE

    CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
        SELECT * FROM SALESREPS
            WHERE MANAGER IN (
                SELECT O.MGR FROM OFFICES O
                WHERE TARGET > INTG)
        FOR UPDATE NOWAIT;

    get_cur_upd cur_upd%ROWTYPE;

BEGIN

    OPEN cur_upd(700);

    FETCH cur_upd INTO get_cur_upd;

    COMMIT WORK;

    FETCH cur_upd INTO get_cur_upd;

END;
/
```

Получаем:

```
SQL> DECLARE
  2
  3     CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
  4         SELECT * FROM SALESREPS
  5             WHERE MANAGER IN (
  6                 SELECT O.MGR FROM OFFICES O
  7                 WHERE TARGET > INTG)
  8             FOR UPDATE NOWAIT;
  9
 10     get_cur_upd cur_upd%ROWTYPE;
 11
 12 BEGIN
 13
 14     OPEN cur_upd(700);
 15
 16     FETCH cur_upd INTO get_cur_upd;
 17
 18     COMMIT WORK;
 19
 20     FETCH cur_upd INTO get_cur_upd;
 21
 22 END;
 23 /
DECLARE
*
```

ошибка в строке 1:

ORA-01002: выборка из последовательности

ORA-06512: на line 20

Оператор **COMMIT WORK** снял блокировку с таблицы и последующая выборка привела к ошибке. Следовательно, как в предыдущих примерах, располагать **COMMIT** нужно после цикла. А что, если нужно обновить строки из курсора не применяя конструкции **FOR UPDATE** ? Можно, если у таблицы есть уникальный ключ, вот так:

```
DECLARE

    CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS
        SELECT * FROM SALESREPS
            WHERE MANAGER IN (
                SELECT O.MGR FROM OFFICES O
                WHERE TARGET > INTG);

BEGIN

    FOR get_cur_upd IN cur_upd(700) LOOP

        UPDATE SALESREPS
            SET SALESREPS.QUOTA = SALESREPS.QUOTA - 10
            WHERE EMPL_NUM = get_cur_upd.EMPL_NUM;

        -- Is not FOR UPDATE --
```

```
COMMIT WORK;  
  
END LOOP;  
  
COMMIT;  
  
END;  
/
```

Получаем:

```
SQL> DECLARE  
2  
3     CURSOR cur_upd(INTG OFFICES.TARGET%TYPE) IS  
4         SELECT * FROM SALESREPS  
5             WHERE MANAGER IN (  
6                 SELECT O.MGR FROM OFFICES O  
7                 WHERE TARGET > INTG);  
8  
9  
10 BEGIN  
11  
12     FOR get_cur_upd IN cur_upd(700) LOOP  
13  
14         UPDATE SALESREPS  
15             SET SALESREPS.QUOTA = SALESREPS.QUOTA - 10  
16             WHERE EMPL_NUM = get_cur_upd.EMPL_NUM;  
17  
18         -- Is not FOR UPDATE --  
19         COMMIT WORK;  
20  
21     END LOOP;  
22  
23     COMMIT;  
24  
25 END;  
26 /
```

Процедура PL/SQL успешно завершена.

Здесь нет конструкции **FOR UPDATE**, но строки обновляются с применением первичного ключа таблицы. Как правило, так обычно и поступают, но **FOR UPDATE** так же в отдельных случаях бывает весьма полезной. И оператор **COMMIT** теперь расположен внутри цикла выборки! Здесь все просто, но только с первого взгляда. Если, что не понятно, можете спрашивать! :)

Шаг 65 - PL/SQL - встроенные функции, часть 1

По-моему настало время немного отвлечься. Пока вы там перевариваете курсоры и все что к ним прилагается я вам пока расскажу о встроенных функциях **PL/SQL**. Если их хорошо знать, то можно писать очень эффективный код на **PL/SQL**. Сделаем так, я пропишу блок, а вы будете по мере изучения смотреть нужные строки! Поехали:

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
DBMS_OUTPUT.enable;
```

```
-- Function CHR() --
```

```
DBMS_OUTPUT.put_line('1*->'||CHR(37)||' '||CHR(38)||' '||CHR(101)||' '||CHR(105));
```

```
-- Function CONCAT() --
```

```
DBMS_OUTPUT.put_line('2*->'||CONCAT('Vasiya', 'Pupkin'));
```

```
-- Function INITCAP() --
```

```
DBMS_OUTPUT.put_line('3*->'||INITCAP('iF yoU or mY scoRE and 7 YEARS ago ...'));
```

```
-- Function LOWER() --
```

```
DBMS_OUTPUT.put_line('4*->'||LOWER('iF yoU or mY scoRE and 7 YEARS ago ...'));
```

```
-- Function LPAD() 1 --
```

```
DBMS_OUTPUT.put_line('5*->'||LPAD('Short String', 20, 'XY'));
```

```
-- Function LPAD() 2 --
```

```
DBMS_OUTPUT.put_line('6*->'||LPAD('Short String', 13, 'PD'));
```

```
-- Function LTRIM() 1 --
```

```
DBMS_OUTPUT.put_line('7*->'||LTRIM('    The White House has a many tree!'));
```

```
-- Function LTRIM() 2 --
```

```
DBMS_OUTPUT.put_line('8*->'||LTRIM('xxxxxThe White House has a many tree!', 'x'));
```

```
-- Function LTRIM() 3 --
```

```
DBMS_OUTPUT.put_line('9*->'||LTRIM('xyxyxyxyxyThe White House has a many tree!', 'xy'));
```

```
-- Function LTRIM() 4 --
```

```
DBMS_OUTPUT.put_line('10*->'||LTRIM('xyxyxxxxxyThe White House has a many tree!', 'xy'));
```

```
-- Function REPLACE() 1 --
```

```
DBMS_OUTPUT.put_line('11*->'||REPLACE('This and That', 'Th', 'B'));
```

```
-- Function REPLACE() 2 --
```

```
DBMS_OUTPUT.put_line('12*->'||REPLACE('This and That', 'Th'));
```

```
-- Function REPLACE() 3 --
```

```
DBMS_OUTPUT.put_line('13*->'||REPLACE('This and That', NULL));
```

```
END;
```

/

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3  DBMS_OUTPUT.enable;
  4  -- Function CHR() --
  5  DBMS_OUTPUT.put_line('1*->'||CHR(37)||' '||CHR(38)||' '||CHR(101)||' '||CHR(105));
  6
  7  -- Function CONCAT() --
  8  DBMS_OUTPUT.put_line('2*->'||CONCAT('Vasiya', 'Pupkin'));
  9
 10  -- Function INITCAP() --
 11  DBMS_OUTPUT.put_line('3*->'||INITCAP('iF yoU or mY scoRE and 7 YEARS ago ...'));
 12
 13  -- Function LOWER() --
 14  DBMS_OUTPUT.put_line('4*->'||LOWER('iF yoU or mY scoRE and 7 YEARS ago ...'));
 15
 16  -- Function LPAD() 1 --
 17  DBMS_OUTPUT.put_line('5*->'||LPAD('Short String', 20, 'XY'));
 18
 19  -- Function LPAD() 2 --
 20  DBMS_OUTPUT.put_line('6*->'||LPAD('Short String', 13, 'PD'));
 21
 22  -- Function LTRIM() 1 --
 23  DBMS_OUTPUT.put_line('7*->'||LTRIM('    The White House has a many tree!'));
 24
 25  -- Function LTRIM() 2 --
 26  DBMS_OUTPUT.put_line('8*->'||LTRIM('xxxxxThe White House has a many tree!', 'x'));
 27
 28  -- Function LTRIM() 3 --
 29  DBMS_OUTPUT.put_line('9*->'||LTRIM('xyxyxyxyxyThe White House has a many tree!', 'xy'));
 30
 31  -- Function LTRIM() 4 --
 32  DBMS_OUTPUT.put_line('10*->'||LTRIM('xyxyxxxxxyThe White House has a many tree!',
'xy'));
 33
 34  -- Function REPLACE() 1 --
 35  DBMS_OUTPUT.put_line('11*->'||REPLACE('This and That', 'Th', 'B'));
 36
 37  -- Function REPLACE() 2 --
 38  DBMS_OUTPUT.put_line('12*->'||REPLACE('This and That', 'Th'));
 39
 40  -- Function REPLACE() 3 --
 41  DBMS_OUTPUT.put_line('13*->'||REPLACE('This and That', NULL));
 42
 43  END;
 44  /
1*->% & e i
2*->VasiyaPupkin
```

```
3*->If You Or My Score And 7 Years Ago ...
4*->if you or my score and 7 years ago ...
5*->XYXYXYShort String
6*->PShort String
7*->The White House has a many tree!
8*->The White House has a many tree!
9*->The White House has a many tree!
10*->The White House has a many tree!
11*->Bis and Bat
12*->is and at
13*->This and That
```

Процедура PL/SQL успешно завершена.

Все это относится к СИМВОЛЬНЫМ ФУНКЦИЯМ, ВОЗВРАЩАЮЩИМ СИМВОЛЬНЫЕ ЗНАЧЕНИЯ.

CHR(x)

Возвращает символ, имеющий код, равный x в наборе символов БД. Пример строка 1*.

CONCAT(строка 1, строка2)

Возвращает "строка 1", конкатенированную, (сцепленную) со "строка 2". То же что и операция "||"! Пример строка 2*.

INITCAP(строка)

Возвращает "строка", в которой каждое слово начинается с прописной буквы и продолжается строчными. Слова разделяются пробелами или не буквенно-цифровыми символами. Символы не являющиеся буквами не изменяются. Пример строка 3*.

LOWER(строка)

Возвращает "строка", со строчными символами. Символы не являющиеся буквами не изменяются. Пример строка 4*.

LPAD(строка 1, x, строка 2)

Вот интересная функция! :) Возвращает "строка 1", дополненную слева до размера x символами "строка 2". Если размер "строка 2", меньше x, то при необходимости она дублируется. Если размер "строка 2" больше x, то берутся только первые x ее символов. Если "строка 2" не указана, то ее заменяют символы пробела. Пример строка 5*, 6*.

LTRIM(строка 1, строка 2)

Возвращает "строка 1", в которой удалены крайние левые символы, идентичные символам "строка 2". Значением по умолчанию для "строка 2", является знак пробела. "строка 1" просматривается с левого края, и при встрече первого символа не совпадающего с "строка 2", возвращается результат. Пример строка 7*, 8*, 9*, 10*.

REPLACE(строка_символов, строка_поиска, [строка_замены])

Возвращает "строка_символов", в которой каждое вхождение "строка_поиска", заменяется на "строка_замены". Если "строка_замены", не указана, то все вхождения "строка_поиска", удаляются из "строка_символов". Пример строка 11*, 12*, 13*.

Далее, продолжим.

Шаг 66 - PL/SQL - встроенные функции, часть 2

Продолжим разбор функций **PL/SQL**. Символьные функции далее:

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
-- Function RPAD() 1 --
```

```
DBMS_OUTPUT.put_line('14*->'||RPAD('This is Cool', 15, '!'));
```

```
-- Function RPAD() 2 --
```

```
DBMS_OUTPUT.put_line('15*->'||RPAD('This is Cool', 19, 'MA'));
```

```
-- Function RTRIM() 1 --
```

```
DBMS_OUTPUT.put_line('16*->'||RTRIM('The White House has a many tree!   '));
```

```
-- Function RTRIM() 2 --
```

```
DBMS_OUTPUT.put_line('17*->'||RTRIM('The White House has a many tree!xxxxxx', 'x'));
```

```
-- Function RTRIM() 3 --
```

```
DBMS_OUTPUT.put_line('18*->'||RTRIM('The White House has a many tree!xyxyxyxyxy', 'xy'));
```

```
-- Function RTRIM() 4 --
```

```
DBMS_OUTPUT.put_line('19*->'||RTRIM('The White House has a many tree!xyxyxxxxxy', 'xy'));
```

```
-- Function SUBSTR() 1 --
```

```
DBMS_OUTPUT.put_line('20*->'||SUBSTR('Braun Fox',1,5));
```

```
-- Function SUBSTR() 2 --
```

```
DBMS_OUTPUT.put_line('21*->'||SUBSTR('Braun Fox',-3,3));
```

```
-- Function SUBSTR() 3 --
```

```
DBMS_OUTPUT.put_line('22*->'||SUBSTR('Braun Fox',3));
```

```
-- Function TRANSLATE() 1 --
```

```
DBMS_OUTPUT.put_line('23*->'||TRANSLATE('KrotPups', 'Pups', 'Boss'));
```

```
-- Function TRANSLATE() 2 --
```

```
DBMS_OUTPUT.put_line('24*->'||TRANSLATE('KrotPups', 'KrotPups', 'Kolobok'));
```

```
-- Function UPPER() --
```

```
DBMS_OUTPUT.put_line('25*->'||UPPER('THE quick bROwn Fox jumped oVer THE LaZy dOG ... '));
```

```
END;
```

```
/
```

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> BEGIN
```

```
2
```

```
3 -- Function RPAD() 1 --
```

```
4  DBMS_OUTPUT.put_line('14*->'||RPAD('This is Cool', 15, '!'));
5
6  -- Function RPAD() 2 --
7  DBMS_OUTPUT.put_line('15*->'||RPAD('This is Cool', 19, 'MA'));
8
9  -- Function RTRIM() 1 --
10 DBMS_OUTPUT.put_line('16*->'||RTRIM('The White House has a many tree!      '));
11
12 -- Function RTRIM() 2 --
13 DBMS_OUTPUT.put_line('17*->'||RTRIM('The White House has a many tree!xxxxxx', 'x'));
14
15 -- Function RTRIM() 3 --
16 DBMS_OUTPUT.put_line('18*->'||RTRIM('The White House has a many tree!xyxyxyxyxy',
'xy'));
17
18 -- Function RTRIM() 4 --
19 DBMS_OUTPUT.put_line('19*->'||RTRIM('The White House has a many tree!xyxyxxxxxy',
'xy'));
20
21 -- Function SUBSTR() 1 --
22 DBMS_OUTPUT.put_line('20*->'||SUBSTR('Braun Fox',1,5));
23
24 -- Function SUBSTR() 2 --
25 DBMS_OUTPUT.put_line('21*->'||SUBSTR('Braun Fox',-3,3));
26
27 -- Function SUBSTR() 3 --
28 DBMS_OUTPUT.put_line('22*->'||SUBSTR('Braun Fox',3));
29
30 -- Function TRANSLATE() 1 --
31 DBMS_OUTPUT.put_line('23*->'||TRANSLATE('KrotPups', 'Pups', 'Boss'));
32
33 -- Function TRANSLATE() 2 --
34 DBMS_OUTPUT.put_line('24*->'||TRANSLATE('KrotPups', 'KrotPups', 'Kolobok'));
35
36 -- Function UPPER() --
37 DBMS_OUTPUT.put_line('25*->'||UPPER('THE quick bROwn Fox jumped oVer THE LaZy dOG ...
'));
38
39 END;
40 /
14*->This is Cool!!!
15*->This is CoolMAMAMAM
16*->The White House has a many tree!
17*->The White House has a many tree!
18*->The White House has a many tree!
19*->The White House has a many tree!
20*->Braun
21*->Fox
22*->aun Fox
23*->KrotBoss
24*->Kolobok
25*->THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG ...
```


Процедура PL/SQL успешно завершена.

RPAD(строка 1, x, строка 2)

Вот еще интересная функция! :) Возвращает "строка 1", дополненную справа до размера **x** символами "строка 2". Если размер "строка 2" меньше **x**, то при необходимости она дублируется. Если размер "строка 2" больше **x** то берутся только первые **x** ее символов. Если "строка 2" не указана, то ее заменяют символы пробела. Пример, строка 5*, 6*. Обратите внимание, что **x** указывается как размер строки символов, отображаемой на экране, а не как реальный размер. Пример 14*, 15*.

RTRIM(строка 1, строка 2)

Возвращает "строка 1", в которой удалены крайние правые символы, идентичные символам "строка 2". Значением по умолчанию для "строка 2", является знак пробела. "строка 1" просматривается с левого края, и при встрече первого символа не совпадающего с "строка 2", возвращается результат. Пример строка 16*, 17*, 18*, 19*.

SUBSTR(строка 1, a, [,b])

Вот с этой функцией, я наиболее часто работал, очень удобная штучка! Возвращает часть "строка 1", начинающуюся с символа с номером **a**, и имеющую длину **b** символов. Если **a = 0**, это равносильно тому, что **a = 1** (начало строки) если **b** положительно возвращаются символы слева направо. Если **b** отрицательно то, начиная с конца строки и считаются справа налево! Если **b** отсутствует, то по умолчанию возвращаются все символы, до конца строки. Если **b** меньше 1, то возвращается значение **NULL**. Если в качестве **a** и **b**, указано число с плавающей точкой, его дробная часть отбрасывается! Пример строка 20*, 21*, 22*.

TRANSLATE(строка_символов, заменяемая_строка, вносимая_строка)

Возвращает "строка_символов", в которой все вхождения каждого символа "заменяемая_строка" замещаются соответствующим символом "вносимая_строка". Функция **TRANSLATE**, является расширением функции **REPLACE**. Если "заменяемая_строка" длиннее чем "вносимая_строка", все ее лишние символы удаляются поскольку для них нет соответствующих символов во "вносимая_строка". "вносимая_строка" не может быть пустой. **Oracle** интерпретирует пустую строку как значение **NULL**, а если любой аргумент функции **TRANSLATE** является **NULL**, то результат тоже будет **NULL**. Пример строка 23*, 24*.

UPPER(строка)

Возвращает "строка", в которой все символы прописные. Символы не являющиеся буквами не изменяются. Пример, строка 4*. Пример строка 25*.

Вот, с символьными функциями возвращающими символьные значения пока все. Далее продолжим, следующую группу функций **PL/SQL**. Пробуйте!

Шаг 67 - PL/SQL - встроенные функции, часть 3

Итак, продолжаем разбирать встроенные функции! Немного занудно конечно! Но, что поделать, зато полезно! Вообще можете использовать это как справочник! Когда надо почитали! :)

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
-- Function ASCII() --
DBMS_OUTPUT.put_line('26*-> '||TO_CHAR(ASCII('A'))||' '||TO_CHAR(ASCII(' '))||'
'||TO_CHAR(ASCII('F')));

-- Function INSTR() --
DBMS_OUTPUT.put_line('27*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 1, 2));

-- Function INSTR() --
DBMS_OUTPUT.put_line('28*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', -1, 2));

-- Function INSTR() --
DBMS_OUTPUT.put_line('29*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 8));

-- Function INSTR() --
DBMS_OUTPUT.put_line('30*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 18));

-- Function LENGTH() --
DBMS_OUTPUT.put_line('31*-> '||TO_CHAR(LENGTH('Hello World!!!')));

-- Function LENGTH() --
DBMS_OUTPUT.put_line('32*-> '||TO_CHAR(LENGTH('Miller is very ill!')));

DBMS_OUTPUT.put_line('Part2 Number function.');
```

```
-- Function ABS() --
DBMS_OUTPUT.put_line('33*-> '||TO_CHAR(ABS(-5))||' '||TO_CHAR(ABS(5)));

-- Function ACOS() ASIN() --
DBMS_OUTPUT.put_line('34*-> '||TO_CHAR(ACOS(0.5))||' '||TO_CHAR(ASIN(0.7)));

END;
/
```

После запуска, смотрим результат:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3  -- Function ASCII() --
  4  DBMS_OUTPUT.put_line('26*-> '||TO_CHAR(ASCII('A'))||' '||TO_CHAR(ASCII(' '))||'
'||TO_CHAR(ASCII('F')));
  5
```

```
6  -- Function INSTR() --
7  DBMS_OUTPUT.put_line('27*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 1, 2)));
8
9  -- Function INSTR() --
10 DBMS_OUTPUT.put_line('28*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', -1, 2)));
11
12 -- Function INSTR() --
13 DBMS_OUTPUT.put_line('29*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 8)));
14
15 -- Function INSTR() --
16 DBMS_OUTPUT.put_line('30*-> '||TO_CHAR(INSTR('Miller is very ill!', 'il', 18)));
17
18 -- Function LENGTH() --
19 DBMS_OUTPUT.put_line('31*-> '||TO_CHAR(LENGTH('Hello World!!!')));
20
21 -- Function LENGTH() --
22 DBMS_OUTPUT.put_line('32*-> '||TO_CHAR(LENGTH('Miller is very ill!')));
23
24 DBMS_OUTPUT.put_line('Part2 Number function.');
```

25

```
26 -- Function ABS() --
27 DBMS_OUTPUT.put_line('33*-> '||TO_CHAR(ABS(-5))||' '||TO_CHAR(ABS(5)));
28
29 -- Function ACOS() ASIN() --
30 DBMS_OUTPUT.put_line('34*-> '||TO_CHAR(ACOS(0.5))||' '||TO_CHAR(ASIN(0.7)));
31
32 END;
33 /
26*-> 65 32 70
27*-> 16
28*-> 2
29*-> 16
30*-> 0
31*-> 14
32*-> 19
Part2 Number function.
33*-> 5 5
34*-> 1,04719755119659774615421446109316762805
,7753974966107530637403533527149871135488
```

Процедура PL/SQL успешно завершена.

Итак, поехали:

ASCII(строка)

Возвращает десятичное представление первого байта "строка", согласно применяемому набору символов. Пример 26*.

INSTR(строка 1, строка 2, [a[,b]])

А, вот эта функция, просто так не разберешься! :) Но, полезная когда поймешь, как она действует. Итак! Возвращает местоположение "строка 2", в "строка 1". "строка 1" просматривается слева, начиная с позиции **a**. Если **a** отрицательно, то "строка 1", просматривается справа. Возвращается позиция указывающая местоположение **b**-го вхождения. Значением по умолчанию, как для **a** так и для **b** является 1, что дает в результате позицию,

первого вхождения, "строка 2", в "строка 1". Если при заданных **a** и **b**, "строка 2" не найдена, возвращается 0. Пример 27*, 28*, 29*, 30*.

LENGTH(строка)

Очень полезная функция! :) Возвращает размер "строка" в символах. Значения типа **CHAR** дополняются пробелами. по этому если "строка" имеет тип **CHAR**, в размере указываются и конечные пробелы. Если "строка", является **NULL** - значением, то и возвращается **NULL**!
Пример 31*, 32*.

С символьными функциями пока все! Переходим к числовым. Они, может и редко будут вами применяться, но знать их не помешает! К слову, эти функции в качестве аргументов используют и возвращают значения типа **NUMBER**!

ABS(x)

Возвращает абсолютное значение для **x**. Пример 33*.

ACOS(x) ASIN(x)

Возвращает арккосинус и арксинус для **x** соответственно. Обратите внимание на количество значащих после запятой! Пример 34*.

Шаг 68 - PL/SQL - встроенные функции, часть 4

Продолжаем наше справочное руководство! Благо оно нам пригодится и иметь его не помешает!

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
-- Function ATAN() TAN() --
DBMS_OUTPUT.put_line('35*-> '||TO_CHAR(ATAN(0.32))||' '||TO_CHAR(TAN(-43 * 3.14159)));

-- Function CEIL() --
DBMS_OUTPUT.put_line('36*-> '||TO_CHAR(CEIL(17.134))||' '||TO_CHAR(CEIL(-17.134)));

-- Function COS() SIN() --
DBMS_OUTPUT.put_line('37*-> '||TO_CHAR(COS(90))||' '||TO_CHAR(SIN(60)));

-- Function EXP() --
DBMS_OUTPUT.put_line('38*-> '||TO_CHAR(EXP(1))||' '||TO_CHAR(EXP(3.456)));

-- Function LN() LOG() --
DBMS_OUTPUT.put_line('39*-> '||TO_CHAR(LN(100))||' '||TO_CHAR(LOG(5, 31))||'
'||TO_CHAR(LOG(2, 12)));

-- Function MOD() --
DBMS_OUTPUT.put_line('40*-> '||TO_CHAR(MOD(34, 8))||' '||TO_CHAR(MOD(45, 7)));

-- Function POWER() --
DBMS_OUTPUT.put_line('41*-> '||TO_CHAR(POWER(8, 4))||' '||TO_CHAR(POWER(65, -7)));

-- Function ROUND() --
DBMS_OUTPUT.put_line('42*-> '||TO_CHAR(ROUND(2.57))||' '||TO_CHAR(ROUND(3.678, 1))||'
'||TO_CHAR(ROUND(14.8, -2)));

-- Function SIGN() --
DBMS_OUTPUT.put_line('43*-> '||TO_CHAR(SIGN(-4))||' '||TO_CHAR(SIGN(0))||'
'||TO_CHAR(SIGN(6)));

-- Function SQRT() --
DBMS_OUTPUT.put_line('44*-> '||TO_CHAR(SQRT(98))||' '||TO_CHAR(SQRT(9)));

-- Function TRUNC() --
DBMS_OUTPUT.put_line('45*-> '||TO_CHAR(TRUNC(-123.456))||'
'||TO_CHAR(TRUNC(123.456,1))||' '||TO_CHAR(TRUNC(123.456, -1)));

END;
/
```

Получаем после запуска:

```
SQL> SET SERVEROUTPUT ON
SQL>
```

```

SQL> BEGIN
  2
  3      -- Function ATAN() TAN() --
  4      DBMS_OUTPUT.put_line('35*-> '||TO_CHAR(ATAN(0.32))||' '||TO_CHAR(TAN(-43 *
3.14159)));
  5
  6      -- Function CEIL() --
  7      DBMS_OUTPUT.put_line('36*-> '||TO_CHAR(CEIL(17.134))||' '||TO_CHAR(CEIL(-
17.134)));
  8
  9      -- Function COS() SIN() --
 10      DBMS_OUTPUT.put_line('37*-> '||TO_CHAR(COS(90))||' '||TO_CHAR(SIN(60)));
 11
 12      -- Function EXP() --
 13      DBMS_OUTPUT.put_line('38*-> '||TO_CHAR(EXP(1))||' '||TO_CHAR(EXP(3.456)));
 14
 15      -- Function LN() LOG() --
 16      DBMS_OUTPUT.put_line('39*-> '||TO_CHAR(LN(100))||' '||TO_CHAR(LOG(5, 31))||'
'||TO_CHAR(LOG(2, 12)));
 17
 18      -- Function MOD() --
 19      DBMS_OUTPUT.put_line('40*-> '||TO_CHAR(MOD(34, 8))||' '||TO_CHAR(MOD(45, 7)));
 20
 21      -- Function POWER() --
 22      DBMS_OUTPUT.put_line('41*-> '||TO_CHAR(POWER(8, 4))||' '||TO_CHAR(POWER(65, -
7)));
 23
 24      -- Function ROUND() --
 25      DBMS_OUTPUT.put_line('42*-> '||TO_CHAR(ROUND(2.57))||' '||TO_CHAR(ROUND(3.678,
1))||' '||TO_CHAR(ROUND(14.8, -2)));
 26
 27      -- Function SIGN() --
 28      DBMS_OUTPUT.put_line('43*-> '||TO_CHAR(SIGN(-4))||' '||TO_CHAR(SIGN(0))||'
'||TO_CHAR(SIGN(6)));
 29
 30      -- Function SQRT() --
 31      DBMS_OUTPUT.put_line('44*-> '||TO_CHAR(SQRT(98))||' '||TO_CHAR(SQRT(9)));
 32
 33      -- Function TRUNC() --
 34      DBMS_OUTPUT.put_line('45*-> '||TO_CHAR(TRUNC(-123.456))||'
'||TO_CHAR(TRUNC(123.456,1))||' '||TO_CHAR(TRUNC(123.456, -1)));
 35
 36 END;
 37 /
35*-> ,3097029445424561999173808103924156700914
,000114104361604459415200698613184275542783
36*-> 18 -17
37*-> -,44807361612917015236547731439963950742 -
,30481062110221670562564946547842584078
38*-> 2,71828182845904523536028747135266249776
31,68996280537916473883871110045120692647
39*-> 4,6051701859880913680359829093687284152
2,13365621497732264414203154309561510056 3,58496250072115618145373894394781650873

```

```
40*-> 2 3
41*-> 4096 ,000000000000203988884709418710246172033037848527959
42*-> 3 3,7 0
43*-> -1 0 1
44*-> 9,89949493661166534161182106946788654999 3
45*-> -123 123,4 120
```

Процедура PL/SQL успешно завершена.

ATAN(x) TAN(x)

Возвращает арктангенс и тангенс **x** соответственно. Пример 35*.

CEIL(x)

Возвращает наименьшее целое число, большее или равное **x**. Пример 36*.

COS(x) SIN(x)

Возвращает косинус и синус **x** соответственно. Пример 37*.

EXP(x)

Возвращает **e** в степени **x**, где **e** = (2,718281828 основание натурального логарифма). Пример 38*.

LN(x)

Возвращает натуральный логарифм **x**, Пример 39*.

LOG(y, x)

Возвращает логарифм **y** по основанию **x**. Основание должно быть положительным числом, отличным от 0 или 1, а **y** может быть любым положительным числом. Пример 39*.

MOD(x, y)

Возвращает остаток от деления **x** нацело на **y**. Если **y** равно 0, то возвращается **x**. Пример 40*.

POWER(x, y)

Возвращает **x** в степени **y**. Основание **x** и порядок **y** могут быть не положительными целыми числами, но если **x** - отрицательное число, то **y** должен быть целым числом. Пример 41*.

ROUND(x, [,y])

Возвращает **x** округленное до **y** разрядов справа от десятичной точки. Значением по умолчанию для **y** является 0, при этом **x** округляется до ближайшего целого числа. Если **y** - отрицательное число, то округляются цифры слева от десятичной точки. **y** должен быть целым числом. Пример 42*.

SIGN(x)

Если **x** < 0 возвращает -1, **x** = 0 возвращает 0, **x** > 0 возвращает 1. Пример 43*.

SQRT(x)

Возвращает квадратный корень **x**. Значение **x** не может быть отрицательным! Пример 44*.

TRUNC(x, [,y])

Возвращает **x** усеченное (не округленное !) до **y** десятичных разрядов. Значением по умолчанию, для **y** является 0, при этом **x** усекается до целого числа. Если **y** отрицательно, то усекаются цифры слева от десятичной точки. Пример 45*.

С числовыми функциями пока все, далее продолжим. Пробуйте! :)

Шаг 69 - PL/SQL - встроенные функции, дата и время

Продолжаем наши справочники! Итак, определим такой блок:

```

SET SERVEROUTPUT ON

BEGIN

-- Function ADD_MONTHS() --
DBMS_OUTPUT.put_line('46*-> '||ADD_MONTHS('09-02-2000', 5));

-- Function ADD_MONTHS() --
DBMS_OUTPUT.put_line('47*-> '||ADD_MONTHS('18-06-2001', 3));

-- Function ADD_MONTHS() --
DBMS_OUTPUT.put_line('48*-> '||ADD_MONTHS('12-02-1981', 2));

-- Function LAST_DAY() --
DBMS_OUTPUT.put_line('49*-> '||LAST_DAY('09-02-2000')||' '||LAST_DAY('19-02-2001'));

-- Function LAST_DAY() --
DBMS_OUTPUT.put_line('50*-> '||LAST_DAY('05-03-2002'));

-- Function MONTHS_BETWEEN() --
DBMS_OUTPUT.put_line('51*-> '||MONTHS_BETWEEN('16-04-1973', '16-03-1997'));

-- Function MONTHS_BETWEEN() --
DBMS_OUTPUT.put_line('52*-> '||MONTHS_BETWEEN('18-04-1972', '23-03-1961'));

-- Function NEW_TIME() --
DBMS_OUTPUT.put_line( '53*-> '||TO_CHAR(NEW_TIME(TO_DATE('12-04-1971 12:00:00', 'DD-
MM-YYYY HH24:MI:SS'),
        'PST', 'EST'), 'DD-MON-YYYY HH24:MI:SS')||': Pacific -> Eastern' );

-- Function NEXT_DAY() --
DBMS_OUTPUT.put_line( '54*-> '||NEXT_DAY('07-04-2003', 'Понедельник'));

-- Function NEXT_DAY() --
DBMS_OUTPUT.put_line( '55*-> '||NEXT_DAY('01-03-1998', 'Четверг'));

END;
/

```

После запуска в **SQL*Plus** получаем:

```

SQL> BEGIN
2
3      -- Function ADD_MONTHS() --
4      DBMS_OUTPUT.put_line('46*-> '||ADD_MONTHS('09-02-2000', 5));
5
6      -- Function ADD_MONTHS() --
7      DBMS_OUTPUT.put_line('47*-> '||ADD_MONTHS('18-06-2001', 3));

```



```

8
9      -- Function ADD_MONTHS() --
10     DBMS_OUTPUT.put_line('48*-> '||ADD_MONTHS('12-02-1981', 2));
11
12     -- Function LAST_DAY() --
13     DBMS_OUTPUT.put_line('49*-> '||LAST_DAY('09-02-2000')||' '||LAST_DAY('19-02-
2001'));
14
15     -- Function LAST_DAY() --
16     DBMS_OUTPUT.put_line('50*-> '||LAST_DAY('05-03-2002'));
17
18     -- Function MONTHS_BETWEEN() --
19     DBMS_OUTPUT.put_line('51*-> '||MONTHS_BETWEEN('16-04-1973', '16-03-1997'));
20
21     -- Function MONTHS_BETWEEN() --
22     DBMS_OUTPUT.put_line('52*-> '||MONTHS_BETWEEN('18-04-1972', '23-03-1961'));
23
24     -- Function NEW_TIME() --
25     DBMS_OUTPUT.put_line( '53*-> '||TO_CHAR(NEW_TIME(TO_DATE('12-04-1971
12:00:00', 'DD-MM-YYYY HH24:MI:SS'),
26                'PST', 'EST'), 'DD-MON-YYYY HH24:MI:SS')||': Pacific -> Eastern' );
27
28     -- Function NEXT_DAY() --
29     DBMS_OUTPUT.put_line( '54*-> '||NEXT_DAY('07-04-2003', 'Понедельник'));
30
31     -- Function NEXT_DAY() --
32     DBMS_OUTPUT.put_line( '55*-> '||NEXT_DAY('01-03-1998', 'Четверг'));
33
34 END;
35 /
46*-> 09.07.00
47*-> 18.09.01
48*-> 12.04.81
49*-> 29.02.00 28.02.01
50*-> 31.03.02
51*-> -287
52*-> 132,83870967741935483870967741935483871
53*-> 12-АПП-1971 15:00:00: Pacific -> Eastern
54*-> 14.04.03
55*-> 05.03.98

```

Процедура PL/SQL успешно завершена.

Если у кого-то, данный блок вызвал ошибки ничего страшного нет, просто у вас скорее всего не выставлен язык системы, об этом мы уже говорили. А, так же формат даты в самой **Windows** установите **dd.MM.YYYY**. Я думаю, что ошибок не будет. Теперь, давайте разберемся с функциями. Данные функции, работают с датами и возвращают дату, за исключением функции **MONTHS_BETWEEN**, которая возвращает, значение типа **NUMBER**. Я считаю этот блок функций очень интересным и полезным для дальнейшего применения.

ADD_MONTHS(d,x)

Возвращает дату **d** плюс **x** месяцев. Значение **x** может быть любым целым числом. Если в месяце, полученном в результате, число дней меньше, чем в месяце **d** то, возвращается

последний день месяца результата. Если, число дней не меньше то день месяца-результата и день месяца **d** совпадают. Временные компоненты даты **d** и результата одинаковы. Пример 46*, 47*, 48*.

LAST_DAY(d)

Возвращает дату последнего дня того месяца, в который входит **d**. Эту функцию, можно применять для определения количества дней оставшихся в текущем месяце. Пример 49*, 50*.

MONTHS_BETWEEN(дата 1, дата 2)

Возвращает число месяцев между "дата 1" и "дата 2". Если дни в "дата 1" и "дата 2" или если обе даты являются последними днями своих месяцев. То, результат представляет собой целое число. В противном случае результат будет содержать дробную часть, по отношению, к 31-дневному месяцу. Пример 51*, 52*.

NEW_TIME(d, пояс 1, пояс 2)

Вот, долго думал, показывать эту функцию или нет, решил ладно, пусть будет. Хотя, я ей еще не пользовался, больно она, какая то... а в прочем решать вам! Итак:

Возвращает дату и время, часового пояса 2 для того момента когда, датой и временем, часового пояса 1 является **d**. Где пояс 1, пояс 2 это строки символов, для поясного времени Америки. В примере 'PST' и 'EST' это Тихоокеанское поясное время и восточное поясное время соответственно. Точно не знаю, есть ли эти строки для России, но вообще, в этом случае, я применял другие способы определения времени по поясам, как правило, в клиентских приложениях, например, применив **C++**. Но может, у кого есть другое мнение на сей счет! Кстати в примере показаны, функции преобразования, с которыми вы еще не сталкивались, по этому, пока можете сами разобраться. Если не совсем понятно, я о них еще расскажу! :) Пример 53*;

NEXT_DAY(d, строка_символов)

Возвращает дату первого дня, наступающего после даты **d** и обозначенного строкой символов. Строка символов указывает день недели на языке текущего сеанса. (Вот с 8.1.5 иногда бывают проблемы с передачей русских слов например, в сеанс **SQL*Plus**, но на удивление 9.0.2 они полностью отсутствуют!) Временной компонент возвращаемого значения тот же, что и временной компонент **d**. Регистр символов строки значения не имеет. Пример 54*, 55*.

Шаг 70 - PL/SQL - встроенные функции, даты и снова даты

Пишем наш маленький справочник далее, вот еще несколько функций работы с датами. Запишем такой блок:

```
SET SERVEROUTPUT ON

BEGIN

-- Function ROUND() --
DBMS_OUTPUT.put_line( '56*-> '||ROUND(TO_DATE('15-05-1998'), 'MM'));

-- Function ROUND() --
DBMS_OUTPUT.put_line( '57*-> '||ROUND(TO_DATE('15-05-1998'), 'DY'));

-- Function ROUND() --
DBMS_OUTPUT.put_line( '58*-> '||ROUND(TO_DATE('15-05-1998'), 'WW'));

-- Function TRUNC() --
DBMS_OUTPUT.put_line( '59*-> '||TO_CHAR(TRUNC(TO_DATE('15-05-1977 15:34:12',
'DD-MM-YYYY HH24:MI:SS'), 'YEAR'), 'DD-MM-YYYY HH24:MI:SS'));

-- Function TRUNC() --
DBMS_OUTPUT.put_line( '60*-> '||TO_CHAR(TRUNC(TO_DATE('12-04-1999 10:24:42',
'DD-MM-YYYY HH24:MI:SS'), 'MM'), 'DD-MM-YYYY HH24:MI:SS'));

-- Function TRUNC() --
DBMS_OUTPUT.put_line( '61*-> '||TO_CHAR(TRUNC(TO_DATE('02-09-2003 05:22:55',
'DD-MM-YYYY HH24:MI:SS'), 'IW'), 'DD-MM-YYYY HH24:MI:SS'));

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '62*-> Now Time is: '||TO_CHAR(SYSDATE, 'MONTH DD YYYY
HH24:MI:SS'));

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '63*-> Now Time is: '||TO_CHAR(SYSDATE, 'DD-MM-YYYY
HH24:MI:SS'));

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3      -- Function ROUND() --
  4      DBMS_OUTPUT.put_line( '56*-> '||ROUND(TO_DATE('15-05-1998'), 'MM'));
  5
  6      -- Function ROUND() --
  7      DBMS_OUTPUT.put_line( '57*-> '||ROUND(TO_DATE('15-05-1998'), 'DY');
```

```

8
9      -- Function ROUND() --
10     DBMS_OUTPUT.put_line( '58*-> '||ROUND(TO_DATE('15-05-1998'), 'WW'));
11
12     -- Function TRUNC() --
13     DBMS_OUTPUT.put_line( '59*-> '||TO_CHAR(TRUNC(TO_DATE('15-05-1977 15:34:12',
14         'DD-MM-YYYY HH24:MI:SS'), 'YEAR'), 'DD-MM-YYYY HH24:MI:SS'));
15
16     -- Function TRUNC() --
17     DBMS_OUTPUT.put_line( '60*-> '||TO_CHAR(TRUNC(TO_DATE('12-04-1999 10:24:42',
18         'DD-MM-YYYY HH24:MI:SS'), 'MM'), 'DD-MM-YYYY HH24:MI:SS'));
19
20     -- Function TRUNC() --
21     DBMS_OUTPUT.put_line( '61*-> '||TO_CHAR(TRUNC(TO_DATE('02-09-2003 05:22:55',
22         'DD-MM-YYYY HH24:MI:SS'), 'IW'), 'DD-MM-YYYY HH24:MI:SS'));
23
24     -- Function SYSDATE() --
25     DBMS_OUTPUT.put_line( '62*-> Now Time is: '||TO_CHAR(SYSDATE, 'MONTH DD YYYY
HH24:MI:SS'));
26
27     -- Function SYSDATE() --
28     DBMS_OUTPUT.put_line( '63*-> Now Time is: '||TO_CHAR(SYSDATE, 'DD-MM-YYYY
HH24:MI:SS'));
29
30 END;
31 /
56*-> 01.05.98
57*-> 18.05.98
58*-> 14.05.98
59*-> 01-01-1977 00:00:00
60*-> 01-04-1999 00:00:00
61*-> 01-09-2003 00:00:00
62*-> Now Time is: НОЯБРЬ 15 2003 19:50:58
63*-> Now Time is: 15-11-2003 19:50:58

```

Процедура PL/SQL успешно завершена.

Итак, начинаем по немногу разбираться.

ROUND(d, [, формат])

Округляет дату **d**, до единицы указанной форматом. Формат соответствует следующей таблице (она же действует и для функции **TRUNC**):

Формат	Единица округления или усечения
CC, SCC	Век
YYYY, YYYY, YEAR, SYEAR, YY, Y	Год (округляется до 1 июля)
IYYY, IYY, IY, I	Год ISO
Q	Квартал (округляется до шестнадцатого дня второго месяца квартала)
MONTH, MON, MM, RM	Месяц (округляется до шестнадцатого дня)

WW	То же день недели, что и первый день года
IW	То же день недели, что и первый день года ISO
W	То же день недели, что и первый день месяца
DDD, DD, J	День
Day, DY, D	Первый день недели
HH, HH12, HH24	Час
MI	Минута

Если "формат" не указан, применяется формат по умолчанию **'DD'**, который округляет **d** до ближайшего дня. Пример 56*, 57*, 58*.

TRUNC(d, [, формат])

Возвращает дату **d** усеченную до единицы, указанной "формат". Если "формат" не указан, применяется формат по умолчанию **'DD'**, который усекает **d** до ближайшего дня. Пример 59*, 60*, 61*.

SYSDATE

Вот, одна из наиболее полезных функций, которая есть в **PL/SQL**. Возвращает текущую дату и время в системе. Возвращаемый формат **DATE**. В примерах 62* и 63* хорошо видно как можно изменить выводимый тип для **SYSDATE**! Эти маски для преобразования, мы еще рассмотрим, хотя я уже почти все их показал! :)

Так, же в **PL/SQL** предусмотрены некоторые арифметические действия с датами, давайте рассмотрим основные из них:

Операция	Тип возвращаемого значения	Описание
d1 - d2	NUMBER (!)	Возвращается разница в днях между d1 и d2. Это значение является действительным числом, где дробная часть означает неполный день.
d1 + d2	-	ЗАПРЕЩЕНА (Сколько, я функций написал из за этого!!!)
d1 + n	DATE	К d1 добавляется n дней и возвращается результат, имеющий тип DATE. n может быть вещественным числом содержащим не полный день.
d1 - n	DATE	Из d1 вычитается n дней и возвращается результат, имеющий тип DATE. n может быть вещественным числом содержащим не полный день.

Давайте посмотрим несколько примеров:

```
DECLARE
```

```
D1 DATE;  
D2 DATE;
```

```
BEGIN
```

```
D1 := TO_DATE('12-10-2003 15:10:34', 'DD-MM-YYYY HH24:MI:SS');
```

```
D2 := TO_DATE('11-03-2003 11:11:22', 'DD-MM-YYYY HH24:MI:SS');

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '64*-> Today: '||TO_CHAR(SYSDATE, 'MONTH DD YYYY'));

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '65*-> Tomorrow: '||TO_CHAR(SYSDATE - 1, 'MONTH DD YYYY'));

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '66*-> Return to The Future: '||TO_CHAR(SYSDATE + 1, 'MONTH DD YYYY'));

-- Function SYSDATE() --
DBMS_OUTPUT.put_line( '67*-> Today: '||TO_CHAR(D1 - D2));

END;
/
```

После запуска в **SQL*Plus** получаем:

```
SQL> DECLARE
2
3 D1 DATE;
4 D2 DATE;
5
6 BEGIN
7
8     D1 := TO_DATE('12-10-2003 15:10:34', 'DD-MM-YYYY HH24:MI:SS');
9     D2 := TO_DATE('11-03-2003 11:11:22', 'DD-MM-YYYY HH24:MI:SS');
10
11 -- Function SYSDATE() --
12 DBMS_OUTPUT.put_line( '64*-> Today: '||TO_CHAR(SYSDATE, 'MONTH DD YYYY'));
13
14 -- Function SYSDATE() --
15 DBMS_OUTPUT.put_line( '65*-> Tomorrow: '||TO_CHAR(SYSDATE - 1, 'MONTH DD
YYYY'));
16
17 -- Function SYSDATE() --
18 DBMS_OUTPUT.put_line( '66*-> Return to The Future: '||TO_CHAR(SYSDATE + 1,
'MONTH DD YYYY'));
19
20 -- Function SYSDATE() --
21 DBMS_OUTPUT.put_line( '67*-> Today: '||TO_CHAR(D1 - D2));
22
23 END;
24 /
64*-> Today: ОЯБРЬ 15 2003
65*-> Tomorrow: ОЯБРЬ 14 2003
66*-> Return to The Future: ОЯБРЬ 16 2003
67*-> Today: 215,1661111111111111111111111111111111111111111111111
```

Процедура PL/SQL успешно завершена.

64* Показывает текущую дату, 65* возвращает нас в прошлое, (чем не машина времени!) 66* отправляет нас в будущее (Г. Уэллс просто отдыхает!) А, последний пример показывает разницу двух дат, в формате **NUMBER**! Вот так работают функции обработки дат! Пробуйте сами!!! :)

Шаг 71 - PL/SQL - функции, преобразования TO_CHAR(...) даты

В [прошлый раз](#) мы рассмотрели уже достаточно большое количество функций, для работы с разнообразными данными в среде **PL/SQL**. А, вот сейчас пришло-таки время внимательнее рассмотреть те самые функции преобразования, с которыми мы с вами уже сталкивались, еще в самом начале пути. Итак, что же это за функции и как они устроены. Как правило, при работе с разными типами данных **PL/SQL** сам неявно вызывает функции преобразования и вам не приходится самим что-то делать. Но **PL/SQL** использует их, с типом преобразования по умолчанию. Например, запишем такой блок:

```
SET SERVEROUTPUT ON

BEGIN

DBMS_OUTPUT.put_line(SYSDATE);

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3 DBMS_OUTPUT.put_line(SYSDATE);
  4
  5 END;
  6 /
16.11.03
```

Процедура PL/SQL успешно завершена.

Такой, тип вывода нас не всегда устраивает, по этому у функции **TO_CHAR** есть множество форматов вывода данных, которые можно применять используя явное преобразование или, проще говоря, ее вызов. Функция **TO_CHAR** имеет следующий синтаксис:

TO_CHAR(d, [, формат [nls_параметр]])

Где **d**, это данные типа **DATE**, а "формат" - это строка, которая состоит из нескольких или одного элементов, описанных ниже в таблице. При этом, если "формат" не указан, то как вы помните, используется формат по умолчанию для конкретного сеанса. "nls_параметр" - управляет выбором языка, но, как правило, достаточно языка по умолчанию, например я этот параметр использовал очень редко. Далее можете посмотреть на таблицу форматов и пример с применением данных из таблицы.

Элемент формата дат	Описание
Знак пунктуации	Все знаки пунктуации дублируются в результирующей строке символов

"Текст"	Текст, заключенный в двойные кавычки так же дублируется
AD, A.D.	Показатель "нашей эры" (с точками или без точек.)
AM, A.M.	Показатель времени до полудня (с точками или без точек.)
BC, B.C.	Показатель "до нашей эры" (с точками или без точек.)
CC, SCC	Век SCC возвращает даты "до нашей эры" как отрицательные значения
D	День недели (1-7)
DAY	Название дня, дополненное пробелами до девяти символов. *
DD	День месяца (1 - 31)
DDD	День года (1 - 366)
DY	Сокращенное название дня. *
IW	Неделя года (1 - 52), (1 - 53) (в основе лежит стандарт ISO)
IYY, IY, I	Последние три, две или одна цифра года ISO
IYYY	Четырех цифровое обозначение года, основанное на стандарте ISO
HH, HH12	Час дня (1-12)
HH24	Час дня (0-23)
J	День по Юлианскому календарю. Число дней с 1 января 4712 года до н.э. Соответствующий результат будет целым значением.
MI	Минута (0-59)
MM	Месяц (1 - 12), JAN = 1, DEC = 12
MONTH	Название месяца, дополненное пробелами до девяти символов. *
MON	Сокращенное название месяца. *
PM P.M.	Показатель времени после полудня (с точками или без точек.)
Q	Квартал года (1 - 4) С января по март - первый квартал.
RM	Месяц, обозначенный римскими цифрами. (I - XII) JAN = I, DEC = XII
RR	Последние две цифры года для других веков.
SS	Секунды (0-59)
SSSS	Секунды после полуночи (0 - 86399) Модель формата 'J.SSSSS' всегда будет давать в результате числовое значение.
WW	Неделя года (1-53). Неделя 1 начинается с первого дня года и продолжается до седьмого дня. таким образом, недели не всегда начинаются с воскресенья как это принято в США, это что то вроде сквозного недельного отсчета.
W	Неделя месяца (1-5) Недели определяются, так же как и WW!
Y, YYY	Год с запятой в указанной позиции.
YEAR, SYEAR	Год, записанный буквами, SYEAR возвращает даты до нашей эры как отрицательные значения. *
YYYY, SYYYY	Четырех цифровой год. Возвращает даты до нашей эры как отрицательные значения.
YYY, YY, Y	Последние три, две или одна цифра года.

Вот блок примера, где я постарался применить разнообразные форматы вывода функции **TO_CHAR()**. Для получения данных с типом **DATE** я применил функцию **SYSDATE**, с которой вы уже знакомы:

```
SET SERVEROUTPUT ON

BEGIN

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MM-YY'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MM-YYYY A.D.));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-J-MM-YYYY A.D.));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'D-DAY-DD-MON, YYYY'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DAY DDD MONTH YYYY'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DAY-MONTH-YYYY'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'Q-DD-RM-YYYY '));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-WW-YYYY HH24:MI:SS'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-W-YYYY HH24:MI:SS'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-YEAR HH24:MI:SS'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH24:MI:SS'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH:MI:SS AM'));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH12:MI:SS P.M.));

  -- Function TO_CHAR() --
  DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH12:MI:SS:J.SSSSS P.M.));

END;
/
```

После **SQL*Plus** получаем:

```
SQL> BEGIN
2
```

```

3      -- Function TO_CHAR() --
4      DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MM-YY'));
5
6      -- Function TO_CHAR() --
7      DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MM-YYYY A.D.));
8
9      -- Function TO_CHAR() --
10     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-J-MM-YYYY A.D.));
11
12     -- Function TO_CHAR() --
13     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'D-DAY-DD-MON, YYYY'));
14
15     -- Function TO_CHAR() --
16     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DAY DDD MONTH YYYY'));
17
18     -- Function TO_CHAR() --
19     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DAY-MONTH-YYYY'));
20
21     -- Function TO_CHAR() --
22     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'Q-DD-RM-YYYY '));
23
24     -- Function TO_CHAR() --
25     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-WW-YYYY HH24:MI:SS'));
26
27     -- Function TO_CHAR() --
28     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-W-YYYY HH24:MI:SS'));
29
30     -- Function TO_CHAR() --
31     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'DD-MONTH-YEAR HH24:MI:SS'));
32
33     -- Function TO_CHAR() --
34     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH24:MI:SS'));
35
36     -- Function TO_CHAR() --
37     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH:MI:SS AM'));
38
39     -- Function TO_CHAR() --
40     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH12:MI:SS P.M.));
41
42     -- Function TO_CHAR() --
43     DBMS_OUTPUT.put_line( TO_CHAR(SYSDATE, 'HH12:MI:SS:J.SSSSS P.M.));
44
45 END;
46 /
16-11-03
16-11-2003 H.3.
16-2452960-11-2003 H.3.
7-ВОСКРЕСЕНЬЕ-16-НОЯ, 2003
ВОСКРЕСЕНЬЕ 320 НОЯБРЬ 2003
ВОСКРЕСЕНЬЕ-НОЯБРЬ -2003
4-16-XI -2003
16-НОЯБРЬ -46-2003 13:13:14
16-НОЯБРЬ -3-2003 13:13:14

```

16-НОЯБРЬ -TWO THOUSAND THREE 13:13:14
13:13:14
01:13:14 PM
01:13:14 PM
01:13:14:2452960.47594 PM

Процедура PL/SQL успешно завершена.

Вот таким образом можно, применяя разнообразные форматы вывода, получать необходимую для работы структуру данных.

Для некоторых форматов при преобразовании дат есть чувствительность к регистру (они в таблице имеют значок *). Например, функция **MON** вернет **JAN**, а если записать **Mon**, то получите **Jan**!

Шаг 72 - PL/SQL - функции, преобразования TO_CHAR(...) числа

Теперь давайте рассмотрим особенности работы с функцией **TO_CHAR()** применительно к преобразованию, чисел. При выполнении этой операции функция имеет следующий синтаксис:

TO_CHAR(число, [, формат[, nls_параметр]])

При этом функция **TO_CHAR()** преобразует тип **NUMBER** в тип **VARCHAR2**. В таблице указано, какие есть форматы и как они используются при преобразовании:

Элемент формата	Пример строки данного формата	Описание
9	99	Каждая цифра 9 представляет значащую цифру результата. Число значащих цифр возвращаемого значения равно числу цифр 9, отрицательное значение предваряется знаком минуса. Все начальные нули заменяются пробелами.
0	0999	Возвращается число с начальными нулями, а не пробелами.
0	9990	Возвращается число с конечными нулями, а не пробелами.
\$	\$999	Возвращаемое значение предваряется знаком доллара не зависимо от используемого символа денежной единицы можно применить совместно с начальными или конечными нулями.
B	B999	Вместо нулевой целой части десятичного число возвращаются пробелы.
MI	999MI	Возвращает отрицательное число, у которого знак минуса указан не в начале, а в конце. В положительном значении на этом месте будет пробел.
S	S9999	Возвращаемое число предваряется знаком: + для положительных, чисел - для отрицательных.
S	9999S	Возвращаемое число заканчивается знаком: + для положительных, чисел - для отрицательных.
PR	99PR	Возвращается отрицательное число в угловых скобках "<", ">". У положительных чисел, на этом месте пробелы.
D	99D9	Возвращает число с десятичной точкой в указанной позиции. Число 9 с обеих сторон указывает максимальное число цифр.
G	9G999	Возвращает число с разделителем групп в указанной позиции. G может появляться в указанной строке формата неоднократно.
C	C99	Возвращает число с символом денежной единицы ISO в указанной позиции. C может появляться в указанной строке формата неоднократно.
L	L999	Возвращает число с символом денежной единицы национального языка в указанной позиции.
,	999,999	Возвращает число с запятой в указанной позиции, не зависимо от выбранного разделителя групп.
.	99.99	Возвращает число с десятичной точкой в указанной позиции, не

		зависимо от выбранного десятичного разделителя.
V	99V999	Возвращает число, умноженное на 10ⁿ , где n - это число цифр 9 после V. При необходимости значение округляется.
EEEE	9.99EEEE	Возвращает число в экспоненциальном представлении.
RM	RM	Возвращает число при помощи римских цифр верхнего регистра.

Для примера, приведем вот такой блок:

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(534523));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(34387, '99999'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(5000, '$9999'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(-9, '9S'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(-34, 'S99'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(3 - 5, '999MI'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(7 - 3, 'S9'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(4 - 5, '99PR'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(8900, 'L9999'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(10000000, '9.9EEEE'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(10, 'RM'));

-- Function TO_CHAR() --
DBMS_OUTPUT.put_line( TO_CHAR(105, 'RM'));

END;
/
```

После запуска получаем:

```
SQL> BEGIN
 2
 3     -- Function TO_CHAR() --
 4     DBMS_OUTPUT.put_line( TO_CHAR(534523));
 5
 6     -- Function TO_CHAR() --
 7     DBMS_OUTPUT.put_line( TO_CHAR(34387, '99999'));
 8
 9     -- Function TO_CHAR() --
10     DBMS_OUTPUT.put_line( TO_CHAR(5000, '$9999'));
11
12     -- Function TO_CHAR() --
13     DBMS_OUTPUT.put_line( TO_CHAR(-9, '9S'));
14
15     -- Function TO_CHAR() --
16     DBMS_OUTPUT.put_line( TO_CHAR(-34, 'S99'));
17
18     -- Function TO_CHAR() --
19     DBMS_OUTPUT.put_line( TO_CHAR(3 - 5, '999MI'));
20
21     -- Function TO_CHAR() --
22     DBMS_OUTPUT.put_line( TO_CHAR(7 - 3, 'S9'));
23
24     -- Function TO_CHAR() --
25     DBMS_OUTPUT.put_line( TO_CHAR(4 - 5, '99PR'));
26
27     -- Function TO_CHAR() --
28     DBMS_OUTPUT.put_line( TO_CHAR(8900, 'L9999'));
29
30     -- Function TO_CHAR() --
31     DBMS_OUTPUT.put_line( TO_CHAR(10000000, '9.9EEEE'));
32
33     -- Function TO_CHAR() --
34     DBMS_OUTPUT.put_line( TO_CHAR(10, 'RM'));
35
36     -- Function TO_CHAR() --
37     DBMS_OUTPUT.put_line( TO_CHAR(105, 'RM'));
38
39 END;
40 /
534523
34387
$5000
9-
-34
2-
+4
<1>
p.8900
1.0E+07
X
```

CV

Процедура PL/SQL успешно завершена.

И для полноты картины, давайте рассмотрим функцию **TO_NUMBER()**, так как с **TO_CHAR()** они практически близнецы, но с двух противоположных сторон. Синтаксис у **TO_NUMBER** следующий:

TO_NUMBER(строка_символов, [, формат [, nls_параметр]])

В данном случае "строка_символов" - это тип **CHAR** или **VARCHAR2**, которая применительно к "формат", преобразуется в типу **NUMBER**. "формат" в данной функции, тот же что и для **TO_CHAR()**. Вот так достаточно просто. А, если в предыдущем примере поменять, например, третью снизу строку вот так:

DBMS_OUTPUT.put_line(TO_CHAR(TO_NUMBER('10000000', '9.9EEEE'), '9.9EEEE'));

То, достаточно хорошо видно, как все взаимосвязано для этих двух функций! Пробуйте!!! :)

Шаг 73 - PL/SQL - функции PL/SQL осталось немного ...

Вот, совсем забыл (просто склероз, какой то!). При работе с датами, есть функция **TO_DATE**, которая производит преобразование обратное **TO_CHAR** при работе с типом **DATE**. Данная функция, преобразует тип **CHAR** или **VARCHAR2** в тип **DATE**. Ее синтаксис таков:

```
TO_DATE(строка_символов, [, формат [, nls_параметр]])
```

Здесь в данном случае "строка_символов" строка типа **CHAR** или **VARCHAR**. Все остальное, как и [шаге 71](#) для **TO_CHAR()**. Таблица форматов преобразования та же! Пример использования, приведен ниже:

```
SET SERVEROUTPUT ON

BEGIN

  DBMS_OUTPUT.enable;

  -- Function TO_DATE() --
  DBMS_OUTPUT.put_line( TO_CHAR(TO_DATE('12-11-2003', 'DD-MM-YYYY'), 'DD-Month-YYYY'));

END;
/
```

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3      DBMS_OUTPUT.enable;
  4
  5      -- Function TO_DATE() --
  6      DBMS_OUTPUT.put_line( TO_CHAR(TO_DATE('12-11-2003', 'DD-MM-YYYY'), 'DD-Month-
YYYY'));
  7
  8 END;
  9 /
12-Ноябрь -2003
```

Процедура PL/SQL успешно завершена.

Вот так достаточно, интересно сработал формат **Month**. Месяц выводится с первой прописной, а затем строчные символы. Итак, давайте, наконец, закончим пока с описаниями функций, от которых вы уже, наверное, подустали, но материал, бесспорно очень полезен. Подведем итог, еще несколькими функциями, которые, несомненно, вам пригодятся в дальнейшем. Одна из таких весьма полезных функций следующая:

```
NVL(выражение 1, выражение 2)
```

А, вот что она производит: Возвращает "выражение 2", если "выражение 1" является значением **NULL**. В противном случае возвращает "выражение 1". Если "выражение 1", не является строкой символов, то возвращаемое значение имеет тот же тип данных, что и "выражение 1", иначе возвращаемое. С помощью этой функции очень легко избавиться от **NULL** в активном наборе запроса. Например, если дать вот такой запрос к таблице **SALESREPS** нашей учебной БД:

```
SELECT s.NAME, NVL(s.MANAGER, 0) MANAGER, NVL(s.QUOTA, 0) QUOTA
FROM SALESREPS s
/
```

В результате получим следующее:

```
SQL> SELECT s.NAME, NVL(s.MANAGER, 0) MANAGER, NVL(s.QUOTA, 0) QUOTA
2 FROM SALESREPS s
3 /
```

NAME	MANAGER	QUOTA
Вася Пупкин	104	350
Маша Распутина	106	300
Филип Киркоров	108	350
Света Разина	0	275
Наташа Королева	106	200
Игорь Николаев	104	300
Крис Кельми	101	0
Игорь Петров	106	350
Дима Маликов	104	275
Маша Сидорова	108	300
Максим Галкин	108	400

11 строк выбрано.

Хорошо видно, что 4-я и 7-я строка, теперь получили вместо **NULL** значение 0! Хотя это и нарушает принцип тройственной логики, но, тем не менее, избавляет от лишних хлопот. Вот так можно легко и просто раз и навсегда покончить с **NULL**-ами! :)

Еще одна довольно интересная функция которая может пригодиться вам в дальнейшем. Функция **UID**. Синтаксис прост:

UID

Данная функция, возвращает целое число однозначно, идентифицирующая текущего пользователя базы данных. При этом у нее нет аргументов.

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
DBMS_OUTPUT.enable;
-- Function UID() --
DBMS_OUTPUT.put_line( TO_CHAR(UID));
```

```
END;  
/
```

После запуска получаем:

```
SQL> SET SERVEROUTPUT ON  
SQL>  
SQL> BEGIN  
2  
3     DBMS_OUTPUT.enable;  
4     -- Function UID() --  
5     DBMS_OUTPUT.put_line( TO_CHAR(UID));  
6  
7 END;  
8 /  
61
```

Процедура PL/SQL успешно завершена.

При этом пользователь у нас **MILLER**, как вы помните. В чем легко убедиться, введя в **SQL*Plus** команду **SHOW USER**:

```
SQL> SHOW USER  
USER имеет значение "MILLER"
```

Кстати именно для этого есть функция:

USER

Возвращает значение типа **VARCHAR2** содержащее имя текущего пользователя **Oracle**. Аргументов не имеет.

Пример:

```
SET SERVEROUTPUT ON  
  
BEGIN  
  
DBMS_OUTPUT.enable;  
-- Function UID() --  
DBMS_OUTPUT.put_line( TO_CHAR(UID));  
  
-- Function USER() --  
DBMS_OUTPUT.put_line( 'User '||USER||' Uid '||TO_CHAR(UID));  
  
END;  
/
```

Получаем:

```
SQL> BEGIN
```

```

2
3     DBMS_OUTPUT.enable;
4     -- Function  UID() --
5     DBMS_OUTPUT.put_line( TO_CHAR(UID));
6
7     -- Function  USER() --
8     DBMS_OUTPUT.put_line( 'User '||USER||' Uid '||TO_CHAR(UID));
9
10  END;
11  /
61
User MILLER Uid 61

```

Процедура PL/SQL успешно завершена.

Вот так например, легко сделать что-то вроде аудита! Но это уже ваша фантазия! Еще одна интересная функция:

USERENV(параметр)

Возвращает значение типа **VARCHAR2** содержащее сведения о текущем сеансе с учетом "параметр". Значения "параметр" содержит следующая таблица:

Значение "параметр"	Поведение функции USERENV
'OSDBA'	Если в текущем сеансе разрешена роль OSDBA то возвращается 'TRUE' в противном случае возвращается 'FALSE'. При этом не забывайте, что возвращается VARCHAR2, а не BOOLEAN!
'LANGUAGE'	Возвращает язык и территорию, которые используются в этом сеансе, а также набор символов базы данных. Это параметр средства NLS. Возвращаемое значение имеет вид: язык_территория.набор_символов
'TERMINAL'	Возвращает идентификатор терминала текущего соединения, зависящий от вида, операционной системы. Для распределенных SQL - операторов возвращается идентификатор локального соединения.
'SESSIONID'	Возвращает идентификатор соединения способного выполнить аудит, если параметр инициализации AUDIT_TRAIL установлен в TRUE.
'ENTRYID'	Возвращает идентификатор доступного входа способного выполнить аудит, если параметр инициализации AUDIT_TRAIL установлен в TRUE.
'LANG'	Возвращает сокращенное название языка, принятого в ISO.

Давайте попробуем это в действии:

SET SERVEROUTPUT ON

BEGIN

```

DBMS_OUTPUT.enable;
-- Function  USER() --
DBMS_OUTPUT.put_line( 'User '||USER||' Uid '||TO_CHAR(UID));

```

```
-- Function USERENV() --
DBMS_OUTPUT.put_line( USERENV('LANGUAGE'));

-- Function USERENV() --
DBMS_OUTPUT.put_line( USERENV('TERMINAL'));

-- Function USERENV() --
DBMS_OUTPUT.put_line( USERENV('SESSIONID'));

-- Function USERENV() --
DBMS_OUTPUT.put_line( USERENV('ENTRYID'));

-- Function USERENV() --
DBMS_OUTPUT.put_line( USERENV('LANG'));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> BEGIN
  2
  3     DBMS_OUTPUT.enable;
  4     -- Function USER() --
  5     DBMS_OUTPUT.put_line( 'User '||USER||' Uid '||TO_CHAR(UID));
  6
  7     -- Function USERENV() --
  8     DBMS_OUTPUT.put_line( USERENV('LANGUAGE'));
  9
 10     -- Function USERENV() --
 11     DBMS_OUTPUT.put_line( USERENV('TERMINAL'));
 12
 13     -- Function USERENV() --
 14     DBMS_OUTPUT.put_line( USERENV('SESSIONID'));
 15
 16     -- Function USERENV() --
 17     DBMS_OUTPUT.put_line( USERENV('ENTRYID'));
 18
 19     -- Function USERENV() --
 20     DBMS_OUTPUT.put_line( USERENV('LANG'));
 21
 22 END;
 23 /
User MILLER Uid 61
RUSSIAN_CIS.CL8MSWIN1251
ORAHOME
1592
0
RU
```

Процедура PL/SQL успешно завершена.

Вот так, действует функция **USERENV**. Вот собственно, пока все со встроенными функциями **PL/SQL**. Могу сразу сказать, что мы с вами рассмотрели далеко не все функции, но пока это достаточный минимум для того, чтобы приступить к изучению дальнейшего материала. Можете пока поработать со всем этим, и если есть вопросы, то задавайте!

Шаг 74 - Архитектура БД Oracle

Думаю пришло время, поговорить о том, как вообще устроена СУБД **Oracle**. Когда речь идет о БД **Oracle**, то обычно имеется в виду система управления БД. Но, для профессиональных пользователей БД **Oracle** необходимо понимание разницы между собственно Базой Данных и экземпляром. Иногда эти два понятия вводят в заблуждение администраторов БД других фирм разработчиков. Высокий уровень сервиса гибкость и производительность, которую БД **Oracle** предоставляет клиентам обеспечивается сложным комплексом структур памяти и процессов операционной системы. Все эти понятия в совокупности называются "экземпляром" (**instance**). Любая БД **Oracle** имеет связанный с ней экземпляр. Тот самый, который мы с вами получили при инсталляции. Сама по себе организация экземпляра позволяет СУБД обслуживать множество типов транзакций, иницируемых одновременно большим количеством пользователей, в то же время обеспечивая высокую производительность, целостность данных и безопасность. При работе БД **Oracle** одновременно присутствует множество процессов, выполняющих специфические задачи, в рамках СУБД. Каждый процесс имеет отдельный блок памяти, в котором сохраняются локальные переменные, стек адресов и другая информация. Все эти процессы используют так называемую - "разделяемую область памяти". В ней хранятся данные общего пользования. Доступ к этой памяти, как для записи, так и для чтения, могут получить одновременно различные процессы и программы. Этот блок памяти вообще называется - "Глобальной Системной Областью" или ГСО. По-английски в документации это звучит как **System Global Area - SGA**. Еще можно уточнить, так как ГСО находится в разделяемом сегменте памяти, ее еще часто называют "Разделяемой Глобальной Областью" **Shared Global Area**. Вообще просто следует запомнить, что это за область и для чего она нужна. А, как ее назвать это уже на ваше усмотрение. :) Кстати, если провести аналогию между процессами системы и, например, организмом человека, то **SGA** это мозг, а процессы это скажем руки, ноги, уши и т.д. Но всеми органами (процессами) управляет единый центр мозг (**SGA**), так как она координирует все процессы обработки информации происходящие в системе. Вот собственно, так и взаимодействует этот сложный "организм".

БД **Oracle** производит свое формирование следующим образом. Весь процесс делится примерно на три этапа:

1. Формирование экземпляра **Oracle** (предустановочная стадия)
2. Установка БД экземпляром (установочная стадия)
3. Открытие БД (стадия открытия)

Далее по порядку. Экземпляр **Oracle** формируется на предустановочной стадии запуска системы. На этой стадии считывается файл параметров **init.ora**, запускаются фоновые процессы, и инициализируется ГСО (**SGA**). При этом имя экземпляра устанавливается в соответствии со значением указанным в **init.ora**. Следующая стадия - установочная. Значения параметров контрольного файла **init.ora** определяют параметры БД, устанавливаемой экземпляром. На данном этапе доступ к контрольному файлу открыт и возможна модификация данных, которые в нем хранятся. И на последней стадии собственно открывается сама База данных. Экземпляр получает исключительный доступ к файлам БД, имена которых, хранятся в контрольном файле и через него они становятся доступны пользователям БД. Фактически, если смотреть на открытие как состояние, то это скорее нормальное рабочее состояние БД. Заметим сразу, что до тех пор, пока БД не будет открыта, к ней может получить доступ только администратор БД и только через утилиту **Server Manager**. (В версии **Oracle 9** этой утилиты нет. Её функции полностью переданы **SQL*Plus**. Так же, схема **INTERNAL** в **Oracle 9** отсутствует.) Наверное, вы немного подустали уже от сухой теории, да и тот парень на галерке, по-моему, уже посапывает! Давайте разбудим его! :) Посмотрим, как можно управлять БД, используя утилиту **Server Manager**.

Войдите в каталог **..\Oracle\ora81\Bin** вашей учебной БД имеющей **SID** - **proba**. Затем создайте в этом каталоге **bat** файл с именем **serverora.bat** и таким содержимым:

```
set nls_lang=russian_cis.ru8pc866  
SVRMGRL.EXE
```

Сразу оговорюсь, делайте это по возможности на самой машине, где установлен сервер **Oracle** и его экземпляр, а не с клиентской машины, так как может возникнуть ряд нюансов, на которых мы пока не будем акцентироваться. После создания файла запустите его на исполнение, вы должны увидеть примерно следующее:

```
Oracle Server Manager Release 3.1.5.0.0 - Production
```

```
Copyright (c) Oracle Corporation 1994, 1995, 1997. Все права защищены.
```

```
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production  
With the Partitioning and Java options  
PL/SQL Release 8.1.5.0.0 - Production
```

```
SRVMGR>_
```

Введите в ответ на приглашение - **internal/oracle@proba** или, если вы все делаете на самом сервере - **internal/oracle**.

Получите в ответ:

```
SRVMGR> internal/oracle  
Связь установлена.  
SRVMGR>_
```

Затем введите на приглашение:

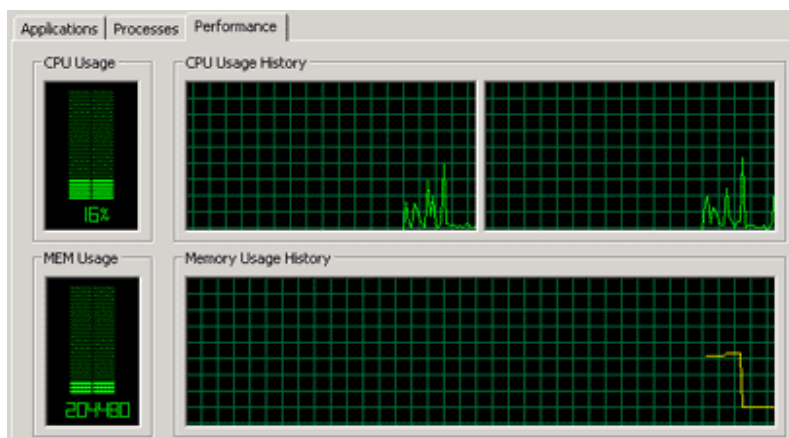
```
SRVMGR> SHUTDOWN NORMAL
```

Через некоторое время увидим следующее:

```
База данных закрыта  
База данных демонтирована.  
Экземпляр ORACLE закрыт.
```

```
SRVMGR>_
```

Если сначала вы запустите диспетчер задач и перейдете на его закладку **Performance**, то будете при этом наблюдать следующую картину:



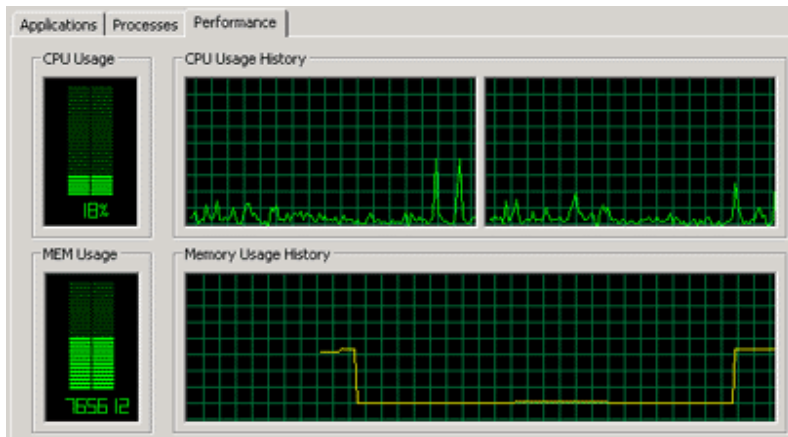
Это картинка с моего сервера, у меня два процессора и **Win2000 Server**. Как видно, большая часть ОЗУ очистилась, так как произошло закрытие сервера **Oracle**. Обратите внимание на последовательность сообщений сначала происходит закрытие БД, а затем БД демонтируется. (подобные сообщения выдавал файл-сервер **Novell**, только там монтировались и демонтировались тома) и в конце экземпляр БД закрывается. Кстати, если еще остановить сам сервис, занимаемой памяти станет еще меньше. Вот так с помощью **Server Manager**, можно остановить ваш экземпляр БД. Далее, давайте его запустим, вводим:

```
SRVMGR> STURTUP FORCE
```

Получаем после нажатия Enter

```
Экземпляр ORACLE запущен.  
Всего байтов System Global Area          547758028  
Fixed Size                               65484 байтов  
Variable Size                           143876096 байтов  
Database Buffers                        403742720 байтов  
Redo Buffers                             73728 байтов  
База данных смонтирована.  
База данных открыта.
```

Теперь смотрите внимательно, первая строка сообщает, что экземпляр запущен! Далее пять строк дают служебную информацию и БД монтируется и открывается. Вот так ваш сервер **Oracle** стартует для того, чтобы начать свою работу.



Теперь хорошо видно, что объем занимаемой памяти увеличился! :) К утилите **Server Manager** мы еще вернемся, так как она выполняет еще ряд полезных функций, а пока еще раз все внимательно разберите. Да и в конце не забудьте дать команду **exit**:

```
SRVMGR> exit
```

Получите:

```
Server Manager закончил работу.
```

И в заключение, экземпляр в котором не установлена БД, называется не занятым (**idle**). Он занимает память но, не выполняет никакой работы. Это как раз примерно то, о чем я говорил выше. Экземпляр остановлен, а сервис еще нет. Хотя в вашем случае сама БД уже есть, просто она остановлена. Надеюсь, теперь понятно, что такое экземпляр, а что такое сервер БД. :)

Шаг 75 - Архитектура БД Oracle ЧАСТЬ II

На рисунке представлена структурная схема экземпляра **Oracle**. Давайте разберем отдельные его компоненты.

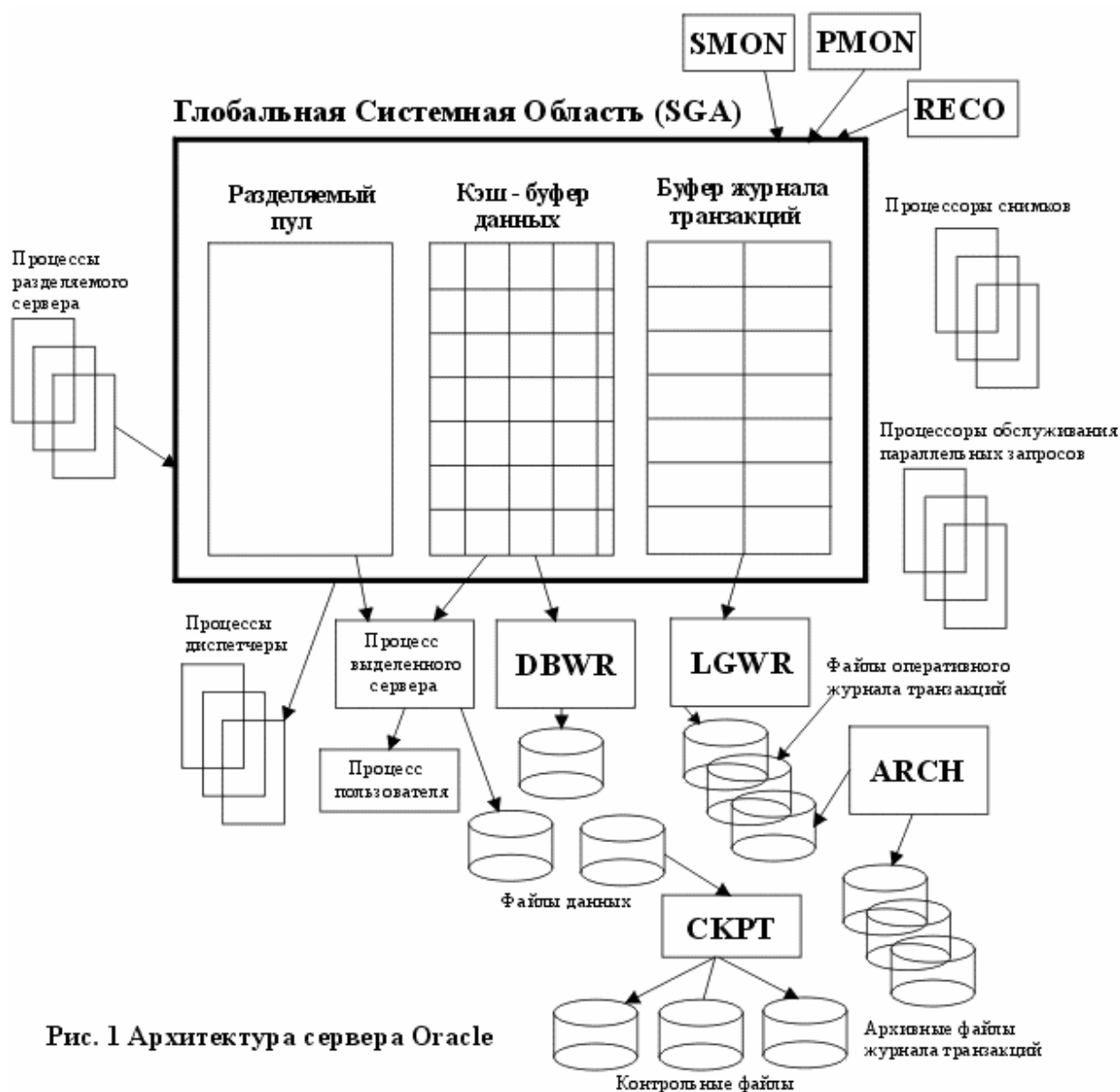


Рис. 1 Архитектура сервера Oracle

Для настройки экземпляра **Oracle**, в соответствии с назначением и особенностями использования конкретной схемы, существует множество параметров и методик. Настройка экземпляра это, как правило, итерационный процесс. Вы начинаете с априори (слова то, какие умные) приемлемых значений параметров, а затем по мере накопления опыта эксплуатации (после того, например как он у вас завалится раза три, четыре, потому, что ручки шаловливые) уточняете их значения. Настройка включает внесение необходимых изменений в параметры, хранящиеся в файле **init.ora**.

В этом файле имеется большое число параметров, многие из которых не документированы, но это иногда не мешает, особо ретивым, поэкспериментировать с ними! :) Но я бы, советовал воспользоваться системным представлением **x\$kspgi**. Давайте дадим такой запрос:

```
SELECT k.kspginm, k.kspgdesc FROM x$kspgi k
/
```

Например на моей БД я получил в ответ следующее:

```
SQL> SELECT k.kspginm, k.kspgdesc FROM x$kspgi k
2 /
```

KSPGINM	KSPGDESC
-----	-----
_trace_files_public	Create publicly accessible trace files
_spin_count	Amount to spin waiting for a latch
_latch_miss_stat_sid	Sid of process for which to collect latch stats
_max_sleep_holding_latch	max time to sleep while holding a latch
_max_exponential_sleep	max sleep during exponential backoff
_latch_wait_posting	post sleeping processes when free latch
_latch_recovery_alignment	align latch recovery structures
_use_vector_post	use vector post
processes	user processes
sessions	user and system sessions
timed_statistics	maintain internal timing statistics
timed_os_statistics	internal os statistic gathering interval in seconds
resource_limit	master switch for resource limit
license_max_sessions	maximum number of non-system user sessions allowed
license_sessions_warning	warning level for number of non-system user sessions
_session_idle_bit_latches	one latch per session or a latch per group of sessions
_single_process	run without detached processes
cpu_count	number of cpu's for this instance
_number_cached_attributes	maximum number of cached attributes per instance
instance_groups	list of instance group names
.	
.	
.	
KSPGINM	KSPGDESC
-----	-----
_domain_index_dml_batch_size	maximum number of rows for one call to domain index dml routines
aq_tm_processes	number of AQ Time Managers to start
hs_autoregister	enable automatic server DD updates in HS agent self-registration

443 строк выбрано.

Впечатляет количество параметров? Я думаю да! Поле **KSPGDESC**, содержит краткое описание каждого параметра. Обратите внимание, что недокументированные параметры, как правило начинаются с символа "_" ! Будьте очень осторожны при изменении этих параметров, если не представляете, что они делают. Можете кстати вернуться к [шагу 20](#) там, последний запрос показывал содержимое файла **init.ora** при обращении к системному представлению **v\$parameter**, примерно что-то вроде:

```
SELECT a.name, a.value
FROM v$parameter a
ORDER BY a.name
/
```

```
.
```

NAME	VALUE
db_block_lru_latches	1
db_block_max_dirty_target	49285
db_block_size	8192
db_domain	com
db_file_direct_io_count	64
db_file_multiblock_read_count	8
db_file_name_convert	
db_files	1024
dblink_encrypt_login	FALSE
db_name	proba
dbwr_io_slaves	0
db_writer_processes	1
disk_asynch_io	TRUE
distributed_transactions	10
dml_locks	264

```
.
```

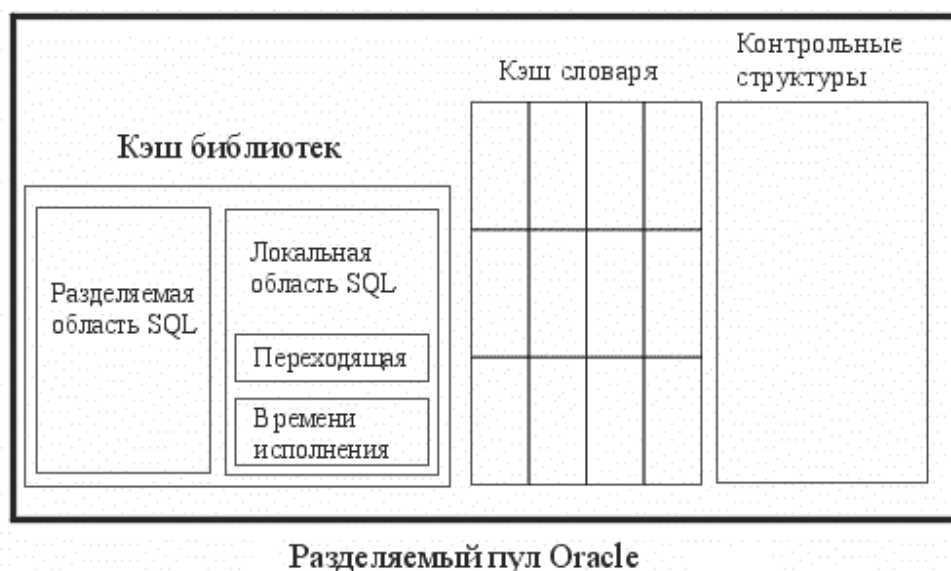
Большинство настроек экземпляра связано с компонентами в **SGA**. Однако кроме них, существуют и некоторые опции настройки, касающиеся фоновых процессов и процессов пользователей. Далее мы будем возвращаться к нашему рисунку еще несколько раз, по этому советую его просто распечатать и держать под рукой! :)

Шаг 76 - Архитектура БД Oracle ЧАСТЬ III

Итак, **SGA**. В **SGA (System Global Area)** - хранятся структуры памяти, необходимые для манипулирования данными, а так же анализа предложений **SQL** и кэширования транзакций. Сама по себе **System Global Area** - разделяемая, то есть к ней одновременно имеют доступ множество процессов, которые в свою очередь могут считывать или модифицировать, содержащиеся в ней данные. Все операции БД, так или иначе, используют информацию, находящуюся в **System Global Area**. **SGA** - выделяется сразу же после создания экземпляра, еще на предустановочной стадии. Освобождается эта область только после того как экземпляр будет закрыт. **SGA** - в свою очередь состоит из следующих компонентов:

1. Разделяемый пул (Shared Pool)
2. Кэш-буфер данных (Database Buffer Cache)
3. Буфер журнала транзакций (Redo Log Buffer)
4. Структуры сервера многозадачной среды (Multi-Threaded Server - MTS)

По порядку - **Shared Pool**.



Разделяемый пул (**Shared Pool**) содержит кэш библиотек, кэш словаря и управляющие структуры сервера (такие как набор символов БД). Кэш библиотеки хранит текст, форматы лексического анализатора и план выполнения предложений **SQL**, которые адресуются БД. Кроме того, здесь находятся заголовки **PL/SQL** пакетов и процедур, выполнявшихся ранее. Так же кэш словаря хранит строки словаря данных, которые были использованы для анализа предложений **SQL**. Сам сервер **Oracle** использует для повышения производительности и скорости выполнения операций и операторов **SQL**, кэш библиотеки. Таким образом, когда передается очередное **SQL** - выражение, сервер, в первую очередь, просматривает кэш в поисках подобного выражения, которое уже передавалось ранее. Если оно найдено, то используется соответствующее дерево лексического анализа, что приводит к повышению скорости выполнения операторов **SQL**. Но, есть одно существенное замечание - выражения **SQL**, должны быть полностью идентичны, вплоть до регистра символов операторов и переменных. Так как **Oracle** сравнивает тексты выражений, применяя алгоритм хеширования, который чувствителен к регистру символов текста. Так же кэш библиотек содержит разделяемые и локальные области **SQL**. Разделяемая

область включает в себя дерево лексического анализа и план выполнения **SQL** - оператора, а локальная область информацию, зависящую от текущего сеанса работы. Локальная область формируется для каждой иницируемой транзакции и освобождается после того, как закрывается соответствующий курсор. Этот процесс определяет параметр **open_cursor** в файле **init.ora**. В соответствии с этим, **Oracle** может повторно использовать информацию, общую для всех выражений **SQL**, а информация, специфическая для данного сеанса, может быть выбрана из локальной области **Shared Pool**. Локальная область **SQL** в свою очередь делится, на переходящую (**persistent**) и область времени выполнения (**runtime**). Переходящая область содержит информацию, которая сохраняет свое значение и может быть использована несколькими выражениями **SQL**. А область времени выполнения - только информацию, для выражения выполняемого в текущий момент. Кэш словаря содержит информацию из словаря данных, необходимую СУБД для лексического анализа **SQL** - выражения. В ней находятся данные, касающиеся сегментирования, привилегий доступа и размеров свободной памяти. Собственно сам размер выделяемого пула определяется параметром **SHARED_POOL_SIZE** в файле **init.ora**. Это кстати очень интересный параметр и когда мы, наконец, доберемся до раздела создания экземпляра, вы с ним познакомитесь более тесно. Если его размер не достаточен, то может наступить момент когда для загрузки очередного объекта не найдется непрерывной области памяти подходящего объема. Нужно просто определять его размер с "запасом"! :)

Шаг 77 - Архитектура БД Oracle ЧАСТЬ IV

Продолжаем немного сухую теоретическую часть. Что поделать, знать это нужно все равно. Итак, следующий пункт: Кэш-буфер данных (**Database Buffer Cache**).

Производительность всей системы в целом зависит от функционирования кэш-буфера данных. Самое интересное, что он состоит из блоков памяти того же размера, что и блоки **Oracle**. С ними мы познакомились в [шаге 22](#). Все данные первым делом загружаются в кэш-буфер. В них же выполняется и любое обновление данных, по этому очень важно, правильно установить размер буфера. Сам **Oracle** переносит данные на диск (используя при этом подкачку **swap** - данных) в соответствии с порядком их размещения в списке **LRU (Least Recently Used** - дословно, наиболее давно использовавшиеся). Этот список отслеживает обращение к блокам данных и учитывает частоту обращений к ним. Про этот механизм мы уже с вами говорили. Когда выполняется обращение к блоку данных, хранящемуся в кэш-буфере, он помещается в тот конец списка, который называется **MRU (Most Recently Used** - только что использованные). При этом, если серверу требуется место в кэш-буфере для загрузки нового блока с диска, он обращается к списку **LRU** и решает какой из блоков перенести на диск, для того чтобы освободить место для нового блока. Блоки наиболее удаленные в списке от **MRU**, самые вероятные кандидаты на удаление из кэш-буфера. Таким образом, дольше всего остаются в кэш-буфере те блоки, обращение к которым производится наиболее часто.

Модифицированные блоки называются грязными (**dirty**) и помещаются в соответствующий **dirty**-список. В этом списке отслеживаются все модификации блоков данных, выполненные за время их нахождения в кэш-буфере и не зафиксированные. Когда **Oracle** получает запрос на изменение данных, соответствующие изменения выполняются в блоках кэш-буфера, а сведения об измененных блоках заносятся в **dirty**-список. При этом одновременно данные о выполненных операциях заносятся в журнал транзакций. В дальнейшем при обращении к блокам данных, попавшим в **dirty**-список, будут, считываться уже модифицированные значения, хотя сами данные при этом могут быть еще не зафиксированы. **Oracle** использует "отложенную" многоблочную процедуру записи на диск. Отсюда следует, что "отложенная" означает - обновление данных выполненное сервером не фиксируется немедленно, а сервер ждет пока не возникнут следующие события:

- внесены изменения в некоторое заранее установленное число блоков данных, при этом требуется освободить место в кэш-буфере для загрузки новых блоков данных.
- обнаружена контрольная точка.

В любом из этих случаев группа модифицированных блоков данных переписывается на диск из кэш-буфера в файлы. Для определения правильного размера кэш-буфера, требуется определенный опыт. Размер кэш-буфера определяется двумя параметрами настройки **DB_BLOCK_SIZE** и **DB_BLOCK_BUFFERS** в файле **init.ora**. Общий объем кэш-буфера(в байтах) определяется, как произведение этих двух параметров **DB_BLOCK_SIZE*DB_BLOCK_BUFFERS**. Вот так устроен и работает механизм под названием кэш-буфер данных.

Шаг 78 - Архитектура БД Oracle ЧАСТЬ V

Теоретическая часть продолжается. :) Разберем одну из концептуальных частей сервера **Oracle**, следующий пункт: Буфер журнала транзакций (**Redo Log Buffer**).

Он представляет собой циклический буфер. Как только буфер заполнится и поступает новая информация, то происходит перезапись в файлы журналов транзакций. Данные о транзакциях хранятся до тех пор, пока не будут переписаны в файл оперативного журнала транзакций. Размер буфера журнала транзакций задается, параметром **LOG_BUFFER**, файла **init.ora**. Значение параметра указывает размер буфера в байтах. Если это значение невелико то фоновый процесс **LGWR** будет часто переписывать информацию из буфера в файл и тем самым, несколько замедлять работу системы. В принципе тоже будет происходить и при большой интенсивности обращений к БД. Давайте дадим вот такой запрос:

```
SELECT a.VALUE FROM V$SYSSTAT a
WHERE NAME = 'redo log space request'
/
```

Получаем:

```
SQL> SELECT a.VALUE FROM V$SYSSTAT a
2  WHERE NAME = 'redo log space request'
3  /
```

```
VALUE
-----
0
```

С помощью системного представления **V\$SYSSTAT** можно оперативно контролировать процесс записи в журнал. Данный запрос показывает как долго пользовательские процессы, находились в состоянии ожидания при обращении к буферу журнала транзакций. В моем случае значение 0, говорит лишь о том, что моя домашняя БД, пока ничем особым не занималась, но может ей еще повезет! :) Для того, чтобы гарантировать последовательный характер записи в буфер журнала сервер **Oracle** управляет доступом к нему при помощи "защелок" (**latch**). Такая защелка представляет собой блокировку процессом **Oracle** некоторой структуры в памяти - аналогично, блокировке файла или строки таблицы. Процесс блокирует посторонние обращения к памяти, выделенной для буфера журнала транзакций. Таким образом, если один процесс наложил защелку на буфер, к нему нельзя обратиться, пока ее не снимут. **Oracle** ограничивает количество транзакций, данные о которых заносятся в журнал одновременно параметром **LOG_SMALL_ENTRY_MAX_SIZE**. Значение параметра в байтах. Значение, принимаемое системой по умолчанию может меняться в зависимости от используемой **OS**. Используя защелки копий журнала транзакций, несколько процессов могут одновременно записывать информацию в буфер журнала транзакций. Так же, можно добавить, что мониторинг работы с буфером журнала транзакций осуществляется с помощью динамического представления **V\$LATCH**.

Теперь настало время поговорить, о так называемых, фоновых процессах **Oracle**. В процессе работы сервер, одновременно работает с тысячами или даже более записями в таблицах, БД. Проводит выборки данных, по запросам пользователей, изменяет и удаляет данные. Для того, чтобы весь этот комплекс работал стабильно и без сбоев, вся работа делится между несколькими вспомогательными программами, которые действуют независимо одна от другой. В совокупности эти программы называются фоновыми процессами сервера **Oracle**. Сразу заметим,

что для платформы **Windows NT** фоновые процессы реализованы как потоки к сервисам **Oracle**. Это позволяет более эффективно использовать адресное пространство памяти общего пользования. Итак к фоновым процессам относятся следующие:

1. SMON - PMON
2. DBWR
3. LGWR
4. Dnnn
5. ARCH
6. CKPT
7. RECO
8. SPNn
9. LCKn
10. Pnnn
11. Snnn

Так же есть процессы **User** и **Server** выполняющие обработку транзакций, конечного пользователя БД. Рассмотрим их, более подробно.

Процессы **SMON**, **PMON** это процессы мониторов БД. **PMON** - (**Process Monitor**) это процесс, который осуществляет контроль за состоянием подключений к БД. Если по какой-либо причине пользователь потерял контакт с БД, не завершив работы корректно **PMON** выполнит автоматическую "уборку" (нечто вроде сборщика мусора в языках программирования). Эта "уборка" предусматривает удаление сеанса, закрепленного за прекратившимся процессом, блокировок, которые были им установлены и непринятых транзакций. Так же он освободит ресурсы **SGA**. Так же **PMON** следит за процессами сервера и диспетчера и автоматически перезапускает их в случае останова. **SMON** - чуть более скромн, но так же немаловажен. После запуска БД, он как это не грандиозно звучит, выполняет автоматическое восстановление экземпляра. В случае если у вас отключили свет и сервер успел "слопать" УПС! У меня такие случаи были и пока я выходил сухим из воды! (постучать по дереву). :) Кроме того, он следит за сегментами БД, фиксирует освобождение пространства во временных сегментах и автоматически объединяет их. Так же заметим, что объединение блоков в табличных пространствах, производится при условии, если параметр **pctincrease** равен 0. Этот параметр, используется при создании табличных пространств. Процессы **SMON**, **PMON** должны быть запущены при старте БД, иначе она не будет функционировать. Вот так работают **SMON** и **PMON**.

Шаг 79 - Архитектура БД Oracle ЧАСТЬ VI

Продолжаем разбирать фоновые процессы. Следующий на очереди, процесс **DBWR**.

DBWR - это (**DataBase Writer**) один из, пожалуй, ключевых процессов отвечающих за перенос данных. Он отвечает за перенос обновленных блоков и производит перезапись в следующих случаях:

1. Обнаружена контрольная точка.
2. Количество элементов в **dirty** - списке достигло заданной величины - половина значения параметра **DB_BLOCK_WRITE_BATCH** из файла **init.ora**.
3. Количество использованных буферов достигло величины, заданной параметром **DB_BLOCK_MAX_SCAN** из файла **init.ora**.
4. Истек заданный для процесса **DBWR** интервал времени (3 с).

Итак, блоки, попавшие в **dirty** - список, переносятся в файлы данных. Вместо того, чтобы записывать каждый блок на диск сразу же после внесения изменений процесс **DBWR** ждет, пока не будет выполнено одно из вышеуказанных условий, затем просматривает **dirty** - список и все отмеченные в нем блоки переписываются в файлы данных на диске. Такой метод организации групповой перезаписи модифицированных блоков БД позволяет уменьшить влияние скоростных характеристик жестких дисков на производительность системы в целом. Процесс **DBWR** имеет множество методов настройки, в зависимости от характеристик и размеров БД. Например, если один процесс не успевает за обновлением данных, то их можно запустить несколько для этого поменяйте параметр **DB_WRITERS** в файле **init.ora** с 1, на цифру количества дисков системы БД. Так же можно изменить параметр **DB_BLOCK_CHECKPOINT_BATCH**. Он определяет максимальное количество блоков, которые процесс **DBWR** записывает для каждой контрольной точки (что, это такое мы еще разберем). Самое главное при настройке **DBWR** не терять чувство реальности! :) Так как некоторые параметры могут наоборот снизить производительность системы.

Далее, процесс **LGWR** - (**Log Writer**). Это четвертый и последний процесс из числа тех, которые обязательны для нормального функционирования БД **Oracle**. Так, что же он делает? Процесс **LGWR**, производит перезапись информации из буфера журнала транзакций, которая находится в ГСО (**SGA**), в файлы оперативного журнала. А, теперь посмотрим при каких условиях это происходит:

1. Транзакция принимается. (Истекает время ожидания для процесса **LGWR**)
2. Буфер журнала транзакций заполняется на треть.
3. Процесс **DBWR**(!) завершает перезапись данных из кэш буфера после обнаружения контрольной точки.

Так же **LGWR** обрабатывает завершение транзакций для нескольких пользователей одновременно, когда один или несколько пользователей запрашивают завершение в то время, когда **LGWR** еще не произвел перезапись буфера. Так же очень важный момент состоит в том, что **Oracle** не считает транзакцию выполненной до тех пор, пока процесс **LGWR** не перезапишет данные о ней из буфера журнала транзакций в файл. И само сообщение о завершении транзакции передается процессу сервера не после изменения данных в файле данных, а после успешного завершения записи в файл журнала транзакций. В файле **init.ora**, имеется два параметра, которые влияют на работу процесса **LGWR**. Это параметры **LOG_CHECKPOINT_INTERVAL**, **LOG_CHECKPOINT_TIMEOUT**. Давайте остановимся на них чуть подробнее. Одной из побочных для процесса **LGWR** задач, является обработка

контрольных точек. **LGWR** - работает с ними только при условии, что процесс **СКРТ** активирован, а работать с контрольными точками это его прямая обязанность. Так вот эти два параметра непосредственно и определяют, интервалы следования контрольных точек в самой БД. При активации параметра **LOG_CHECKPOINT_INTERVAL**, контрольная точка генерируется при условии, что указанное количество блоков операционной системы (а, не блоков БД!) записывается в журнал транзакций. А вот при активации параметра **LOG_CHECKPOINT_TIMEOUT** контрольная точка генерируется при условии, что истек период ожидания времени в секундах. Теперь ясно видно, что обработка контрольных точек при функционировании БД довольно не простая задача. Эти параметры нужно использовать весьма осторожно! Если для настройки используется параметр **LOG_CHECKPOINT_INTERVAL**, то указанное количество блоков ОС должно соотноситься с размером группы журнала транзакций! Когда эта группа заполняется - генерируются контрольная точка. Так же можно использовать параметр **LOG_CHECKPOINT_TO_ALERT** из файла **init.ora**, если он активирован то, после генерации каждой контрольной точки будет проставлена соответствующая пометка, в файле регистрации **alert.log**, что может оказаться полезным при давнейшем процессе анализа генерации контрольных точек БД. Надеюсь, если кому-либо понятно хотя бы половина моего изложения, я буду весьма рад! :)

Шаг 80 - Архитектура БД Oracle ЧАСТЬ VII

Продолжаем разбирать фоновые процессы БД. Фоновый процесс **Dnnn**.

Как мы уже отмечали, если нет то, остановимся на этом. Все процессы сервера, могут закрепляться за определенными пользовательскими процессами, либо использоваться несколькими процессами пользователя. В последнем случае они называются "разделяемые" (**shared**) процессами или серверами. Для работы с применением разделяемого сервера необходима установка **Multi Threaded Server (MTS)**. При использовании разделяемых процессов, в системе должен существовать как минимум один процесс диспетчер. Процесс - диспетчер передает запросы пользователей в очередь ГСО и возвращает ответы сервера, соответствующему процессу пользователя.

Фоновый процесс **Dnnn**, где как вы поняли **nnn** - это число, определяется по количеству, параметром **MTS_DISPATCHERS**. Он указывает протокол, который будет использован диспетчером и количество процессов.

Например:

```
MTS_DISPATCHERS = "tcp, 4"  
MTS_DISPATCHERS = "spx, 2"
```

Либо вот так:

```
MTS_DISPATCHERS = ("tcp, 4" , "spx, 2")
```

Следующий процесс **ARCH** - (**Archiver** - "архиватор"). Отвечает за копирование полностью заполненного оперативного файла журнала транзакций, в архивные файлы журнала транзакций. Для того, чтобы запустить это процесс нужно установить, как вы уже догадались параметр **ARCHIVE_LOG_START** в файле **init.ora** в значение **TRUE**. И тогда указанный процесс будет выполнять архивацию, журналов транзакций.

Следующий процесс **CKPT** - это как раз тот процесс, о котором мы упоминали в [прошлый раз](#). Он непосредственно отвечает за обработку контрольных точек. Обычно такая обработка, обновление файлов данных и заголовка контрольных файлов - выполняется процессом **LGWR**. А вот **CKPT** необходим для того, чтобы снизить нагрузку на **LGWR**. Вот собственно так он и работает.

Следующий процесс **RECO** - (**recovery**). Отвечает за восстановление незавершенных транзакций. Он запускается автоматически, если система сконфигурирована для распределенных транзакций. За это отвечает параметр **DISTRIBUTED_TRANSACTION** в файле **init.ora**. Когда возникает подозрительная транзакция, процесс **RECO** выполняет свою работу без вмешательства администратора БД. В общем, самостоятельный парень! :)

Следующий процесс **SNPn** - выполняет автоматическое обновление снимков БД (**snapshot**). Так же запускает процедуры в соответствии с расписанием, зафиксированным в пакете **DBMS_JOB**. Параметр **JOB_QUEUE_PROCESS** в файле **init.ora** задает количество запускаемых процессов **SNPn**, а параметр **JOB_QUEUE_INTERVAL** длительность в течении, которой процесс "засыпает" прежде чем выполнить задание.

Следующий процесс **LCKn** - в среде производящей параллельное обслуживание нескольких экземпляров БД на **LCKn** возлагается ответственность за координацию блокировок устанавливаемых разными экземплярами БД. Если в системе нет параллельного обслуживания, то запуск **LCKn** не нужен. Да и ответственности меньше.

Последнее, процесс **Pnnn** - это процесс параллельных запросов. Сервер **Oracle** запускает и останавливает процессы **Pnnn** в зависимости от активности работы БД и настройки опций параллельных запросов. Эти процессы принимают участие в формировании компонентов БД. Количество запущенных процессов, определяется параметрами **PARALLEL_MIN_SERVERS** и **PARALLEL_MAX_SERVERS** соответственно. Вот таким образом мы немного разобрались с фоновыми процессами БД **Oracle**. Я думаю, что данный материал будет для вас полезен и наведет на дальнейшее более глубокое изучение сервера **Oracle**. :)

Шаг 81 - Архитектура. Серверы, пользователи и т. д.

Если сказать по большому счету, мы с вами уже довольно далеко продвинулись, в понимании что же такое сервер БД. Тем не менее в части теории, еще многое нужно научиться понимать и правильно применять на практике.

Давайте, в нашей теоретической части рассмотрим еще кое, что. Я к тому это веду, что бы все это не показалось для вас слишком скучным и неинтересным, уверяю вас это далеко не так! Так, что продолжим!

ПРОЦЕССЫ ПОЛЬЗОВАТЕЛЯ И СЕРВЕРА.

Приложения и клиентские части связываются с сервером **Oracle** посредством "процессов пользователя". Каждый процесс пользователя имеет подключение к процессу сервера (смотри рисунок [шага 75](#)), который может быть либо жестко связан с одним процессом пользователя, либо распределяться между многими. Процесс сервера, анализирует и выполняет переданные ему операторы **SQL** и возвращает результат процессу пользователя. Так же процесс сервера считывает блоки данных из файла данных и размещает их в кэш-буфере данных.

Каждому процессу пользователя выделяется область памяти, которая называется "глобальной областью процесса" - ГОП (**Process Global Area - PGA**) содержимое **PGA** зависит от режима подключения процесса пользователя к процессу сервера. Если процесс пользователя имеет выделенный процесс сервера, то в **PGA** размещается информация о текущем сеансе работы пользователя, стек и информация о состоянии курсора. Если же процесс пользователя связан с разделяемым процессом сервера, то информация о текущем сеансе и текущем состоянии курсора хранится в **SGA**. В общем это не очень существенно, хотя некоторое увеличение размера **SGA** все же потребуется. В этом и состоит разница между двумя типами взаимодействия клиента и сервера в БД **Oracle**. Теперь давайте рассмотрим еще один не маловажный аспект.

ТРАНЗАКЦИЯ ИЗНУТРИ.

Начало транзакции это когда пользователь получает соединение с сервером, через драйвер **SQL*Net**. Это подключение может быть выделенным(**dedicated**) как мы, и говорили выше, а может быть разделяемым (**shared**). В первом случае происходит подключение к собственному процессу сервера, а во втором - к уже знакомому нам разделяемому через процесс диспетчер. Сервер хэширует получаемые от пользователя, выражения **SQL** и сравнивает сформированный хэш-код, с хэш-кодом сохраненным в **SGA** ранее. Если в разделяемом пуле найдено точное совпадение выражений **SQL**, то используется сформированное ранее дерево, лексического разбора и план выполнения выражения. А, если нет, то сервер производит разбор выражения.

В случае, если производится запрос данных, полученный набор возвращается пользователю. Если же происходит модификация данных, то здесь все немного сложнее. Например, в течении транзакции должно произойти обновление данных. После того как затребованные блоки данных, будут считаны в кэш буфер, уже там (!) в памяти они модифицируются. Модифицированные блоки данных помечаются как "грязные" - (**dirty**) и помещаются в **dirty** - список. Так же формируется информация, для журнала регистрации транзакций, которая заносится в кэш журнала. Затем в зависимости, от продолжительности текущей транзакции может произойти следующее:

1. Информация о транзакции, которая записывается в кэш журнала, заполняет этот буфер на треть, что вынуждает процесс **LGWR** переписать содержимое буфера в файл.

2. Количество блоков, отмеченное в **dirty** - списке, достигло заданного значения. Это вынуждает процесс **DBWR** - выполнить перезапись модифицированных блоков из кэш - буфера данных в файлы данных. Это в свою очередь заставляет процесс **LGWR** переписать содержимое буфера журнала транзакций в файл.
3. Появилась контрольная точка БД. Это событие запускает вышеописанную процедуру перезаписи модифицированных блоков, данных в файлы и перезаписи, буфера журнала транзакций в соответствующий файл.
4. Количество свободных блоков в кэш - буфере уменьшилось и достигло критической величины. Это так же приведет к перезаписи данных из кэш - буфера в файлы.
5. Возникла невосстановимая ошибка БД. Это заставляет прекратить выполнение транзакции и запустить механизм отката. Соответственно формируется сообщение об ошибке, которое направляется серверу.

Вот таким образом строится работа БД, при выполнении текущих, транзакций. Естественно, для того, чтобы процесс транзакции завершился успешно нужно, чтобы по крайней мере процессы **DBWR** и **LGWR** сработали без ошибок.

Шаг 82 - Простейший мониторинг экземпляра БД

Вот, наконец, и настал новый 2004 год! Всем желаю, всех благ и чтобы у всех Орклоидов и не только, сбылись все мечты и все желания! Я наконец вернулся из небольшой, но весьма насыщенной событиями поездки и думаю продолжить наши с вами занятия БД **Oracle**. :) Кто-нибудь там с галерки может быть, напомним нам, на чем мы остановились!? Тишина ... ну, что ж тогда напомним, что мы с вами остановились, на разборе процесса транзакции. Надеюсь, что всем стало немного понятнее как она работает. И мы с вами в плотную подошли к теме простых приемов мониторинга, нашего с вами экземпляра БД **Oracle**.

Итак, давайте попробуем разобраться. Мониторинг БД, как правило, бывает необходим в силу ряда причин, одной из которых, может быть возникновение разнообразных сбоев и неполадок. Так, например, при работе БД создает файлы трассировки процессов, по которым легко определить основные неполадки при функционировании экземпляра БД. Располагаются эти файлы согласно параметров **USER_DUMP_DEST** и **BACKGROUND_DUMP_DEST** в файле **init.ora**. Когда определенный фоновый процесс прерывается или какая-либо операция завершается неудачно, происходит формирования файла трассировки процесса. При этом в него сразу записывается информация о возникшей ошибке. Далее администратор БД, может провести "следствие", выяснив, что же послужило причиной той или иной сбойной ситуации. Либо когда что-то не совсем ясно, может передать файл трассировки в службу **Oracle World Wide Web Customer Support**, где ему помогут разобраться с данной проблемой. Так же существует файл **alert.log**, в который записывается информация при аварийном прерывании какого-либо фонового процесса, что так же не маловажно для администратора БД. Вообще по большому счету, чем лучше ваш Администратор БД понимает и представляет функционирование сервера **Oracle**, тем меньше вероятности, что вы лишитесь вашей ценной информации! :) Но, это так к слову. Одним из немаловажных выводов, который можно сделать из прошлых шагов это то, что, собственно, устойчивое функционирование БД в основном определяется правильной и безотказной работой фоновых процессов БД. Ранее мы с ними уже успели познакомиться, по этому повторяться не буду.

А получить информацию о работе фоновых процессов можно при помощи команд операционной системы. Так, например, в **OS UNIX** каждый фоновый процесс **Oracle** представляет собой отдельную задачу - **task**. Полезно рассматривать информацию, которую предоставляет ваша ОС относительно использования задач памяти и процессора. Например, в **OS UNIX** это команды, **sar, ps, vmstat, top**. Но **OS UNIX** - это отдельная тема, а мы с вами более детально рассматриваем **Windows NT**. А, вот здесь картина более утонченная. :) Все дело в том, что весь экземпляр БД **Oracle** в данной ОС выполнен как единый фоновый процесс (интересная формулировка, не правда ли, экземпляр - фоновый процесс) самой ОС и называется - сервисом. А, вот те самые фоновые процессы БД, в данном случае выполнены в **NT** как потоки. Они находятся внутри самого сервиса собственно экземпляра БД и ведут постоянное взаимодействие с ним. В самой **NT** имеется множество средств для трассировки и мониторинга сервисов. А средства администрирования потоков здесь несколько сложнее. Одним из средств мониторинга потоков может служить **Performance Monitor**. С его помощью можно анализировать использование памяти и переключателей контекста для всех потоков, которые принадлежат данному сервису. Эта утилита обычно штатная для **NT**. Для примера давайте дадим вот такой запрос:

```
SELECT a.spid, b.name FROM v$process a, v$bgprocess b
WHERE a.addr = b.paddr
/
```

Получаем:

```
SQL> SELECT a.spid, b.name FROM v$process a, v$bgprocess b
2  WHERE a.addr = b.paddr
3  /
```

SPID	NAME
1284	PMON
1400	DBW0
964	LGWR
1424	CKPT
1428	SMON
1416	RECO
1436	CJQ0
1464	QMNO

8 строк выбрано.

Теперь, если перевести десятичный столбец **SPID** в шестнадцатеричный, то можно получить соответствие идентификатора потока **NT** с фоновым процессом **Oracle**. Что-то вроде вот этого:

SPID(HEX)	NAME
504	PMON
578	DBW0
3C4	LGWR
590	CKPT
594	SMON
588	RECO
59C	CJQ0
5B8	QMNO

У вас эти значения могут быть другими, но это не важно. Я понимаю, что данная информация немного суховатая, но необходимая для понимания и освоения нашей с вами цели, по этому пока отдохните и продолжим в следующий раз. :)

Шаг 83 - Представления V\$Process и V\$Bgprocess

Итак, следуем далее по пути мониторинга экземпляра. Для того, чтобы проводить детальный анализ состояния БД, до самых ее "недр", в ваше распоряжение, как администратора, предоставляется такой инструмент, как системные представления. Они, как правило, начинаются с символа - **V\$...** Для примера с помощью этих представлений можно провести мониторинг процессов пользователя и проследить даже за фоновыми процессами. Начнем с такого достаточно удобного средства мониторинга, как представление **V\$Process**. Данное представление содержит информацию о всех процессах, которые подключены к БД, в том числе о фоновых и процессах пользователя. Следующее представление, на которое следует обратить внимание **V\$Bgprocess**. Оно содержит, список возможных фоновых процессов, причем в представлении включен дополнительный столбец **PADDR** в котором выводится шестнадцатеричный адрес выполняемого фонового процесса. Когда процесс не запущен столбец **PADDR** имеет значение 0. Давайте рассмотрим столбцы представления **V\$Process** более детально, так как это пригодится вам в дальнейшем. Итак напомним, что лучше это производить в **SQLPlus** - так как он в этом случае наиболее удобен. Так же не забудьте что, для того, чтобы работать с системными представлениями, вам нужны права администратора, по этому заходите пользователем **SYS** либо **SYSTEM!** Запускайте **SQLPlus** и введите следующую команду:

```
desc v$process;
```

Получаем:

```
SQL> desc v$process;
Имя                Пусто?  Тип
-----
ADDR                RAW(4)
PID                 NUMBER
SPID                VARCHAR2(12)
USERNAME            VARCHAR2(15)
SERIAL#             NUMBER
TERMINAL            VARCHAR2(16)
PROGRAM             VARCHAR2(64)
TRACEID             VARCHAR2(255)
BACKGROUND          VARCHAR2(1)
LATCHWAIT           VARCHAR2(8)
LATCHSPIN           VARCHAR2(8)
PGA_USED_MEM        NUMBER
PGA_ALLOC_MEM        NUMBER
PGA_FREEABLE_MEM    NUMBER
PGA_MAX_MEM         NUMBER
```

```
SQL>
```

А теперь приведем описание наиболее интересных из них:

Столбец	Описание
ADDR	Адрес процесса Oracle .
PID	Идентификатор (ID) процесса Oracle .
SPID	Идентификатор системного процесса OS .

USERNAME	Владелец процесса OS .
SERIAL#	Номер процесса Oracle .
TERMINAL	Идентификатор терминала OS (другими словами, имя вашего компьютера в сети. В моем случае ORAHOME!)
PROGRAM	Подключение программы OS (другими словами имя exe -ка вашей проги, которая работает с Oracle в данный момент!)
BACKGROUND	1 для фонового процесса и NULL для процесса пользователя.

Далее вводите следующую команду для того, чтобы получить описание представления **V\$bgprocess**:

```
desc v$bgprocess;
```

Получаем:

```
SQL> desc v$bgprocess;
Имя          Пусто?  Тип
-----
PADDR          RAW(4)
NAME           VARCHAR2(5)
DESCRIPTION    VARCHAR2(64)
ERROR          NUMBER
```

```
SQL>>
```

И еще описание наиболее интересных столбцов:

Столбец	Описание
PADDR	Адрес процесса Oracle (то же, что и в столбце ADDR представления v\$process)
NAME	Наименование фонового процесса
DESCRIPTION	Описание фонового процесса
ERROR	Код ошибки (0, если ошибка отсутствует)

А вот теперь давайте попробуем получить адреса и наименования работающих фоновых процессов на моем домашнем сервере **Oracle**. Запишем вот такой запрос с применением объединения этих двух представлений и посмотрим, что получится:

```
SELECT NVL(a.spid,'NULL') SPID, NVL(b.name,'NULL') NAME
FROM v$process a, v$bgprocess b
WHERE b.paddr(+) = a.addr
/
```

Получаем:

```
SQL> SELECT NVL(a.spid,'NULL') SPID, NVL(b.name,'NULL') NAME
2 FROM v$process a, v$bgprocess b
3 WHERE b.paddr(+) = a.addr
```

4 /

SPID	NAME
NULL	NULL
2080	PMON
2112	DBW0
2136	LGWR
2160	CKPT
2208	SMON
2220	RECO
2224	CJQ0
2228	QMN0
2232	NULL
2236	NULL

SPID	NAME
2536	NULL

12 строк выбрано.

Сразу замечу, что операнд **b.paddr(+)** = **a.addr** дает правое внешнее объединение, к ним мы еще вернемся. А пока можете убедиться, что все процессы сервера, успешно запущены и занимаются своими делами. Вот так применяя **V\$** представления можно получать всю информацию о состоянии БД.

Шаг 84 - Представление V\$session

Продолжаем мониторинг :) Давайте подробнее остановимся на представлении **V\$session**. Посмотрим сколько оно содержит полей запускаем **SQLPlus**:

```
desc v$session;
```

Получаем:

```
SQL> desc v$session;
Имя                Пусто?  Тип
-----
SADDR              RAW(4)
SID                NUMBER
SERIAL#            NUMBER
AUDSID             NUMBER
PADDR              RAW(4)
USER#              NUMBER
USERNAME           VARCHAR2(30)
COMMAND            NUMBER
OWNERID            NUMBER
TADDR              VARCHAR2(8)
LOCKWAIT           VARCHAR2(8)
STATUS             VARCHAR2(8)
SERVER             VARCHAR2(9)
SCHEMA#            NUMBER
SCHEMANAME          VARCHAR2(30)
OSUSER             VARCHAR2(30)
PROCESS            VARCHAR2(12)
MACHINE            VARCHAR2(64)
TERMINAL           VARCHAR2(16)
PROGRAM            VARCHAR2(64)
TYPE               VARCHAR2(10)
SQL_ADDRESS        RAW(4)
SQL_HASH_VALUE     NUMBER
PREV_SQL_ADDR      RAW(4)
PREV_HASH_VALUE    NUMBER
MODULE             VARCHAR2(48)
MODULE_HASH        NUMBER
ACTION             VARCHAR2(32)
ACTION_HASH        NUMBER
CLIENT_INFO        VARCHAR2(64)
FIXED_TABLE_SEQUENCE NUMBER
ROW_WAIT_OBJ#      NUMBER
ROW_WAIT_FILE#     NUMBER
ROW_WAIT_BLOCK#    NUMBER
ROW_WAIT_ROW#      NUMBER
LOGON_TIME         DATE
LAST_CALL_ET       NUMBER
PDML_ENABLED       VARCHAR2(3)
FAILOVER_TYPE      VARCHAR2(13)
FAILOVER_METHOD    VARCHAR2(10)
```

```

FAILED_OVER          VARCHAR2(3)
RESOURCE_CONSUMER_GROUP  VARCHAR2(32)
PDML_STATUS          VARCHAR2(8)
PDDL_STATUS          VARCHAR2(8)
PQ_STATUS            VARCHAR2(8)
CURRENT_QUEUE_DURATION    NUMBER
CLIENT_IDENTIFIER     VARCHAR2(64)

```

SQL>

Теперь опишем наиболее интересные поля:

Поле	Описание
SID	Идентификатор сеанса
SERIAL#	Номер сеанса
PADDR	Адрес сеанса родителя
USER#	Идентификатор пользователя в Oracle (Выбирается из таблицы SYS.USER\$)
USERNAME	Имя пользователя в Oracle
COMMAND	Текущая команда, выполняемая в этом сеансе. Соответствие номеров и команд задается в таблице SYS.AUDIT_ACTIONS .
STATUS	Статус сеанса (ACTIVE, INACTIVE, KILLED)
SERVER	Тип подключения к серверу (DEDICATED, SHARED, PSEUDO, NONE)
OSUSER	Имя пользователя в OS
PROGRAM	Программа OS , которая выполнила подключение к БД.
TERMINAL	Имя или тип терминала с которого выполнено подключение.
TYPE	Тип терминала (BACKGROUND или USER)
SQL_ADDRESS SQL_HASH_VALUE	Используется для уникальной идентификации текущего выполняемого выражения SQL .

Вот вам для примера довольно мудреный запрос, если его выполнить просто так, то полученная информация, будет не очень наглядно, по этому пока этого не делайте. Вот сам запрос:

```

SELECT
  b.name bgproc, p.spid, s.sid, p.serial#, s.osuser,
  s.username, s.terminal,
  DECODE(a.name, 'UNKNOWN', '-----', a.name) action
FROM
  v$process p, v$session s, v$bgprocess b,
  sys.audit_actions a
WHERE
  p.addr=s.paddr(+) AND b.paddr(+) = s.paddr AND
  a.action = NVL(s.action, 0)
ORDER BY
  sid;

```

Сразу понять, то что он делает не совсем просто! Так что, я тут посидел и перевел этот запрос, в более удобоваримый, вариант, хотя как сказать! :) Получился вот такой неименованный блок:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
CURSOR IS_MON IS
```

```
SELECT
```

```
  b.name, p.spid, s.sid, p.serial#, s.osuser,  
  s.username, s.terminal, DECODE(a.name, 'UNKNOWN', '-----', a.name)
```

```
FROM
```

```
  v$process p, v$session s, v$bgprocess b,  
  sys.audit_actions a
```

```
WHERE
```

```
  p.addr=s.paddr(+) AND b.paddr(+) = s.paddr AND  
  a.action = NVL(s.action, 0)
```

```
ORDER BY
```

```
  sid;
```

```
  p_spid v$process.spid%TYPE;  
  p_serial v$process.serial#%TYPE;  
  s_sid v$session.sid%TYPE;  
  s_osuser v$session.osuser%TYPE;  
  s_username v$session.username%TYPE;  
  s_terminal v$session.terminal%TYPE;  
  b_name v$bgprocess.name%TYPE;  
  a_name sys.audit_actions.name%TYPE;
```

```
BEGIN
```

```
DBMS_OUTPUT.enable;
```

```
DBMS_OUTPUT.put_line('BGProg'|| '||SPID'|| '||SID'|| '||Serial'||  
  ' ||OSUser'|| '||UserName'|| '||Terminal'|| '||DB Action');
```

```
DBMS_OUTPUT.put_line('-----'|| '||----'|| '||---'|| '||-----'||  
  ' ||'-----'|| '||-----'|| '||-----'|| '||-----');
```

```
OPEN IS_MON;
```

```
LOOP
```

```
  FETCH IS_MON INTO b_name, p_spid, s_sid, p_serial, s_osuser, s_username, s_terminal, a_name;
```

```
  EXIT WHEN IS_MON%NOTFOUND;
```

```
  DBMS_OUTPUT.put_line(RPAD(NVL(b_name,'NULL'),8,' ')||RPAD(NVL(p_spid,'NULL'),6,' ')||  
    RPAD(TO_CHAR(NVL(s_sid, 0)),5,' ')||RPAD(TO_CHAR(NVL(p_serial,0)),9,' ')||  
    RPAD(NVL(s_osuser,'NULL'),16,' ')||RPAD(NVL(s_username,'NULL'),10,' ')||  
    RPAD(NVL(s_terminal,'NULL'),10,' ')||a_name);
```

```
END LOOP;
```

```
CLOSE IS_MON;
```

```
END;  
/
```

После прогона в **SQLPlus** получаем:

```
SQL> SET SERVEROUTPUT ON  
SQL>  
SQL> DECLARE  
2  
3 CURSOR IS_MON IS  
4 SELECT  
5   b.name, p.spid, s.sid, p.serial#, s.osuser,  
6   s.username, s.terminal, DECODE(a.name, 'UNKNOWN', '-----', a.name)  
7 FROM  
8   v$process p, v$session s, v$bgprocess b,  
9   sys.audit_actions a  
10 WHERE  
11  p.addr=s.paddr(+) AND b.paddr(+) = s.paddr AND  
12  a.action = NVL(s.action, 0)  
13 ORDER BY  
14  sid;  
15  
16   p_spid v$process.spid%TYPE;  
17   p_serial v$process.serial#%TYPE;  
18   s_sid v$session.sid%TYPE;  
19   s_osuser v$session.osuser%TYPE;  
20   s_username v$session.username%TYPE;  
21   s_terminal v$session.terminal%TYPE;  
22   b_name v$bgprocess.name%TYPE;  
23   a_name sys.audit_actions.name%TYPE;  
24  
25 BEGIN  
26  
27 DBMS_OUTPUT.enable;  
28 DBMS_OUTPUT.put_line('BGProg'|| ' ||'SPID'|| ' ||'SID'|| ' ||'Serial'||  
29   ' ||'OSUser'|| ' ||'UserName'|| ' ||'Terminal'|| ' ||'DB Action');  
30 DBMS_OUTPUT.put_line('-----'|| ' ||'----'|| ' ||'---'|| ' ||'-----'||  
31   ' ||'-----'|| ' ||'-----'|| ' ||'-----'|| ' ||'-----');  
32  
33  
34 OPEN IS_MON;  
35  
36 LOOP  
37  
38  FETCH IS_MON INTO b_name, p_spid, s_sid, p_serial, s_osuser, s_username, s_terminal,  
a_name;  
39  
40  EXIT WHEN IS_MON%NOTFOUND;  
41  
42  DBMS_OUTPUT.put_line(RPAD(NVL(b_name,'NULL'),8,' ')||RPAD(NVL(p_spid,'NULL'),6,' ')||  
43    RPAD(TO_CHAR(NVL(s_sid, 0)),5,' ')||RPAD(TO_CHAR(NVL(p_serial,0)),9,' ')||  
44    RPAD(NVL(s_osuser,'NULL'),16,' ')||RPAD(NVL(s_username,'NULL'),10,' ')||
```

```

45      RPAD(NVL(s_terminal,'NULL'),10,' ')||a_name);
46
47 END LOOP;
48
49 CLOSE IS_MON;
50
51 END;
52 /

```

BGProg	SPID	SID	Serial	OSUser	UserName	Terminal	DB	Action
PMON	2080	1	1	SYSTEM	NULL	ORAHOME	----	
DBW0	2112	2	1	SYSTEM	NULL	ORAHOME	----	
LGWR	2136	3	1	SYSTEM	NULL	ORAHOME	----	
CKPT	2160	4	1	SYSTEM	NULL	ORAHOME	----	
SMON	2208	5	1	SYSTEM	NULL	ORAHOME	----	
RECO	2220	6	1	SYSTEM	NULL	ORAHOME	----	
CJQ0	2224	7	1	SYSTEM	NULL	ORAHOME	----	
QMNO	2228	8	1	SYSTEM	NULL	ORAHOME	----	
NULL	2536	9	3	ORAHOME\sergl	SYS	ORAHOME	----	
NULL	NULL	0	0	NULL	NULL	NULL	----	
NULL	2232	0	1	NULL	NULL	NULL	----	
NULL	2236	0	1	NULL	NULL	NULL	----	

Процедура PL/SQL успешно завершена.

```

SQL>
SQL> EXIT

```

Вот тут очень хорошо видно, что там происходит, т.е. внутри вашего сервера **Oracle**. Здесь **ORAHOME\sergl** это имя моего 2003-го сервера, на котором у меня стоит **Oracle 9.2.0** и **sergl** это пользователь моего сеанса, на сервере. Если посмотреть внимательно, то будет хорошо видно, что здесь и для чего, если кто не разобрался пишите, расскажу. :)

Шаг 85 - Другие представления для мониторинга

Продолжаем идти дальше по пути мониторинга. Итак, запросив представление **v\$access** можно получить информацию о тех объектах БД, к которым в текущий момент обращаются пользователи. Его можно использовать для обнаружения посторонних приложений или недокументированных процедур, которые обращаются к БД и для решения проблем безопасности данных. Так же представление **v\$mts** содержит информацию, о разделяемых процессах сервера. Оно содержит информацию о максимальном количестве подключений, запущенных серверах, прерванных серверах и загрузке серверов.

Мониторинг разделяемой глобальной системной области.

Для мониторинга ГСО можно использовать представление **v\$sqlarea**, оно содержит информацию о выражениях **SQL**, которые находятся в разделяемой области, включая их текст, количество пользователей, использовавших, это выражение, количество блоков на диске и в памяти, к которым произошло обращение в процессе выполнения команд. К примеру, используя столбцы **disk_reads** и **buffer_gets** можно отследить количество блоков, которые считаны с диска и из кэш буфера. Так можно определить запросы, наиболее интенсивно использующие ресурсы БД. Так же представление пользователя **v\$open_cursor** помогает найти незакрытые курсоры. Например, запустите последний запрос из предыдущего шага и посмотрите число в столбце **SID** пользователя:

```

          SID
NULL  2536 9   3  ORAHOME\sergl SYS    ORAHOME  -----

```

У меня было число 9 как видно из примера. А теперь дайте вот такой запрос:

```

SELECT b.piece, a.sql_text
FROM v$open_cursor a, v$sqltext b
WHERE
  a.sid = 9 and          -- 9 My SID
  a.address = b.address and
  a.hash_value = b.hash_value
ORDER BY
  b.address, b.hash_value, b.piece asc
/

```

После **SQLPlus** получаем:

```

SQL> SELECT b.piece, a.sql_text
2  FROM v$open_cursor a, v$sqltext b
3  WHERE
4  a.sid = 9 and          -- 9 My SID
5  a.address = b.address and
6  a.hash_value = b.hash_value
7  ORDER BY
8  b.address, b.hash_value, b.piece asc
9  /

PIECE SQL_TEXT
-----
0 SELECT b.piece, a.sql_text FROM v$open_cursor a, v$sqltext b

```

```
1 SELECT b.piece, a.sql_text FROM v$open_cursor a, v$sqltext b
2 SELECT b.piece, a.sql_text FROM v$open_cursor a, v$sqltext b
3 SELECT b.piece, a.sql_text FROM v$open_cursor a, v$sqltext b
```

SQL>

У меня четыре курсора в работе. Где представление **v\$sqltext** показывает текст запроса. Большие выражения извлекаются последовательно, согласно столбца **PIECE** представления **v\$sqltext**.

Мониторинг ГСО.

Для получения информации о процессах ГСО, используют два представления **v\$sga** и **v\$sgqstat**. Первое содержит информацию, о размерах каждого компонента ГСО - кэше журнала регистрации транзакций, кэш буфере данных и разделяемом пуле. Второе представление содержит более подробную информацию о размерах всех компонентов, даже самых мелких, включая размеры стековых областей различного назначения. Например, для получения сведений о размере свободной памяти в ГСО дадим следующий запрос:

```
SELECT pool, bytes FROM v$sgastat
WHERE name = 'free memory'
/
```

Получаем:

```
SQL> SELECT pool, bytes FROM v$sgastat
2 WHERE name = 'free memory'
3 /
```

POOL	BYTES
shared pool	45061772
large pool	8388608
java pool	33554432

Думаю хорошо видно, сколько и где свободно памяти.

Мониторинг кэшей, библиотек и словаря.

Получит информацию о данных объектах можно, посредством представлений **v\$librarycache** и **v\$rowcache**. Первое содержит данные о каждом типе объекта, второе аналогичную информацию о КЭШе словаря данных.

Мониторинг архивирования.

Информацию о процессах архивирования, можно найти в представлении **v\$archive**. Запросы к этому представлению позволяют узнать о работе процесса **ARCH**.

Вот вкратце, основные но далеко не все данные о мониторинге экземпляра вашей БД. Если приложить усилия, то во всем этом можно разобраться подробнее, обратившись к документации по серверу **Oracle**. Хотя может кто-то пойдет и дальше ... кто знает? :)

Шаг 86 - PL/SQL - Язык программирования Oracle

В "[Шаг 73 - PL/SQL - функции PL/SQL осталось немного ...](#)" мы с вами закончили близкое рассмотрение функций **PL/SQL** - вот теперь давайте вернемся к этому вопросу, а именно изучению языка **PL/SQL**. В "[Шаг 38 - PL/SQL - вводный курс](#)" я уже останавливался на описании языка **PL/SQL**, но кое что, еще осталось и думаю, что это вам будет не безынтересно. Итак **SQL** - в **PL/SQL** соответствует стандарту **ANSI** (American National Standards Institute - Американский национальный институт стандартов.) определенному документом **ANSI X3.135-1992 "Database Language SQL"**. Именно этот стандарт известный как **SQL92** (или **SQL2**), сам по себе определяет язык **SQL** и не описывает **3GLPL/SQL**. Стандарт **SQL92** имеет три уровня согласования: **Entry** (начальный), **Intermediate** (промежуточный) и **Full** (полный). Oracle 7 варианта 7.2 - (и все другие версии более высокого уровня, в том числе Oracle8) соответствует стандартам **Entry SQL92**, что одобрено **NIST** (National Institute for Standards and Technology - Национальный Институт стандартов и технологий). В настоящее время Oracle - работает с **ANSI**, чтобы обеспечить, соответствие будущих версий Oracle и **PL/SQL** полному стандарту. Может показаться неожиданным, но язык **PL/SQL** разработан на основе языка третьего поколения **Ada** (хотя мне лично, как кажется я уже говорил он очень напоминает язык **Pascal**!). Многие конструкции, применяемые в **Ada** можно найти в **PL/SQL** - такие, например как модули - так называемые пакеты. Но о них мы поговорим позже. Как Вы, наверное, поняли базовой единицей языка **PL/SQL** является блок (block), который имеет следующую структуру:

DECLARE

.....

BEGIN

.....

EXCEPTION

.....

END;

Думаю, знакомая вам конструкция!? :) Это так называемый анонимный блок. Такой блок каждый раз при выполнении компилируется. Так же он не хранится в базе данных и не может быть вызван из другого блока. В данном случае приступим к изучению именованных блоков - процедур и функций. Такие блоки хранятся в базе данных и могут быть использованы повторно. А вот это не маловажный фактор! Итак, собственно первая конструкция, которую мы рассмотрим, это именованный блок или ПРОЦЕДУРА. Синтаксис его объявления таков:

```
----- CREATE [OR REPLACE] PROCEDURE - имя_процедуры -----  
-----  
----- (аргумент [IN] [OUT] [IN OUT] тип, ..... ) AS [IS] -----  
----- тело процедуры -----
```

где:

- (имя_процедуры) - имя создаваемой процедуры,
- (аргумент) - имя параметра процедуры,
- (тип) - это тип соответствующего параметра,
- (тело процедуры) - это блок PL/SQL в котором содержится текст процедуры.

Для изменения программного кода процедуры ее необходимо удалить, а затем создать вновь, для того, чтобы делать это за один проход применяется дополнительный оператор объявления REPLACE, что буквально понимается, как заменить. Если его не применять, то процедуру необходимо удалить с помощью оператора DROP PROCEDURE имя. Вот таким образом, определяется данная конструкция. Далее мы с вами поучимся создавать процедуры.

Шаг 87 - PL/SQL - Именованные блоки процедуры

Закончив знакомство с синтаксисом и объявлениями, давайте наконец, займемся делом. Напишем нашу с вами первую процедуру и познакомимся со всем, что с этим связано. Итак, запускаем **SQL*Plus** и отправим на исполнение следующий код: (программирование однако!)

```
CREATE PROCEDURE TESTPRG
AS

BEGIN

    NULL;

END TESTPRG;
/
```

Получаем после компиляции:

```
SQL> CREATE PROCEDURE TESTPRG
  2 AS
  3
  4 BEGIN
  5
  6 NULL;
  7
  8 END TESTPRG;
  9 /
```

Процедура создана.

SQL>

Ура! Наша первая с вами процедура создана и находится внутри нашего сервера и не только находится, но еще и готова к дальнейшему использованию! Итак, для начала давайте убедимся, что она есть и здорова! Введем следующий запрос:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_TYPE = 'PROCEDURE'
/
```

Получаем после прохода:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
  2 FROM USER_OBJECTS
  3 WHERE OBJECT_TYPE = 'PROCEDURE'
  4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TESTPRG	PROCEDURE	VALID

SQL>

Что ж теперь в вашем владении есть процедура **TESTPRG** и она совершенно исправна, о чем говорит запись в поле **STATUS** со значением **VALID**. К слову сказать, при выполнении оператора **CREATE PROCEDURE** вследствие того, что он является оператором **DDL**, то неявно вызывается оператор **COMMIT**. Помните в [шаре 6](#), я намеренно сделал ошибку сказав, что сделайте **COMMIT** после оператора **GRANT**! После чего получил кучу нареканий, от некоторых гугу жанра! Но, я это сделал по той простой причине, а впрочем какая разница, сделал я это намеренно или нет! :) Идем дальше. Итак, теперь надеюсь кое-что стало яснее. При первом приближении хорошо видно, что наша процедура совершенно бесполезна, так как, в объявлении ее тела стоит только оператор **NULL**, который как мы знаем ничего собственно не делает кроме как объявляет сам себя. Но это не так ведь мы с вами учимся, значит это не совсем бесполезно! :) Оставим в покое нашу процедуру, в плане полезности и попробуем, немного переделать ее:

```
CREATE PROCEDURE TESTPRG
IS
BEGIN
  -- Empty Operator -----
  NULL;
END TESTPRG;
/
```

Получаем:

```
SQL> CREATE PROCEDURE TESTPRG
2  IS
3
4  BEGIN
5    -- Empty Operator -----
6    NULL;
7
8  END TESTPRG;
9  /
CREATE PROCEDURE TESTPRG
*
```

ошибка в строке 1:

ORA-00955: имя уже задействовано для существующего объекта

Ой - а вот здесь, как раз нужно было ее удалить, а потом вновь создать, например вот так:

```
DROP PROCEDURE TESTPRG
/
```

Увидим:

```
SQL> DROP PROCEDURE TESTPRG
2  /
```

Процедура удалена.

Но, это не совсем рентабельно, давайте снова создадим нашу с вами процедуру, но несколько по другому:

```
CREATE OR REPLACE PROCEDURE TESTPRG
AS
BEGIN
    NULL;
END TESTPRG;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE PROCEDURE TESTPRG
2 AS
3
4 BEGIN
5
6 NULL;
7
8 END TESTPRG;
9 /
```

Процедура создана.

SQL>

Вот теперь изменить ее можно так же за один проход не применяя **DROP PROCEDURE TESTPRG!** Попробуем проделать следующее:

```
CREATE OR REPLACE PROCEDURE TESTPRG
IS
BEGIN
    -- Empty Operator -----
    NULL;
END TESTPRG;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE PROCEDURE TESTPRG
2 IS
3
4 BEGIN
5     -- Empty Operator -----
6     NULL;
7
8 END TESTPRG;
```

9 /

Процедура создана.

SQL>

Все что мы сделали - это добавили строку комментария и сменили **AS** на **IS**, что само по себе не принципиально, но позволяет показать как можно создать процедуру изменить ее или удалить! Вот и все три действия, которыми мы как правило пользуемся! Думаю, что теперь вы сами можете что-нибудь создать! :)

Шаг 88 - PL/SQL - Работаем с процедурами

Вот теперь давайте создадим, что то осмысленное и попытаемся получить хоть какой-то результат... Снова вступает в бой **MILLER/KOLOBOK** наперевес с **SQL*Plus**. Запишем такую процедуру:

```
CREATE OR REPLACE PROCEDURE TESTPRGTWO
IS
BEGIN
    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line('HELLO!!! I AM TESTPRGTWO!!! REMEMBER ME?!');
END TESTPRGTWO;
/
```

После компиляции получаем:

```
SQL> CREATE OR REPLACE PROCEDURE TESTPRGTWO
2  IS
3
4  BEGIN
5
6  DBMS_OUTPUT.enable;
7      DBMS_OUTPUT.put_line('HELLO!!! I AM TESTPRGTWO!!! REMEMBER ME?!');
8
9  END TESTPRGTWO;
10 /
```

Процедура создана.

SQL>

Итак, вот теперь наша процедура выводит приветственное послание - **HELLO!!! I AM TESTPRGTWO!!! REMEMBER ME?!** Но как в этом убедиться, что она вообще, что то выводит? Можно вот так - применив, оператор **EXECUTE**:

```
SET SERVEROUTPUT ON
EXEC TESTPRGTWO;
```

Увидим:

```
SQL> SET SERVEROUTPUT ON
SQL> EXEC TESTPRGTWO;
HELLO!!! I AM TESTPRGTWO!!! I HEVE REPLACE MYSELF NOW!
```

Процедура PL/SQL успешно завершена.

Вот теперь судя по последней записи, совершенно ясно, что наша процедура откомпилирована, хранится в БД (поэтому она еще называется хранимая процедура) и выполняет определенные

действия. Есть еще и второй способ, это вызвать процедуру на исполнение напрямую из анонимного блока, скажем вот так:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
BEGIN
```

```
TESTPRGTWO;
```

```
END;
```

```
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> DECLARE
```

```
2
```

```
3 BEGIN
```

```
4
```

```
5 TESTPRGTWO;
```

```
6
```

```
7 END;
```

```
8 /
```

```
HELLO!!! I AM TESTPRGTWO!!! I HEVE REPLACE MYSELF NOW!
```

Процедура PL/SQL успешно завершена.

Теперь, анонимный блок вызвал именованный, и после выполнения завершился. Теперь, у нас с вами две процедуры, одна из которых, вообще ничего не делает, а другая приветствует, да и то, только если попросят! :) Давайте посмотрим запрос из [прошлого шага](#), но более полный:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
```

```
2 FROM USER_OBJECTS
```

```
3 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
-------------	-------------	--------

BOYS	TABLE	VALID
CUSTOMERS	TABLE	VALID
GIRLS	TABLE	VALID
OFFICES	TABLE	VALID
ORDERS	TABLE	VALID
PRODUCTS	TABLE	VALID
SALESREPS	TABLE	VALID
SYS_C003505	INDEX	VALID
SYS_C003506	INDEX	VALID
SYS_C003507	INDEX	VALID
SYS_C003511	INDEX	VALID
SYS_C003512	INDEX	VALID

SYS_C003513	INDEX	VALID
SYS_C003515	INDEX	VALID
TESTPRG	PROCEDURE	VALID
TESTPRGTWO	PROCEDURE	VALID

16 rows selected

SQL>

Вот теперь хорошо видно, что у нас с вами семь таблиц, семь индексов и, о чудо! Две процедуры! Надеюсь, что вскоре их станет больше! :)

Шаг 89 - PL/SQL - Процедуры и их параметры

Наверное вы уже заметили, что в наших прошлых процедурах, явно чего-то не хватает. Думаю, если кто-либо из вас знаком с языками программирования, например **C++** или **Pascal**, то вы могли задать вопрос, а можно в **PL/SQL** - процедурах передавать параметры? Да не просто можно, а нужно! И работа с параметрами процедур **PL/SQL** довольно интересна! Давайте попробуем написать процедуру с параметрами:

```
CREATE OR REPLACE PROCEDURE TESTPRM(NUM IN NUMBER)
IS
    in_COMP VARCHAR2(50);

BEGIN

    SELECT COMPANY INTO in_COMP FROM customers
    WHERE customers.CUST_NUM = NUM;

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(in_COMP);

END TESTPRM;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE PROCEDURE TESTPRM(NUM IN NUMBER)
2  IS
3
4  in_COMP VARCHAR2(50);
5
6  BEGIN
7
8  SELECT COMPANY INTO in_COMP FROM customers
9  WHERE customers.CUST_NUM = NUM;
10
11  DBMS_OUTPUT.enable;
12  DBMS_OUTPUT.put_line(in_COMP);
13
14  END TESTPRM;
15  /
```

Процедура создана.

Здесь, я применил параметр **NUM** с типом **NUMBER**. **IN** в данном случае определяет, что параметр является входным. Что так же справедливо и по умолчанию. То есть, если параметр имеет только тип **IN**, то предикат **IN** можно не указывать. Так же напомним, что **NUM** является формальным параметром функции (**formal parameters**). А вот входное значение функции является фактическим параметром (**actual parameters**). Формальный параметр **NUM**, является только вместилищем для фактически передаваемого параметра и все операции производятся с

формальным параметром (в программировании, если я правильно помню, это называется передача параметра в процедуру по значению).

Сам предикат **IN** означает следующее: **IN** - Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве параметра только для чтения - он не может быть изменен. Когда процедура завершается и управление передается в вызывающую среду, фактический параметр не изменяется.

Что, собственно я и говорил выше. Вернемся к нашей процедуре, она содержит, один оператор **SELECT** (уже что-то полезное!), который выбирает одну запись из таблицы **customers**, в данном случае заказчика по его номеру. Для примера вызовем ее из анонимного блока предварительно определив и передав ей параметр:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
BEGIN
```

```
TESTPRM(2112);
```

```
END;
```

```
/
```

Получаем после вызова:

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> DECLARE
```

```
2
```

```
3 BEGIN
```

```
4
```

```
5 TESTPRM(2112);
```

```
6
```

```
7 END;
```

```
8 /
```

```
Апельсин
```

Процедура PL/SQL успешно завершена.

Вот такой "желтый и круглый" заказчик! :) Ясно видно, как сработал оператор **SELECT** и функция **TESTPRM** вывела результат. При этом фактический параметр, был передан и не изменялся. А теперь попробуем написать более сложную процедуру:

```
CREATE OR REPLACE PROCEDURE TESTOUT(NUM IN NUMBER, DT OUT VARCHAR2)
IS
```

```
BEGIN
```

```
    SELECT COMPANY INTO DT FROM customers
    WHERE customers.CUST_NUM = NUM;
```

```
END TESTOUT;  
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE PROCEDURE TESTOUT(NUM IN NUMBER, DT OUT VARCHAR2)  
2 IS  
3  
4 BEGIN  
5  
6 SELECT COMPANY INTO DT FROM customers  
7 WHERE customers.CUST_NUM = NUM;  
8  
9 END TESTOUT;  
10 /
```

Процедура создана.

Рассмотрим параметры более внимательно, про первый мы уже все знаем, а вот второй **DT** определен как **OUT** и имеет тип **VARCHAR2**. Что это значит? А, вот что. **DT** выходной параметр и принимает значение из самой процедуры на возврат, т.е.

OUT - Любое значение имеющее фактический параметр при вызове процедуры игнорируется. Внутри процедуры формальный параметр рассматривается как параметр только для записи - ему можно присвоить значение, но считать из него значение нельзя. (!) Когда процедура завершается и управление передается в вызывающую среду, содержимое формального параметра присваивается фактическому параметру.

Вот так работает тип **OUT**. Давайте определим анонимный блок и посмотрим, что же получается:

```
DECLARE  
  
FRDT VARCHAR2(100);  
  
BEGIN  
  
FRDT := 'HELLO';  
TESTOUT(2103, FRDT);  
DBMS_OUTPUT.enable;  
DBMS_OUTPUT.put_line('CUSTOMER '||FRDT);  
  
END;  
/
```

Получаем:

```
SQL> DECLARE  
2  
3 FRDT VARCHAR2(100);  
4  
5 BEGIN  
6
```



```
7  FRDT := 'HELLO';
8
9  TESTOUT(2103, FRDT);
10
11 DBMS_OUTPUT.enable;
12   DBMS_OUTPUT.put_line('CUSTOMER '||FRDT);
13
14 END;
15 /
CUSTOMER Крупное предприятие
```

Процедура PL/SQL успешно завершена.

Заметили, что значение **HELLO** не появилось? Естественно, так как параметр **DT** функции **TESTOUT** является выходным. Если сказать честно, то лично меня такой тип возврата данных из процедуры немного шокирует, так как в классическом программировании такое применяется по моему редко, хотя я и могу ошибаться. Но, тем не менее может быть в каком-то случае такой вызов вполне оправдан! Можете пока поработать со всем этим. :)

Шаг 90 - PL/SQL - Процедуры и их параметры II

Идем дальше по пути познания процедур. Итак, мы с вами рассмотрели тип передаваемых параметров - **IN** и **OUT**. Существует еще один тип параметров процедур, а именно как вы уже, наверное, догадываетесь это тип **IN OUT** - то есть при объявлении это выглядит примерно вот так:

```
CREATE OR REPLACE PROCEDURE MYPROC(MYPARAM IN OUT NUMBER .....
```

Вот с этим параметром давайте разберемся подробнее. Давайте сформулируем правила, для такого тип параметров.

IN OUT - этот вид представляет собой комбинацию видов **IN** и **OUT**. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр может быть считан и в него может быть записано значение. При завершении процедуры и возврате управления в вызывающую среду, содержимое формального параметра присваивается фактическому параметру.

Теперь давайте создадим следующую процедуру примерно вот такого вида:

```
CREATE OR REPLACE PROCEDURE TESTINOUT(NUM IN OUT NUMBER, DT OUT VARCHAR2)
IS
BEGIN
    SELECT COMPANY INTO DT FROM customers
    WHERE customers.CUST_NUM = NUM;

    SELECT CUST_REP INTO NUM FROM customers
    WHERE customers.CUST_NUM = NUM;

END TESTINOUT;
/
```

Не торопитесь ее компилировать, давайте порассуждаем. Итак, теперь **NUM** параметр **IN OUT** - как видно второй **SELECT** принимает его как параметр и возвращает через него значение. То есть принимает условие и вернет результат. Удобно, не так ли? Компилируем:

```
SQL> CREATE OR REPLACE PROCEDURE TESTINOUT(NUM IN OUT NUMBER, DT OUT VARCHAR2)
2  IS
3
4  BEGIN
5
6  SELECT COMPANY INTO DT FROM customers
7  WHERE customers.CUST_NUM = NUM;
8
9  SELECT CUST_REP INTO NUM FROM customers
10 WHERE customers.CUST_NUM = NUM;
11
12 END TESTINOUT;
13 /
```

Процедура создана.

Вот теперь если сделать вот такой вызов:

```
SET SERVEROUTPUT ON

DECLARE

FRDT VARCHAR2(100);
FRNM NUMBER := 2103;

BEGIN

    TESTINOUT(FRNM, FRDT);

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line('CUSTOMER '||FRDT||' CUST_REP '||TO_CHAR(FRNM));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3  FRDT VARCHAR2(100);
4  FRNM NUMBER := 2103;
5
6  BEGIN
7
8  TESTINOUT(FRNM, FRDT);
9
10 DBMS_OUTPUT.enable;
11   DBMS_OUTPUT.put_line('CUSTOMER '||FRDT||' CUST_REP '||TO_CHAR(FRNM));
12
13 END;
14 /
CUSTOMER Крупное предприятие CUST_REP 105
```

Процедура PL/SQL успешно завершена.

Не трудно заметить, что первое значение фактического параметра **FRNM** равно **2103**, а вот вернул он значение **105** - т.е. один параметр выполнил двойную работу. Вполне не плохо. К слову, если вы попытаетесь сделать что-то вроде:

```
SET SERVEROUTPUT ON

DECLARE

FRDT VARCHAR2(100);
```

```
BEGIN  
  
    TESTINOUT(2103, FRDT);  
  
END;  
/
```

То получите в ответ следующее:

```
DECLARE  
  
FRDT VARCHAR2(100);  
  
BEGIN  
  
    TESTINOUT(2103, FRDT);  
  
END;  
  
ORA-06550: Строка 7, столбец 12:  
PLS-00363: выражение '2103' не м.б. использовано как адресат назначения  
ORA-06550: Строка 7, столбец 2:  
PL/SQL: Statement ignored  
  
SQL>
```

В литерале 2103 нельзя сохранить возвращаемый результат, так как литерал не хранится в памяти после использования. Ну, что я думаю и так очевидно. Надеюсь, с данным типом параметра теперь вам все стало ясно! :)

Давайте теперь остановимся на понятии тела (body) процедуры. Тело процедуры содержит собственно исполняемый код и располагается между ключевыми словами **BEGIN** и **EXCEPTION**. Далее идет блок **EXCEPTION** и **END**, в котором располагается собственно обработчик исключительной ситуации. Раздел объявлений располагается между операторами **CREATE** и **IS** или **AS** (вот и снова язык Ada!) - а выглядит это все примерно вот так:

```
CREATE OR REPLACE PROCEDURE [имя процедуры] IS or AS  
  
    зона объявления переменных.  
  
BEGIN  
  
    выполняемый раздел  
  
EXCEPTION  
  
    раздел исключительных ситуаций  
  
END [имя процедуры]
```

Такой скелет имеет процедура и все что к ней прилагается. Запомните это получше! :)

Шаг 91 - PL/SQL - Процедуры и параметры III

Думаю, что вам уже кажется, что с процедурами и их параметрами мы разобрались, а вот и нет! Есть еще кое-что. Думаю, кто-то из вас заметил, что при написании формального параметра процедуры, например, определяя ее как **VARCHAR2** или **NUMBER** я никогда не делал вот так:

```
CREATE OR REPLACE PROCEDURE some_proc(NUM IN OUT NUMBER(3,2), DT OUT VARCHAR2(100))
```

И вот почему. Накладывать ограничения на формальные параметры функций в **PL/SQL** - ЗАПРЕЩЕНО! Например, вот такой пример, приведет к ошибке компиляции:

```
CREATE OR REPLACE PROCEDURE TESTINOUT(NUM IN OUT NUMBER(3,2), DT OUT VARCHAR2(100))
IS
BEGIN
SELECT COMPANY INTO DT FROM customers
WHERE customers.CUST_NUM = NUM;

SELECT CUST_REP INTO NUM FROM customers
WHERE customers.CUST_NUM = NUM;

END TESTINOUT;
/
```

В результате получите:

```
SQL> CREATE OR REPLACE PROCEDURE TESTINOUT(NUM IN OUT NUMBER(3,2), DT OUT
VARCHAR2(100))
2 IS
3
4 BEGIN
5
6 SELECT COMPANY INTO DT FROM customers
7 WHERE customers.CUST_NUM = NUM;
8
9 SELECT CUST_REP INTO NUM FROM customers
10 WHERE customers.CUST_NUM = NUM;
11
12 END TESTINOUT;
13 /
```

Предупреждение: Процедура создана с ошибками компиляции.

Что и требовалось доказать. Можете убрать неверные объявления и еще раз перекомпилировать процедуру для того, чтобы она осталась исправной. А, вот вам еще один подводный камешек. Запишем вот такую процедуру:

```
CREATE OR REPLACE PROCEDURE PTEST(I_PAR IN OUT NUMBER, II_PAR IN OUT VARCHAR2)
IS
```

```
BEGIN

  I_PAR := 15.6;
  II_PAR := 'POIUYTREWQLKJHGFDSA';

END PTEST;
/
```

Компилируем:

```
SQL> CREATE OR REPLACE PROCEDURE PTEST(I_PAR IN OUT NUMBER, II_PAR IN OUT VARCHAR2)
  2 IS
  3
  4 BEGIN
  5
  6   I_PAR := 15.6;
  7   II_PAR := 'POIUYTREWQLKJHGFDSA';
  8
  9 END PTEST;
10 /
```

Процедура создана.

Вот теперь **I_PAR** и **II_PAR** получили неявное ограничение посредством объявлений:

```
I_PAR := 15.6;
II_PAR := 'POIUYTREWQLKJHGFDSA';
```

т.е. получилось, что то вроде:

```
CREATE OR REPLACE PROCEDURE PTEST(I_PAR IN OUT NUMBER(3,4), II_PAR IN OUT
VARCHAR2(19))
```

Теперь, если произвести вот такой вызов:

```
DECLARE

V_STR VARCHAR2(10);
V_NUM NUMBER(3,4);

BEGIN

  PTEST(V_NUM, V_STR);

END;
/
```

Получаем, что-то довольно странное:

```
SQL> DECLARE
  2
  3 V_STR VARCHAR2(10);
```

```
4 V_NUM NUMBER(3,4);
5
6 BEGIN
7
8 PTEST(V_NUM, V_STR);
9
10 END;
11 /
DECLARE
*
```

ошибка в строке 1:

ORA-06502: PL/SQL: : буфер символьных строк слишком маленький ошибка числа или значения

ORA-06512: на "MILLER.PTEST", line 7

ORA-06512: на line 8

SQL>

Не сразу ясно, что происходит, так? А все очень просто, **V_STR VARCHAR2(10)** переопределила ограничение переменной **II_PAR** при ее явном вызове и запись строки длиной 19 символов в переменную всего в 10 символов привело к ошибке! Очень важно это понимать, иначе в дальнейшем вы запутаетесь совсем! Здесь ошибку вызвала сама вызывающая программа, а не код процедуры, как может показаться! Так вот во избежание ошибок, подобных **ORA-06502** при создании процедур документируйте все ограничения налагаемые на фактические параметры - вносите в хранимые процедуры комментарии, а так же кроме описания каждого параметра записывайте функции выполняемые самой процедурой! Вот тогда я думаю, у вас все получится!

Так же единственным способом наложения ограничения на формальный параметр функции является использование оператора **%TYPE**. Мы с вами о нем говорили. В свете этого можно переписать нашу функцию пример - скажем, вот так:

```
CREATE OR REPLACE PROCEDURE PTEST(
    I_PAR IN OUT CUSTOMERS.CUST_NUM%TYPE,
    II_PAR IN OUT CUSTOMERS.COMPANY%TYPE)
IS
BEGIN
    I_PAR := 15.6;
    II_PAR := 'POIUYTREWQLKJHGFDSA';

END PTEST;
/
```

Такой способ удобен тем, что при изменении полей таблицы автоматом меняются параметры процедур, что облегчает сопровождение кода хранимых процедур, не нужно менять все параметры связанные с данным полем! Получаем:

```
SQL> CREATE OR REPLACE PROCEDURE PTEST(
2     I_PAR IN OUT CUSTOMERS.CUST_NUM%TYPE,
3     II_PAR IN OUT CUSTOMERS.COMPANY%TYPE)
4 IS
5
```

```
6 BEGIN
7
8     I_PAR := 15.6;
9     II_PAR := 'POIUYTREWQLKJHGFDSA';
10
11 END PTEST;
12 /
```

Процедура создана.

Ошибок нет! Значит, все прошло успешно! Как работать с параметрами это дело вкуса, а на него, как говорится, товарищей совсем не бывает! Вот пока можете все это переварить, а я пойду попить чаю! :)

Шаг 92 - PL/SQL - Процедуры и параметры еще немного

Что ж давайте покончим с этими параметрами. Осталось рассмотреть тип применения параметров при передаче их в процедуру. Рассмотрим пример. Создадим процедуру следующего вида:

```
CREATE OR REPLACE PROCEDURE TEST_POZ(  
    PR_A IN NUMBER,  
    PR_B IN NUMBER,  
    PR_C IN VARCHAR2,  
    PR_D IN VARCHAR2  
)  
IS  
  
BEGIN  
  
NULL;  
  
END TEST_POZ;  
/
```

Ничего особенного она проделывать не будет, но зато с явным энтузиазмом будет принимать аж четыре параметра! Компилируем:

```
SQL> CREATE OR REPLACE PROCEDURE TEST_POZ(  
2  PR_A IN NUMBER,  
3  PR_B IN NUMBER,  
4  PR_C IN VARCHAR2,  
5  PR_D IN VARCHAR2  
6  )  
7  IS  
8  
9  BEGIN  
10  
11  NULL;  
12  
13  END TEST_POZ;  
14  /
```

Процедура создана.

Все прошло успешно, вот и славно! А вот теперь запишем такой анонимный блок:

```
SET SERVEROUTPUT ON  
  
DECLARE  
  
PR_1 NUMBER;  
PR_2 NUMBER;  
PR_3 VARCHAR2(100);  
PR_4 VARCHAR2(100);
```

```
BEGIN

TEST_POZ(PR_1, PR_2, PR_3, PR_4);

END;
/
```

Запускаем и получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3 PR_1 NUMBER;
4 PR_2 NUMBER;
5 PR_3 VARCHAR2(100);
6 PR_4 VARCHAR2(100);
7
8 BEGIN
9
10 TEST_POZ(PR_1, PR_2, PR_3, PR_4);
11
12 END;
13 /
```

Процедура PL/SQL успешно завершена.

Смотрите, мы объявили четыре параметра и передали их нашей функции, в данном конкретном случае мы применили так называемое - "позиционное представление" (**positional notation**)! Такой тип передачи параметров применяется во всех языках программирования, например в таком как **C** и **C++**! Я сразу рекомендую вам пользоваться именно таким способом передачи! Хотя это еще далеко не все!

Запишем следующий анонимный блок:

```
SET SERVEROUTPUT ON

DECLARE

PR_1 NUMBER;
PR_2 NUMBER;
PR_3 VARCHAR2(100);
PR_4 VARCHAR2(100);

BEGIN

TEST_POZ(PR_A => PR_1, PR_B => PR_2, PR_C => PR_3, PR_D => PR_4);

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3 PR_1 NUMBER;
  4 PR_2 NUMBER;
  5 PR_3 VARCHAR2(100);
  6 PR_4 VARCHAR2(100);
  7
  8 BEGIN
  9
 10 TEST_POZ(PR_A => PR_1, PR_B => PR_2, PR_C => PR_3, PR_D => PR_4);
 11
 12 END;
 13 /
```

Процедура PL/SQL успешно завершена.

В данном случае я использовал - "именное представление" (**named notation**), которое **PL/SQL** унаследовал от языка **Ada**. В данном случае указываются как формальные, так и фактические параметры. Идем далее. Запишем следующий анонимный блок:

```
SET SERVEROUTPUT ON

DECLARE

PR_1 NUMBER;
PR_2 NUMBER;
PR_3 VARCHAR2(100);
PR_4 VARCHAR2(100);

BEGIN

TEST_POZ(PR_B => PR_2, PR_C => PR_3, PR_D => PR_4, PR_A => PR_1);

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
  2
  3 PR_1 NUMBER;
  4 PR_2 NUMBER;
  5 PR_3 VARCHAR2(100);
  6 PR_4 VARCHAR2(100);
  7
  8 BEGIN
  9
 10 TEST_POZ(PR_B => PR_2, PR_C => PR_3, PR_D => PR_4, PR_A => PR_1);
```

```
11
12 END;
13 /
```

Процедура PL/SQL успешно завершена.

В этом примере хорошо видно, что именованное представление позволяет изменить порядок следования параметров и вызывать их, так как вам того хотелось бы. Хотя может это не всегда оправдано! :) Далее запишем следующий анонимный блок:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
PR_1 NUMBER;
PR_2 NUMBER;
PR_3 VARCHAR2(100);
PR_4 VARCHAR2(100);
```

```
BEGIN
```

```
TEST_POZ(PR_1, PR_2, PR_C => PR_3, PR_D => PR_4);
```

```
END;
/
```

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> DECLARE
```

```
2
3 PR_1 NUMBER;
4 PR_2 NUMBER;
5 PR_3 VARCHAR2(100);
6 PR_4 VARCHAR2(100);
7
8 BEGIN
9
10 TEST_POZ(PR_1, PR_2, PR_C => PR_3, PR_D => PR_4);
11
12 END;
13 /
```

Процедура PL/SQL успешно завершена.

Здесь хорошо видно, что позиционное и именованное представление можно комбинировать, и использовать совместно. Хотя я думаю, что так легче запутать код, чтобы потом никто нифига не понял! :) Хотя это все оставляется на усмотрение программиста и стиль написания кода! Так же смею заметить, что - чем больше параметров в процедуре, тем сложнее ее вызывать и тем труднее убеждаться в наличии всех требуемых параметров. Если необходимо передать в процедуру или получить из нее достаточно большое число параметров, то рекомендуется определить тип записи, полями которой будут эти параметры. Затем можно использовать единственный параметр имеющий тип записи. В **PL/SQL** - так же не установлено явное ограничение на количество передаваемых в процедуру параметров.

Значение параметров по умолчанию...

Дело в том, что как и все переменные формальные параметры процедуры могут иметь значения по умолчанию. В таком случае значение параметру, имеющему такое определение можно не передавать. Если же фактический параметр все-таки передан, то принимается именно его значение. Итак, значение по умолчанию указывается вот так:

----- имя_парметра [вид] {:= | DEFAULT} исходное_значение -----

Давайте перепишем нашу первую процедуру с параметрами по умолчанию:

```
CREATE OR REPLACE PROCEDURE TEST_POZ(  
  PR_A IN NUMBER,  
  PR_B IN NUMBER,  
  PR_C IN VARCHAR2 := 'HELLO',  
  PR_D IN VARCHAR2 DEFAULT 'WORLD!!!')  
IS  
  
BEGIN  
  
NULL;  
  
END TEST_POZ;  
/
```

Получаем:

```
SQL> CREATE OR REPLACE PROCEDURE TEST_POZ(  
2  PR_A IN NUMBER,  
3  PR_B IN NUMBER,  
4  PR_C IN VARCHAR2 := 'HELLO',  
5  PR_D IN VARCHAR2 DEFAULT 'WORLD!!!')  
6 IS  
7  
8 BEGIN  
9  
10 NULL;  
11  
12 END TEST_POZ;  
13 /
```

Процедура создана.

К слову, используйте параметры по умолчанию в конце списка всех параметров процедуры, при этом будет возможность использовать как именное, так и позиционное представление. Вот и все с параметрами процедур, теперь вам стало понятнее как все это работает в **PL/SQL**! Пробуйте!

Шаг 93 - PL/SQL - Функции

Вот теперь, наконец, давайте рассмотрим такое понятие как функции **PL/SQL**. Раньше в шагах мы с вами рассмотрели так называемые встроенные функции **PL/SQL**. А сейчас мы попробуем сами научиться писать то, что называется функциями. По своей сути функция это то же, что и процедура, она может принимать параметры по всем тем же правилам, что и процедуры, и кроме всего она может возвращать значения! Но не применением **OUT** типа передаваемого параметра, а сама по себе. То есть функция, принимает параметры и возвращает одно(!), значение! В принципе в функции можно применять параметры с типом **OUT** - но это очень плохая идея! Такой метод я использовать не рекомендую! Определение функции таково:

```
----- CREATE [OR REPLACE] FUNCTION - имя_функции -----  
----- (аргумент [IN] [OUT] [IN OUT] тип, ..... ) AS [IS] -----  
----- тело процедуры -----  
----- RETURN (возвращаемое_значение) -----
```

Кое-что вам уже знакомо, за исключением того, что присутствует оператор **RETURN**. Посредством этого оператора функция возвращает значение. Функция, как правило, вызывается внутри какого-либо определения, т.к. вызывать функцию как оператор нет смысла. Но как вы дальше убедитесь, с помощью функции можно делать очень полезные вещи. Итак, давайте напишем функцию преобразования **BOOLEAN** типа в тип **VARCHAR2** - это самая простая задачка во всех учебниках. Итак:

```
CREATE OR REPLACE FUNCTION BOOL_TO_CHAR(INBL IN BOOLEAN) RETURN VARCHAR2  
IS  
  
OUT_ST VARCHAR2(5);  
  
BEGIN  
  
    IF (INBL) THEN  
        OUT_ST := 'TRUE';  
    ELSIF (NOT INBL) THEN  
        OUT_ST := 'FALSE';  
    ELSE  
        OUT_ST := 'NULL';  
    END IF;  
  
    RETURN(OUT_ST);  
  
END BOOL_TO_CHAR;  
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE FUNCTION BOOL_TO_CHAR(INBL IN BOOLEAN) RETURN VARCHAR2  
2 IS  
3  
4 OUT_ST VARCHAR2(5);  
5  
6 BEGIN
```

```
7
8 IF (INBL) THEN
9   OUT_ST := 'TRUE';
10  ELSIF (NOT INBL) THEN
11    OUT_ST := 'FALSE';
12  ELSE
13    OUT_ST := 'NULL';
14  END IF;
15
16  RETURN(OUT_ST);
17
18 END BOOL_TO_CHAR;
19 /
```

Функция создана.

Теперь попробуем применить ее на практике. Запишем такой анонимный блок:

```
SET SERVEROUTPUT ON

DECLARE

BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(BOOL_TO_CHAR(TRUE));
    DBMS_OUTPUT.put_line(BOOL_TO_CHAR(FALSE));
    DBMS_OUTPUT.put_line(BOOL_TO_CHAR(NULL));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3 BEGIN
4
5   DBMS_OUTPUT.enable;
6   DBMS_OUTPUT.put_line(BOOL_TO_CHAR(TRUE));
7   DBMS_OUTPUT.put_line(BOOL_TO_CHAR(FALSE));
8   DBMS_OUTPUT.put_line(BOOL_TO_CHAR(NULL));
9
10  END;
11 /
TRUE
FALSE
NULL
```

Процедура PL/SQL успешно завершена.

Как видите, наша функция **BOOL_TO_CHAR** вызвана внутри определения **DBMS_OUTPUT.put_line(..)**, так обычно и происходит. Хорошо видно, что мы получили строки, передав булевы значения.

Теперь давайте поговорим об операторе **RETURN**. Этот оператор возвращает значение функции, приводя его к типу возвращаемого функцией.

----- RETURN (значение) -----

Где **значение** - это то что и возвращает функция. Здесь скобки "()" - это только стиль при написании функций, для того чтобы было немного понятнее. Операторов **RETURN** в функции может быть несколько, при этом первый из них, завершит ее работу и вернет управление в вызывающую процедуру! Давайте напишем еще одну функцию преобразования **BOOLEAN** в **VARCHAR2**, но при этом используем немного другую логику:

```
CREATE OR REPLACE FUNCTION BOOL_TO_CHARTWO(INBL IN BOOLEAN) RETURN VARCHAR2
IS
BEGIN
    IF (INBL) THEN
        RETURN('TRUE');
    ELSIF (NOT INBL) THEN
        RETURN('FALSE');
    ELSE
        RETURN('NULL');
    END IF;
END BOOL_TO_CHARTWO;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE FUNCTION BOOL_TO_CHARTWO(INBL IN BOOLEAN) RETURN VARCHAR2
2  IS
3
4  BEGIN
5
6  IF (INBL) THEN
7  RETURN('TRUE');
8  ELSIF (NOT INBL) THEN
9  RETURN('FALSE');
10 ELSE
11 RETURN('NULL');
12 END IF;
13
14 END BOOL_TO_CHARTWO;
15 /
```

Функция создана.

Хорошо видно, что мы заменили промежуточную переменную и применили три оператора **RETURN**. В данном случае это будет то же, что и первая функция хоть и немного в другом контексте. Запишем вот такой анонимный блок:

```
SET SERVEROUTPUT ON

DECLARE

BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(TRUE));
    DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(FALSE));
    DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(NULL));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3 BEGIN
4
5 DBMS_OUTPUT.enable;
6 DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(TRUE));
7 DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(FALSE));
8 DBMS_OUTPUT.put_line(BOOL_TO_CHARTWO(NULL));
9
10 END;
11 /
TRUE
FALSE
NULL
```

Процедура PL/SQL успешно завершена.

Что и требовалось доказать! Так же смею заметить, что в **PL/SQL** с успехом можно применять рекурсию. Рекурсивные вызовы иногда делают код меньше, но запутаннее! Приведу один пример расчета факториала числа, это я подглядел у Билла Гейтса в его **MSDN** и переложил на **PL/SQL**, не все же ему таскать у других! :) Итак:

```
CREATE OR REPLACE FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
IS

BEGIN

IF (NUM <=1) THEN
    RETURN (NUM);
ELSE
```

```
    RETURN (NUM * FACTORIAL(NUM-1));

END IF;

END FACTORIAL;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
 2 IS
 3
 4 BEGIN
 5
 6 IF (NUM <=1) THEN
 7   RETURN (NUM);
 8 ELSE
 9   RETURN (NUM * FACTORIAL(NUM-1));
10
11 END IF;
12
13 END FACTORIAL;
14 /
```

Функция создана.

Запишем анонимный блок для трех значений - вот такой:

```
SET SERVEROUTPUT ON

DECLARE

BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(5)));
    DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(7)));
    DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(12)));

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
 2
 3 BEGIN
 4
 5   DBMS_OUTPUT.enable;
 6   DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(5)));
```

```
7  DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(7)));
8  DBMS_OUTPUT.put_line(TO_CHAR(FACTORIAL(12)));
9
10 END;
11 /
120
5040
479001600
```

Процедура PL/SQL успешно завершена.

Ух, ты! Работает! Привет Биллу! Получили три значения факториала чисел 5, 7, 12. Проверьте правильно или нет?

Вот собственно так пишутся функции. Хотите задание? А вот - в **PL/SQL** нет функции сложения и вычитания одного времени суток и другого! Напишите функции, которые, например, складывают и вычитают, скажем, 10:34 и 5:08! Я такое делал. Интересно, что у вас получится? Пробуйте!

Шаг 94 - PL/SQL - Функции и процедуры - размещение

Мы с вами уже многое знаем о процедурах и функциях, вот теперь давайте поговорим о том, где находятся и хранятся откомпилированные процедуры и функции. После того, как команда **CREATE OR REPLACE** создает процедуру или функцию, она сразу сохраняется в БД, в скомпилированной форме, которая называется **р-кодом** (p-code). В **р-коде** содержатся все обработанные ссылки подпрограммы, а исходный текст преобразован, в вид удобный для чтения системой поддержки **PL/SQL**. При вызове хранимой процедуры **р-код** считывается с диска и выполняется. Собственно сам **Р-код** аналогичен объектному коду генерируемому компиляторами языков программирования высокого уровня. В **Р-коде** содержатся обработанные ссылки на объекты (это свойство ранней привязки переменных, о которой мы говорили с вами ранее) по этому выполнение **Р-кода** является сравнительно не дорогой (нересурсоемкой) операцией. Да к слову напомним, что удалить код процедуры или функции из вашей БД (схемы) можно применив, оператор **DROP** - вот таким образом (в шаге 87 мы уже это делали):

```
---- DROP PROCEDURE имя_процедуры -----
```

```
---- DROP FUNCTION имя_функции -----
```

Теперь давайте вспомним каким образом можно получить информацию о наличии процедур и их работоспособности. Самое простое это выполнить такой запрос к системному представлению **USER_OBJECTS** вот так:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
/
```

В моем случае получилось следующее:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS

BOOL_TO_CHAR	FUNCTION	VALID
BOOL_TO_CHARTWO	FUNCTION	VALID
BOYS	TABLE	VALID
CUSTOMERS	TABLE	VALID
FACTORIAL	FUNCTION	VALID
GIRLS	TABLE	VALID
OFFICES	TABLE	VALID
ORDERS	TABLE	VALID
PRODUCTS	TABLE	VALID
PTEST	PROCEDURE	VALID
SALESREPS	TABLE	VALID
SYS_C003505	INDEX	VALID
SYS_C003506	INDEX	VALID
SYS_C003507	INDEX	VALID
SYS_C003511	INDEX	VALID
SYS_C003512	INDEX	VALID

SYS_C003513	INDEX	VALID
SYS_C003515	INDEX	VALID
TESTINOUT	PROCEDURE	VALID
TESTOUT	PROCEDURE	VALID
OBJECT_NAME	OBJECT_TYPE	STATUS
-----	-----	-----
TESTPRG	PROCEDURE	VALID
TESTPRGTWO	PROCEDURE	VALID
TESTPRM	PROCEDURE	VALID
TEST_POZ	PROCEDURE	VALID

24 строк выбрано.

Я включил только три столбца представления, которые дают основную информацию, но если хотите можете использовать все столбцы. Хорошо видно, что у нас с вами все объекты имеют статус **VALID**, то есть исправны. А, вот как, например увидеть текст хранимой процедуры или функции, для этого используйте системное представление **USER_SOURCE**. Давайте к примеру выведем текст функции из прошлого шага - **FACTORIAL**:

```
SELECT * FROM USER_SOURCE
WHERE NAME = 'FACTORIAL'
/
```

Получаем:

```
SQL> SELECT * FROM USER_SOURCE
2 WHERE NAME = 'FACTORIAL'
3 /
```

NAME	TYPE	LINE TEXT
-----	-----	-----
FACTORIAL	FUNCTION	1 FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
FACTORIAL	FUNCTION	2 IS
FACTORIAL	FUNCTION	3
FACTORIAL	FUNCTION	4 BEGIN
FACTORIAL	FUNCTION	5
FACTORIAL	FUNCTION	6 IF (NUM <=1) THEN
FACTORIAL	FUNCTION	7 RETURN (NUM);
FACTORIAL	FUNCTION	8 ELSE
FACTORIAL	FUNCTION	9 RETURN (NUM * FACTORIAL(NUM-1));
FACTORIAL	FUNCTION	10
FACTORIAL	FUNCTION	11 END IF;
FACTORIAL	FUNCTION	12
FACTORIAL	FUNCTION	13 END FACTORIAL;

13 строк выбрано.

Вот и содержимое самой функции! Все можно найти в системных представлениях. А, что если при создании функции или процедуры происходит ошибка компиляции? Давайте рассмотрим такой вариант. Создадим процедуру намеренно с ошибкой:

```
CREATE OR REPLACE PROCEDURE TESTERR(NUM IN NUMBER)
IS
K NUMBER;

BEGIN

K := NUM

END TESTERR;
/
```

Получаем:

```
SQL> CREATE OR REPLACE PROCEDURE TESTERR(NUM IN NUMBER)
2  IS
3
4  K NUMBER;
5
6  BEGIN
7
8  K := NUM
9
10 END TESTERR;
11 /
```

Предупреждение: Процедура создана с ошибками компиляции.

Правильно, в данном случае мы забыли поставить ";" после завершения строки **K := NUM**. Вот теперь давайте дадим такой запрос:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE STATUS = 'INVALID'
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2  FROM USER_OBJECTS
3  WHERE STATUS = 'INVALID'
4  /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TESTERR	PROCEDURE	INVALID

Странно, процедура вроде бы создана, ее **р-код** присутствует, но она не исправна! Попробуем вызвать ее:

```
EXEC TESTERR;
```

Ответ будет таким:

```
SQL> EXEC TESTERR;
BEGIN TESTERR; END;

*
ошибка в строке 1:
ORA-06550: Строка 1, столбец 7:
PLS-00905: неприемлемый объект MILLER.TESTERR
ORA-06550: Строка 1, столбец 7:
PL/SQL: Statement ignored
```

Все верно - процедура с ошибками! Так вот, я морочил вам голову, только по тому, чтобы вы ясно представляли себе куда бежать, если что-то не выходит! А вот теперь повторим компиляцию:

```
.
.
7
8 K := NUM
9
10 END TESTERR;
11 /
```

Предупреждение: Процедура создана с ошибками компиляции.

И дадим такую строку:

```
SHOW ERRORS
```

Получаем:

```
SQL> SHOW ERRORS
Ошибки для PROCEDURE TESTERR:

LINE/COL ERROR
-----
10/1   PLS-00103: Встретился символ "END" в то время как ожидалось одно
      из следующих:
      . ( * @ % & = - + ; < / > at in is mod not rem
      <an exponent (**)> <> or != or ~= >= <= <> and or like
      between ||
      Символ ";" заменен на "END", чтобы можно было продолжать.
```

Вот теперь кое, кое что стало яснее. Ошибка **PLS-00103** означает наличие незавершенного оператора, команда **SHOW ERRORS**, считывает данные из системного представления **USER_ERRORS**, вот его описание:

```
SQL> DESC USER_ERRORS
Имя      Пусто?  Тип
-----
NAME      NOT NULL VARCHAR2(30)
TYPE      VARCHAR2(12)
```

```
SEQUENCE NOT NULL NUMBER
LINE      NOT NULL NUMBER
POSITION  NOT NULL NUMBER
TEXT      NOT NULL VARCHAR2(4000)
```

Теперь давайте дадим вот такой запрос:

```
SELECT NAME, TEXT FROM USER_ERRORS
/
```

Вот и содержимое:

```
SQL> SELECT NAME, TEXT FROM USER_ERRORS
2 /
```

```
NAME      TEXT
-----
```

```
TESTERR    PLS-00103: Встретился символ "END" в то время как ожидалось одно из следующих:
```

```
. ( * @ % & = - + ; < / > at in is mod not rem
<an exponent (**)> <> or != or ~= >= <= <> and or like
between ||
```

Символ ";" заменен на "END", чтобы можно было продолжать.

Вообще это дело вкуса, но лучше использовать **SHOW ERRORS** - так удобнее. И будет ясно видно где ошибка! Давайте удалим нашу "инвалидную" процедуру и вы сможете сами поработать над тем, что я вам излагал! Итак:

```
DROP PROCEDURE TESTERR
/
```

```
SQL> DROP PROCEDURE TESTERR
2 /
```

Процедура удалена.

Пока все, поработайте сами! :)

Шаг 95 - PL/SQL - Хранимые процедуры, объявления, зависимости

Для ясности картины давайте подведем некую черту под, классификацией хранимых и локальных (анонимных или неименованных) процедурах. Скажу сразу, старайтесь как можно больше использовать именно хранимые подпрограммы (процедуры и функции), а анонимные блоки, только в процессе создания и отладки. Вот таблица, для их сравнения:

Хранимые подпрограммы	Локальные подпрограммы
Хранятся в БД в скомпилированном р-коде, при вызове процедуру не нужно компилировать.	Компилируются фрагменты и содержащиеся в них блоки. При повторном выполнении производится новая компиляция.
Могут вызываться из любого блока запущенного на выполнение пользователем который имеет привилегии EXECUTE для данной подпрограммы.	Могут вызываться только из содержащего их блока.
Код подпрограммы хранится отдельно от вызывающего блока, по этому вызывающий блок короче и легче для понимания. Кроме того, при желании с подпрограммой и вызывающим блоком можно работать по отдельности.	Подпрограмма и вызывающий блок находятся в одном месте, что может привести к путанице. Если изменения вносятся в вызывающий блок то подпрограмму необходимо компилировать заново.
Скомпилированный р-код можно закрепить в разделяемом пуле при помощи модульной процедуры DBMS_SHARED_POOL.KEEP . Это приводит к повышению производительности системы в целом.	Непосредственно локальные подпрограммы нельзя закреплять в разделяемом пуле.

Вот собственно кратко различия, между двумя видами программных блоков. А теперь давайте поговорим на тему, зависимости объектов БД. Дело в том, что при компиляции процедуры или функции, как и все объекты **Oracle**, на которые производится ссылки записываются в словарь данных. Возникает так называемая зависимость (**depend**) объектов друг от друга. Давайте рассмотрим это на примере. Пусть, скажем, имеется таблица **TBLA**, (создадим ее):

```
CREATE TABLE TBLA(
  FIELDA VARCHAR2(100),
  FIELDB NUMBER(3,5)
)
```

Получаем:

```
SQL> CREATE TABLE TBLA(
2  FIELDA VARCHAR2(100),
3  FIELDB NUMBER(3,5)
4  )
5  /
```

Таблица создана.

Предположим что, после этого вы создаете процедуру такого вида:

```
CREATE PROCEDURE TEST_DEPEND(PRMA IN TBLA.FIELDATYPE, PRMB IN TBLA.FIELDDBTYPE)
IS
BEGIN
NULL;
END TEST_DEPEND;
/
```

Получаем после компиляции:

```
SQL> CREATE PROCEDURE TEST_DEPEND(PRMA IN TBLA.FIELDATYPE, PRMB IN
TBLA.FIELDDBTYPE)
2 IS
3
4 BEGIN
5
6 NULL;
7
8 END TEST_DEPEND;
9 /
```

Процедура создана.

Посмотрим на состояние объектов:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_DEPEND	PROCEDURE	VALID
TBLA	TABLE	VALID

Все прекрасно, но это только пока. Допустим по каким-либо причинам вы изменили таблицу **TBLA** вот так:

```
ALTER TABLE TBLA ADD FIELDG NUMBER(5,7)
/
```

Получаем в результате:

```
SQL> ALTER TABLE TBLA ADD FIELD NUMBER(5,7)
2 /
```

Таблица изменена.

А вот теперь, если посмотреть на состояние объектов, то мы увидим следующее:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_DEPEND	PROCEDURE	INVALID
TBLA	TABLE	VALID

Ууупс! (Как поет Бритни Спирс) А, процедура то **TEST_DEPEND** стала **INVALID**! Что же делать?! Может вернуть все на круги своя? Пробуем:

```
ALTER TABLE TBLA DROP COLUMN FIELD
/
```

Получаем:

```
SQL> ALTER TABLE TBLA DROP COLUMN FIELD
2 /
```

Таблица изменена.

Снова смотрим состояние объекта:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_DEPEND	PROCEDURE	INVALID
TBLA	TABLE	VALID

Ууупс! (I did it again!) Не помогло! Что же делать? Да все просто! Если изменили связанный объект, нужно перекомпилировать вашу процедуру! Вот так:

```
ALTER PROCEDURE TEST_DEPEND COMPILE  
/
```

Получаем:

```
SQL> ALTER PROCEDURE TEST_DEPEND COMPILE  
2 /
```

Процедура изменена.

А теперь убедимся, что все пришло на круги своя! Даем запрос:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS  
2 FROM USER_OBJECTS  
3 WHERE OBJECT_NAME IN ('TBLA','TEST_DEPEND')  
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_DEPEND	PROCEDURE	VALID
TBLA	TABLE	VALID

Надеюсь, ясно, если один объект БД связан с другим и в какой-то момент при изменении одного из них с помощью оператора **DDL** или как-либо еще - другой объект необходимо переинициализировать! Немного не удобно, зато дешево и сердито! В любом случае учитесь сопровождать свой код, так чтобы не было ошибок! :)

Шаг 96 - PL/SQL - Понятие пакета в языке PL/SQL

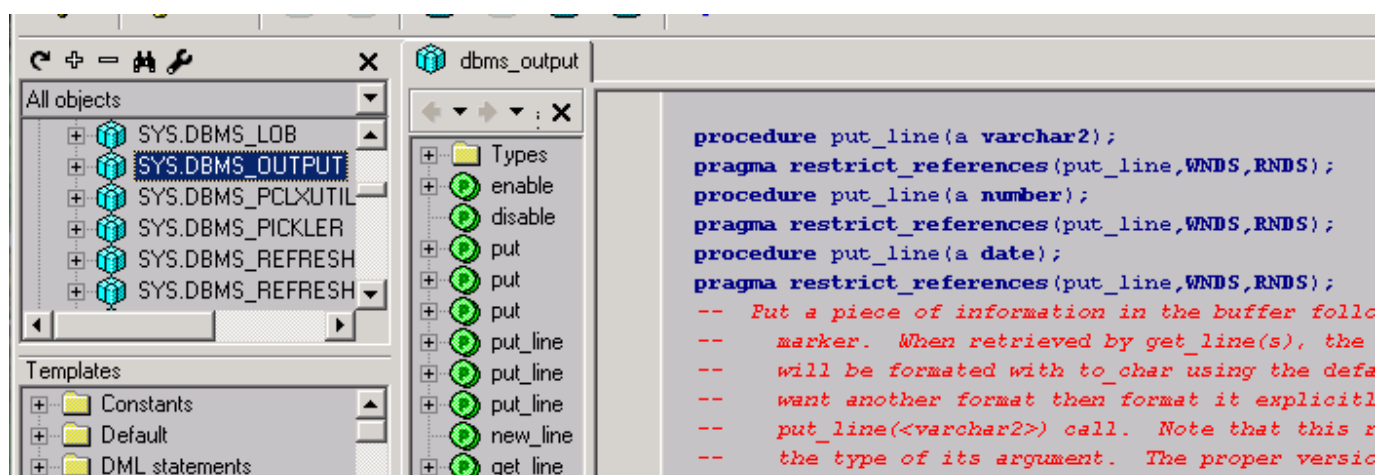
Немаловажным фактором является то, что мы с вами уже научились работать с процедурами **PL/SQL**. Как их создавать, передавать им параметры, изменять, удалять - я думаю это для вас уже не вызывает затруднений. Однако в **PL/SQL** есть еще одно очень, я бы сказал концептуальное и достаточно интересное понятие, а именно пакеты (**package**). Еще их называют модулями. Сама концепция пакета в **PL/SQL** пришла все из того же языка программирования **Ada**. Пакет позволяет в точном определении хранить связанные объекты в одном месте. Что это значит? Допустим, у вас есть группа функций или процедур которая выполняет, скажем, расчет заработной платы. Как правило, это 20-ть - 30-ть функций и процедур. Такое количество хранимых конструкций как правило приводит к тому, что после отладки вашего проекта, Вы уже смутно представляете себе, что делает каждая функция сама по себе, и как говорил персонаж моего любимого мультфильма - "Не понятно, что где валяется и когда все это кончится!" :) Так вот здесь нам на помощь и приходят пакеты (модули кому как больше нравится). Я думаю, вы все помните такую конструкцию:

```

.
.
DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line('HELLO WORLD!');
.
.

```

Так вот это и есть пакет, который занимается тем что выводит "информацию на экран" (в переносном, смысле!). Если вы помните, то ранее я часто говорил - "Воспользуемся пакетом ... бла бла бла", тогда я не акцентировал внимание, а вот сейчас как раз настало время. Дело в том, что при установке сервера **Oracle** - происходит установка очень большого количества пакетов. Для работы с динамическими запросами **DBMS_SQL**, для работы с **Web** пакет **DBMS_WEB** и т.д. Все это огромное количество готового кода, который разработчики фирмы **Oracle** предоставляют в ваше распоряжение. К примеру, если посмотреть "глазами" **PL/SQL - Developer** то, пакет **DBMS_OUTPUT** выглядит в системе примерно вот так:



Слева хорошо виден сам пакет, а справа первая сверху строчка "**procedure put_line(a varchar2);**" и есть как раз функция вывода "на экран". В отличие от процедур и функций, которые могут содержаться локально в блоке или хранится в базе данных, пакеты могут быть

только хранимыми и никогда локальными! К слову при использовании пакетов, производительность системы увеличивается. По своей сути модуль представляет собой именованный раздел объявлений. В него могут входить, различные объявления, как то:

- процедуры
- функции
- типы
- курсоры

После их размещения, на них можно ссылаться из других блоков **PL/SQL**. По этому в них можно описывать глобальные переменные для **PL/SQL**. Давайте рассмотрим как описывается модуль. Каждый модуль состоит из двух частей описания (заголовка) и тела. Заголовок модуля определяется так:

```
CREATE OR REPLACE PACKAGE имя_модуля {IS AS}
описание_процедуры |
описание_функции |
объявление_переменной |
определение_типа |
объявление_исключительной_ситуации |
объявление_курсора |
END [имя_модуля];
```

Тело модуля определяется так:

```
CREATE OR REPLACE PACKAGE BODY имя_модуля {IS AS}
код_инициализации_процедуры |
код_инициализации_функции |
END [имя_модуля];
```

По большому счету тело модуля не является обязательной частью. Если заголовок модуля содержит описание, скажем нескольких переменных, типов и курсоров, то создавать тело модуля нет необходимости. Такой способ целесообразен при объявлении глобальных переменных, поскольку все объекты модуля видимы вне его пределов.

Возьмем на заметку следующее:

- Элементы модуля могут указываться в любом порядке. Однако, как и в разделе объявлений, объект должен быть объявлен до того как на него будут произведены ссылки. Например, если частью условия **WHERE** курсора является некоторая переменная, то она должна быть объявлена до объявления курсора.
- Присутствие элементов всех видов совсем не обязательно! Например, модуль может состоять только из объявлений процедур и функций, без объявления курсоров или типов.
- Объявление процедур и функций должны быть предварительными. В этом отличие модуля от раздела объявлений блока, где могут находиться как предварительные объявления, так и реальный код процедур и функций. Программный код процедур и функций модуля содержится в тексте этого модуля.

В следующий раз поучимся создавать то, что мы с вами рассмотрели.

Шаг 97 - PL/SQL - Учимся создавать пакеты и работать с ними

Теоретически, наверное, почти понятно, что такое пакет (модуль) слово за практическим исполнением. Давайте создадим простой пакет, в котором содержится три функции и одна процедура, так для полноты картины. :) А, затем немного поработает со всем этим. Я создам простые ничего особо не значащие функции. Если у вас возникнет желание усложнить это дело и написать какой-нибудь полезный пакет, я только приветствую! Но, когда вы будете набирать телефон компании **Oracle** в честолобивой попытке продать написанное вами, не забудьте меня при получении гонорара! :) Вот наши функции в чистом виде:

```
-- PROCEDURE Out_Screen -- *****
CREATE OR REPLACE PROCEDURE Out_Screen(TOSC IN VARCHAR2)
IS

BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(TOSC);

END Out_Screen;
/

-- FUNCTION Min_Two_Num -- *****
CREATE OR REPLACE FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

    RETURN (A - B);

END Min_Two_Num;
/

-- FUNCTION Add_Two_Num -- *****
CREATE OR REPLACE FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

    RETURN (A + B);

END Add_Two_Num;
/

-- FUNCTION FACTORIAL -- *****
CREATE OR REPLACE FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
IS

BEGIN

    IF (NUM <=1) THEN
        RETURN (NUM);
```

```
ELSE
RETURN (NUM * FACTORIAL(NUM-1));

END IF;

END FACTORIAL;
/
```

Первая изображает что-то вроде **print**, вторая складывает два числа. Третья отнимает два числа. А четвертая вам уже знакома, та самая рекурсия для расчета факториала числа. Давайте, поместим это все в "пакет" и заставим работать. Создаем описание заголовка пакета **test_pkg**:

```
-- НАШ ПЕРВЫЙ ПАКЕТ -----
CREATE OR REPLACE PACKAGE test_pkg IS

    PROCEDURE Out_Screen(TOSC IN VARCHAR2);

    FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

    FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

    FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER;

END test_pkg;
/
```

Компилируем и ждем результат:

```
SQL> -- НАШ ПЕРВЫЙ ПАКЕТ -----
SQL> CREATE OR REPLACE PACKAGE test_pkg IS
  2
  3  PROCEDURE Out_Screen(TOSC IN VARCHAR2);
  4
  5  FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
  6
  7  FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
  8
  9  FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER;
 10
 11 END test_pkg;
 12 /
```

Пакет создан.

Результат положительный, идем дальше. Создадим само тело нашего пакета:

```
-- PACKAGE BODY test_pkg -- *****
CREATE OR REPLACE PACKAGE BODY test_pkg IS

-- PROCEDURE Out_Screen -- *****
PROCEDURE Out_Screen(TOSC IN VARCHAR2)
IS
```



```
BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(TOSC);

END Out_Screen;

-- FUNCTION Min_Two_Num -- *****
FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

    RETURN (A - B);

END Min_Two_Num;

-- FUNCTION Add_Two_Num -- *****
FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

    RETURN (A + B);

END Add_Two_Num;

-- FUNCTION FACTORIAL -- *****
FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
IS

BEGIN

    IF (NUM <=1) THEN
        RETURN (NUM);
    ELSE
        RETURN (NUM * FACTORIAL(NUM-1));
    END IF;

END FACTORIAL;

END test_pkg;
/
```

Ждем результата компиляции:

```
SQL> -- PACKAGE BODY test_pkg -- *****
SQL> CREATE OR REPLACE PACKAGE BODY test_pkg IS
2
3 -- PROCEDURE Out_Screen -- *****
4 PROCEDURE Out_Screen(TOSC IN VARCHAR2)
```

```
5 IS
6
7 BEGIN
8
9   DBMS_OUTPUT.enable;
10   DBMS_OUTPUT.put_line(TOSC);
11
12 END Out_Screen;
13
14 -- FUNCTION Min_Two_Num -- *****
15 FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
16 IS
17
18 BEGIN
19
20   RETURN (A - B);
21
22 END Min_Two_Num;
23
24 -- FUNCTION Add_Two_Num -- *****
25 FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
26 IS
27
28 BEGIN
29
30   RETURN (A + B);
31
32 END Add_Two_Num;
33
34 -- FUNCTION FACTORIAL -- *****
35 FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER
36 IS
37
38 BEGIN
39
40   IF (NUM <=1) THEN
41     RETURN (NUM);
42   ELSE
43     RETURN (NUM * FACTORIAL(NUM-1));
44   END IF;
45 END IF;
46
47 END FACTORIAL;
48
49 END test_pkg;
50 /
```

Тело пакета создано.

SQL>

Ух ты и снова успешно! Итак, наш с вами пакет создан, он имеет тело и готов к работе. Как уже надеюсь, ясно обратится к объекту в пакете возможно применив точечную нотацию. Вот так:

```
BEGIN
test_pkg.Out_Screen('HELLO!!!');
END;
```

Здесь **test_pkg** - имя нашего пакета, **Out_Screen** имя объекта в пакете и его параметры если таковые есть. Осталось попробовать работу нашего пакета на деле. Запишем такой анонимный блок:

```
SET SERVEROUTPUT ON

DECLARE

BEGIN

test_pkg.Out_Screen(TO_CHAR(test_pkg.Min_Two_Num(10,4)));
test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(5,2)));
test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(test_pkg.FACTORIAL(5),4)));

END;
/
```

После компиляции:

```
SQL> SET SERVEROUTPUT ON
SQL>
SQL> DECLARE
2
3 BEGIN
4
5 test_pkg.Out_Screen(TO_CHAR(test_pkg.Min_Two_Num(10,4)));
6 test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(5,2)));
7 test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(test_pkg.FACTORIAL(5),4)));
8
9 END;
10 /
6
7
124
```

Процедура PL/SQL успешно завершена.

Пакет работает и весьма успешно. Кстати заметили, что код стал более читаемый хотя на первый взгляд не совсем понятно, что как работает. Так, вот пакет это еще одно средство для скрытия кода реализации. То есть функции есть, их параметры есть, а как это работает не видно! Хотя содержимое тела пакета, при необходимости можно увидеть. Вызов процедур и функций может так же иметь именное или позиционное представление параметров. Внутри тела модуля можно ссылаться на объекты указанные в заголовке без указания имени модуля. При первом вызове модуль конкретизируется (**instantiated**) - он считывается с диска в память, а затем запускается его **р-код**. В этот момент для всех переменных, описанных в модуле, выделяется память. Таким образом два сеанса выполняющие подпрограммы одного и того же модуля будут использовать различные области памяти. Вот кратко, что касается создания пакетов. Но это еще не все. :)

Шаг 98 - PL/SQL - Пакеты - Изменение, удаление

Теперь возьмем ситуацию, когда нам нужно добавить в пакет функцию умножающую два числа. Определенную вот так:

```
-- FUNCTION Mul_Two_Num --
CREATE OR REPLACE FUNCTION Mul_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

RETURN (A * B);

END Mul_Two_Num;
```

Как это сделать? Достаточно просто сначала добавить ее описание в заголовке модуля:

```
-- НАШ ПЕРВЫЙ ПАКЕТ --
CREATE OR REPLACE PACKAGE test_pkg IS

    PROCEDURE Out_Screen(TOSC IN VARCHAR2);

    FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

    FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

    FUNCTION Mul_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

    FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER;

END test_pkg;
/
```

Компилируем и ждем результат:

```
SQL> CREATE OR REPLACE PACKAGE test_pkg IS
2
3     PROCEDURE Out_Screen(TOSC IN VARCHAR2);
4
5     FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
6
7     FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
8
9     FUNCTION Mul_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
10
11    FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER;
12
13 END test_pkg;
14 /
```

Пакет создан.

Теперь давайте посмотрим, что же происходит с телом пакета. Дадим вот такой запрос к представлению **USER_OBJECTS**:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_TYPE = 'PACKAGE BODY'
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 WHERE OBJECT_TYPE = 'PACKAGE BODY'
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_PKG	PACKAGE BODY	INVALID

Оп! А, тело теперь недействительно! Исправляем ситуацию:

```
-- PACKAGE BODY test_pkg --
CREATE OR REPLACE PACKAGE BODY test_pkg IS

.
.
.

-- FUNCTION Mul_Two_Num --
FUNCTION Mul_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
IS

BEGIN

RETURN (A * B);

END Mul_Two_Num;

.
.
.

END test_pkg;
/
```

Добавляем функцию **Mul_Two_Num** к телу пакета и компилируем:

```
SQL> -- PACKAGE BODY test_pkg -- *****
SQL> CREATE OR REPLACE PACKAGE BODY test_pkg IS
2
.
.
```

```
.
.

33
34 -- FUNCTION Mul_Two_Num -- *****
35 FUNCTION Mul_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER
36 IS
37
38 BEGIN
39
40 RETURN (A * B);
41
42 END Mul_Two_Num;

.
.
.

58
59 END test_pkg;
60 /
```

Тело пакета создано.

Вот и все. Привожу текст в сокращенном виде дабы не занимать, лишнее место на **Web** - сервере. :) Вы можете проделать то же используя текст предыдущего шага. Главное не забывайте при изменении заголовка, если есть тело пакета хотя бы просто перекомпилировать его. В том случае, если вы добавили в заголовок не процедуру или функцию, а скажем переменную или курсор. Любое изменение заголовка делает тело пакет недействительным (**invalid**)! Просто перекомпилировать пакет можно так:

```
ALTER PACKAGE test_pkg COMPILE
/
```

Получим:

```
SQL> ALTER PACKAGE test_pkg COMPILE
2 /
```

Пакет изменен.

Посмотрим еще раз на состояние тела пакета:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_TYPE = 'PACKAGE BODY'
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
```

```
3 WHERE OBJECT_TYPE = 'PACKAGE BODY'
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TEST_PKG	PACKAGE BODY	VALID

Вот собственно все и получилось! Попробуем, как себя чувствует новоиспеченная функция в пакете:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
BEGIN
```

```
test_pkg.Out_Screen(TO_CHAR(test_pkg.Min_Two_Num(10,4)));
test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(5,2)));
test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(test_pkg.FACTORIAL(5),4)));
test_pkg.Out_Screen(TO_CHAR(test_pkg.Mul_Two_Num(2,2)));
```

```
END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> DECLARE
```

```
2
```

```
3 BEGIN
```

```
4
```

```
5 test_pkg.Out_Screen(TO_CHAR(test_pkg.Min_Two_Num(10,4)));
```

```
6 test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(5,2)));
```

```
7 test_pkg.Out_Screen(TO_CHAR(test_pkg.Add_Two_Num(test_pkg.FACTORIAL(5),4)));
```

```
8 test_pkg.Out_Screen(TO_CHAR(test_pkg.Mul_Two_Num(2,2)));
```

```
9
```

```
10 END;
```

```
11 /
```

```
6
```

```
7
```

```
124
```

```
4
```

Процедура PL/SQL успешно завершена.

Как в школьном курсе математики $2 * 2$ всегда равно 4! (А не три и не пять это надо знать!) :) Теперь давайте рассмотрим, как же можно удалить пакет. Достаточно просто. Применить оператор **DDL** - **DROP**. Вот так:

```
DROP PACKAGE test_pkg
/
```

Получаем:

```
SQL> DROP PACKAGE test_pkg  
2 /
```

Пакет удален.

При этом удаляется весь пакет! Больше ничего определять не нужно. Давайте убедимся в этом:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS  
FROM USER_OBJECTS  
WHERE OBJECT_TYPE IN ('PACKAGE', 'PACKAGE BODY')  
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS  
2 FROM USER_OBJECTS  
3 WHERE OBJECT_TYPE = 'PACKAGE BODY'  
4 /
```

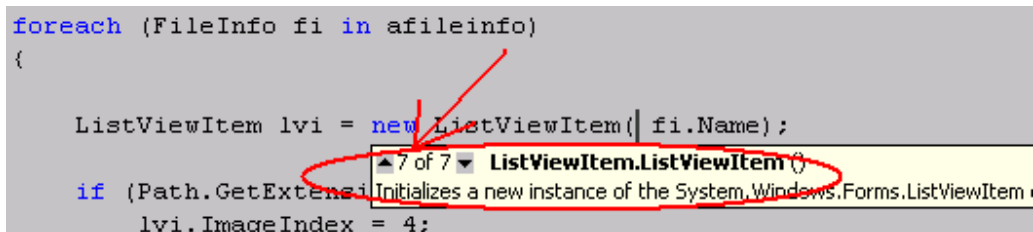
OBJECT_NAME	OBJECT_TYPE	STATUS

0 строк выбрано!

То есть делали мы делали и все бесполезно! Но, думаю только не для вас! А, это самое главное. Прodelайте все еще раз и закрепите полученные навыки. :)

Шаг 99 - PL/SQL - Пакеты - Переопределение, функций и процедур

Думаю, что наверняка кто-то из вас знаком с языками высокого уровня и с принципами объектно ориентированного программирования. Так вот. При определении пакетов можно применять очень интересный фокус под названием "переопределение функций и процедур" (**overloaded**). Например в **C++** или **C#**, на котором я сейчас в основном пишу клиентские части для **Oracle** сервера, есть понятие "перегруженные функции". Когда в среде **Visual Studio.NET**, вы открываете скобку после определяющей сигнатуры функции то видите, то что изображено на рисунке:



```
foreach (FileInfo fi in afileinfo)
{
    ListViewItem lvi = new ListViewItem( fi.Name);
    if (Path.GetExtension(fi.Name) == ".txt")
    {
        lvi.ImageIndex = 4;
    }
}
```

The screenshot shows a dropdown menu for the `new ListViewItem()` call. The dropdown displays "7 of 7" and "ListViewItem.ListViewItem()". A red circle highlights the dropdown, and a red arrow points to it.

Цифра 7 означает, что функция имеет семь перегруженных сигнатур. При выборе конкретной из них, срабатывает именно та, чья сигнатура определена в данный момент. Процедуры и функции внутри модуля (пакета) так же могут быть переопределены. То есть можно определить несколько процедур или функций с одним и тем же именем, но с разными параметрами. В **PL/SQL** - есть смысл применять данное свойство пакета, применительно к обработке объектов различных типов. Здесь как раз и просматривается, некое подобие или попытка привести **PL/SQL** - к объектно-ориентированной модели, применяя основные принципы ООП. Давайте практически попробуем это реализовать. Запишем такой заголовок пакета:

```
CREATE OR REPLACE PACKAGE TST_OVERLOAD IS
```

```
    PROCEDURE Out_Screen(TOSC IN VARCHAR2);
```

```
    FUNCTION Add_One_Num(NM IN NUMBER, BM IN NUMBER) RETURN NUMBER;
```

```
    FUNCTION Add_One_Num(A IN INTEGER) RETURN NUMBER;
```

```
    FUNCTION Add_One_Num(M IN NUMBER, K IN VARCHAR2) RETURN VARCHAR;
```

```
END TST_OVERLOAD;
```

```
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE PACKAGE TST_OVERLOAD IS
```

```
2
```

```
3   PROCEDURE Out_Screen(TOSC IN VARCHAR2);
```

```
4
```

```
5   FUNCTION Add_One_Num(NM IN NUMBER, BM IN NUMBER) RETURN NUMBER;
```

```
6
```

```
7   FUNCTION Add_One_Num(A IN INTEGER) RETURN NUMBER;
```

```
8
```

```
9   FUNCTION Add_One_Num(M IN NUMBER, K IN VARCHAR2) RETURN VARCHAR;
```

```
10
11 END TST_OVERLOAD;
12 /
```

Пакет создан.

В данном случае функция **Add_One_Num** имеет три типа инициализации. К стати обратите внимание на следующий интересный факт:

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_TYPE = 'PACKAGE'
/
```

Получаем:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
2 FROM USER_OBJECTS
3 WHERE OBJECT_TYPE = 'PACKAGE'
4 /
```

OBJECT_NAME	OBJECT_TYPE	STATUS
TST_OVERLOAD	PACKAGE	VALID

Оп! А поле **STATUS** показывает **VALID**! Но тела то, еще вообще нет! Где логика? Странный факт, но оставим это на совести Лари Элисона! :) Создаем тело пакета, чтобы успокоить совесть! Запишем:

```
-- PACKAGE BODY test_pkg --
CREATE OR REPLACE PACKAGE BODY TST_OVERLOAD IS

-- PROCEDURE Out_Screen --
PROCEDURE Out_Screen(TOSC IN VARCHAR2)
IS

BEGIN

    DBMS_OUTPUT.enable;
    DBMS_OUTPUT.put_line(TOSC);

END Out_Screen;

-- FUNCTION Add_One_Num -- One
FUNCTION Add_One_Num(NM IN NUMBER, BM IN NUMBER) RETURN NUMBER
IS

BEGIN

    RETURN (NM + BM);

END Add_One_Num;
```

```
-- FUNCTION Add_One_Num -- Two
FUNCTION Add_One_Num(A IN INTEGER) RETURN NUMBER
IS

BEGIN

    RETURN (A + 20);

END Add_One_Num;

-- FUNCTION Add_One_Num --
FUNCTION Add_One_Num(M IN NUMBER, K IN VARCHAR2) RETURN VARCHAR
IS

BEGIN

    RETURN (TO_CHAR(M + TO_NUMBER(K)));

END Add_One_Num;

END TST_OVERLOAD;
/
```

После компиляции получаем:

```
SQL> CREATE OR REPLACE PACKAGE BODY TST_OVERLOAD IS
  2
  3 -- PROCEDURE Out_Screen --
  4 PROCEDURE Out_Screen(TOSC IN VARCHAR2)
  5 IS
  6
  7 BEGIN
  8
  9   DBMS_OUTPUT.enable;
 10   DBMS_OUTPUT.put_line(TOSC);
 11
 12 END Out_Screen;
 13
 14 -- FUNCTION Add_One_Num -- One
 15 FUNCTION Add_One_Num(NM IN NUMBER, BM IN NUMBER) RETURN NUMBER
 16 IS
 17
 18 BEGIN
 19
 20   RETURN (NM + BM);
 21
 22 END Add_One_Num;
 23
 24 -- FUNCTION Add_One_Num -- Two
 25 FUNCTION Add_One_Num(A IN INTEGER) RETURN NUMBER
 26 IS
 27
```

```
28 BEGIN
29
30 RETURN (A + 20);
31
32 END Add_One_Num;
33
34 -- FUNCTION Add_One_Num --
35 FUNCTION Add_One_Num(M IN NUMBER, K IN VARCHAR2) RETURN VARCHAR
36 IS
37
38 BEGIN
39
40 RETURN (TO_CHAR(M + TO_NUMBER(K)));
41
42 END Add_One_Num;
43
44 END TST_OVERLOAD;
45 /
```

Тело пакета создано.

Чему мы и радуемся! Наконец получили пакет с переопределенными функциями. Теперь можно проверить как ведет себя наша немного комическая функция **Add_One_Num!** :) Хотя это собственно не важно, главное чтобы вы хорошо усвоили, как это все работает. Итак, запишем анонимный блок:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
BEGIN
```

```
TST_OVERLOAD.Out_Screen(TO_CHAR(
TST_OVERLOAD.Add_One_Num(TST_OVERLOAD.Add_One_Num(6),
TO_CHAR(TST_OVERLOAD.Add_One_Num(4, '7'))));
TST_OVERLOAD.Out_Screen(TO_CHAR(TST_OVERLOAD.Add_One_Num(2,2)));
```

```
END;
```

```
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
```

```
SQL>
```

```
SQL> DECLARE
```

```
2
```

```
3 BEGIN
```

```
4
```

```
5 TST_OVERLOAD.Out_Screen(TO_CHAR(
TST_OVERLOAD.Add_One_Num(TST_OVERLOAD.Add_One_Num(6),
6     TO_CHAR(TST_OVERLOAD.Add_One_Num(4, '7'))));
7 TST_OVERLOAD.Out_Screen(TO_CHAR(TST_OVERLOAD.Add_One_Num(2,2)));
```

```
8
9 END;
10 /
37
4
```

Процедура PL/SQL успешно завершена.

И снова арифметика за первый класс торжествует 2 плюс 2 равно 4! И с этим не поспорить. Но, имеются некоторые ограничения! Что поделать - такова жизнь!

1. Нельзя переопределить процедуру или функцию, если их сигнатуры отличаются только именами или видами.

Например:

- ```
2. PROCEDURE Add_One_Num(NM IN NUMBER);
3. PROCEDURE Add_One_Num(NM OUT NUMBER);
```

или

```
PROCEDURE Add_One_Num(A IN NUMBER);
PROCEDURE Add_One_Num(B IN NUMBER);
```

4. Нельзя переопределить процедуру или функцию, если их сигнатуры отличаются лишь типами возвращаемых ими данных.

Например:

- ```
5. FUNCTION Add_One_Num RETURN DATE;
6. FUNCTION Add_One_Num RETURN VARCHAR;
```

7. Типы параметров процедур или функций должны принадлежать различным семействам типов. Например, **CHAR** и **VARCHAR** это одно и тоже по этому нельзя переопределять процедуры вот так:

- ```
8. PROCEDURE Add_One_Num(NM IN CHAR);
9. PROCEDURE Add_One_Num(NM IN VARCHAR);
```

Но самое интересное, что на этапе компиляции вы не получите сообщений об ошибке, а вот при вызове такой функции возникнет ошибка **PLS-307: too many declarations of "имя процедуры функции" match this call** (этому вызову соответствует слишком много объявлений ... бла бла бла ) так что, будьте внимательны, переопределяя функции и процедуры в пакетах. :+)

## Шаг 100 - PL/SQL - Уровни строгости - Прагма RESTRICT\_REFERENCES

Думаю, многие из вас помнят или, по крайней мере, работали с такой конструкцией:

```

.
.
SELECT TO_CHAR(FIELD1), SUBSTR(FIELD2,4,5) FROM
.
.

```

Здесь хорошо видно применение встроенных функций и процедур в операторе **SQL**. Вызовы этих функций по своей сути процедурны, по этому ранее такие вольности не допускались. Но в **PL/SQL** версии 2.1 и выше, именно для хранимых функций такие ограничения отменены. Что собственно очень удобно само по себе. Если обычная или модульная функция отвечает определенным требованиям, то ее можно вызывать во время выполнения **SQL**-оператора. Если функция создана вами, ее тоже можно вызвать и как встроенную функцию из **SQL**-оператора. Но при этом она должна отвечать определенным требованиям. Эти требования определяются в терминах, так называемых уровней строгости. Существует четыре различных уровня строгости. Уровень строгости (**purity level**), определяет структуры данных, которые может считывать или модифицировать функция. Они имеют следующие определения:

1. Любая функция, вызываемая из **SQL**-оператора, не может модифицировать таблицы базы данных (**WNDS**).
2. Для того, чтобы функция могла быть выполнена удаленно (через связь базы данных) или параллельно, она не должна читать или записывать значения модульных переменных (**RNPS** или **WNPS**).
3. Функция, вызываемая из команды **SELECT**, **VALUES** или **SET**, могут записывать модульные переменные. Во всех других командах, функции должны иметь уровень строгости **WNPS**.
4. Функция строга настолько, насколько строги вызываемые ею подпрограммы. Если функция вызывает хранимую процедуру, которая выполняет к примеру обновление информации (оператор **UPDATE**), то функция не имеет уровня строгости **WNDS** и следовательно не может быть использована в **SQL**-операторе.
5. Независимо от уровня строгости, хранимые функции **PL/SQL** - нельзя использовать в ограничении **CHECK** команды **CREATE TABLE** или **ALTER TABLE**. А так же использовать для указания значения по умолчанию, для столбца, так как в этих ситуациях требуется, чтобы описания не изменялись.

То есть, если сказать более просто, не применяйте вызовы тех функций, которые модифицируют таблицы! Давайте опишем уровни строгости функций вот такой табличкой:

| Уровень строгости | Значение                                                              | Описание                                                                 |
|-------------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------|
| WNDS              | Write no database state (не записывать состояния базы данных)         | Функция не модифицирует таблицы базы данных. (При помощи операторов DML) |
| RNDS              | Read no database state (не читать состояния базы данных)              | Функция не читает таблицы базы данных. (При помощи оператора SELECT)     |
| WNPS              | Write no package state (не записывать состояния модульных переменных) | Функция не модифицирует модульные переменные                             |

|      |                                                    |                                                                                                                                                             |
|------|----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|      | записывать состояния модуля)                       | (Модульные переменные не используются в левой части операции присваивания и в операторе FETCH)                                                              |
| RNPS | Read no package state (не читать состояние модуля) | Функция не анализирует модульные переменные (Модульные переменные не используются в правой части операции присваивания и в процедурных или SQL - выражении) |

Кроме тех ограничений, которые мы с вами рассмотрели, функция созданная пользователем, то есть вами, должна отвечать так же дополнительным требованиям, чтобы ее можно было вызвать из **SQL**-операторов. Кстати все встроенные функции тоже отвечают этим требованиям. А вот собственно и требования:

1. Функция должна храниться в БД или отдельно или быть частью модуля. Она не должна быть локальной по отношению к другому блоку.
2. Функция должна иметь входные параметры только с типом **IN**! Но не **IN OUT** или **OUT**!
3. Для формальных параметров должны использоваться только те типы, которые применяются в БД, но не типы **PL/SQL**, такие как **BOOLEAN** или **RECORD**! Типы БД это - **NUMBER**, **CHAR**, **VARCHAR2**, **ROWID**, **LONG**, **LONG ROW** и **DATE**.
4. Тип возвращаемый функцией так же должен быть типом БД!

Рассмотрим пример для всего вышеизложенного. Напишем простую функцию, которая по ключевому полю конкатенирует строки из таблицы **SALESREPS**. А затем выполним **SQL**-оператор, применив нашу функцию.

Записываем:

```
CREATE OR REPLACE FUNCTION FullData(EMPL IN SALESREPS.EMPL_NUM%TYPE) RETURN
VARCHAR2
IS
 i_REZ VARCHAR2(120);
BEGIN
 SELECT NAME || ' ' || TITLE INTO i_REZ
 FROM SALESREPS WHERE EMPL_NUM = EMPL;
 RETURN i_REZ;
END FullData;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE FUNCTION FullData(EMPL IN SALESREPS.EMPL_NUM%TYPE) RETURN
VARCHAR2
2 IS
3
4 i_REZ VARCHAR2(120);
5
6 BEGIN
```

```

7
8 SELECT NAME || ' ' || TITLE INTO i_REZ
9 FROM SALESREPS WHERE EMPL_NUM = EMPL;
10
11 RETURN i_REZ;
12
13 END FullData;
14 /

```

Функция создана.

А теперь **SQL** - оператор:

```

SELECT EMPL_NUM, FullData(EMPL_NUM) "Full Data" FROM SALESREPS
/

```

```

SQL> SELECT EMPL_NUM, FullData(EMPL_NUM) "Full Data" FROM SALESREPS
2 /

```

EMPL\_NUM Full Data

```

105 Вася Пупкин Рапорт продажа
109 Маша Распутина Рапорт продажа
102 Филип Киркоров Рапорт продажа
106 Света Разина Круто втюхал
104 Наташа Королева Рапорт пусто
101 Игорь Николаев Рапорт продажа
110 Крис Кельми Рапорт продажа
108 Игорь Петров Рапорт продажа
103 Дима Маликов Рапорт продано
107 Маша Сидорова Продано все
120 Максим Галкин Продано все

```

11 строк выбрано.

Как видите наш **SQL**- оператор сработал без замечаний! Так как все, что мы делали соответствует **purity level**! А вот для модульных функций дело обстоит несколько иначе. Для того, чтобы в модулях определять уровни строгости необходима прагма - **RESTRICT\_REFERENCES** (ограничить ссылки). Эта прагма устанавливает уровень строгости для конкретной функции. Записывается она следующим образом:

```

----- PRAGMA RESTRICT_REFERENCES (имя_функции, WNDS, [WNPS], [RNDS], [RNPS]) -----

```

Например заголовок нашего модуля из [шага 97](#) мог быть записан вот так:

```

CREATE OR REPLACE PACKAGE test_pkg IS

PROCEDURE Out_Screen(TOSC IN VARCHAR2);

FUNCTION Add_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;
PRAGMA RESTRICT_REFERENCES (Add_Two_Num, WNDS, WNPS, RNPS);

```



```
FUNCTION Min_Two_Num(A IN NUMBER, B IN NUMBER) RETURN NUMBER;

FUNCTION FACTORIAL(NUM IN NUMBER) RETURN NUMBER;
PRAGMA RESTRICT_REFERENCES (FACTORIAL, WNDS, WNPS, RNPS);

END test_pkg;
/
```

Здесь мы применили ее дважды, так как задали прагму для двух функций. В чем собственно необходимость использования прагмы **RESTRICT\_REFERENCES**? Почему прагма применяется в модульной функции и не обязательна для автономной? Ответить на этот вопрос можно проанализировав взаимосвязи, между заголовком и телом модуля. Вспомним, что блоки **PL/SQL** вызывающие модульную функцию зависят только от заголовка модуля, но не от его тела. Более того при создании вызывающего блока тело модуля вообще может отсутствовать. Поэтому компилятору **PL/SQL** необходимо указание помогающее определить уровни строгости модульной функции, чтобы проверить корректность использования функции в вызывающем блоке. При последующих модификациях, тела модуля код функции проверяется на соответствие заданной прагме. Те же правила справедливы и для переопределенных функций. Модули, которые встроены в сам **PL/SQL** не являются строгими. По этому не могут быть включены в **SQL** - оператор! Например **DBMS\_OUTPUT**. И на последок, при вызове функций в **SQL** - операторах, формальные параметры могут иметь значение по умолчанию. Но применять позиционное представление запрещается! Вот собственно, этот раздел и замыкает тематику работы с пакетами.

## Шаг 101 - Oracle - Создание схемы БД - Назначение прав

Думаю, настало время разобраться с тем как же создаются "схемы" в БД **Oracle**. Под понятием схема имеется ввиду сам созданный аккаунт или говоря прямо - пользователь БД! Целью создания пользователя как раз и является получение схемы БД, с определенными правами и привилегиями. Создание пользователя в БД **Oracle** достаточно не сложный, но в тоже время довольно концептуальный момент. Для создания пользователя (схемы), применяется команда **DDL - CREATE USER**. Она имеет следующий синтаксис:

```
----- CREATE USER [пользователь] IDENTIFIED BY [пароль] -----
----- DEFAULT TABLESPACE [tablespace] QUOTA целое число [K][M] ON [tablespace] -----
----- TEMPORARY TABLESPACE [tablespace] QUOTA целое число [K][M] ON [tablespace] --
```

Этот синтаксис не совсем полный, но для начала нам достаточно, далее мы рассмотрим то, что было не указано. Как правило, создание пользователей (схем) БД производится при подключении к БД, пользователем **SYS** или **SYSTEM**. Так как эти две схемы имеют права администраторов БД. Рассмотрим основные параметры команды - **CREATE USER**:

- **[пользователь] [Username]** - Имя пользователя (название схемы).
- **[пароль] [Password]** - Пароль для учетной записи.
- **DEFAULT TABLESPACE** - Табличное пространство в котором будут находиться создаваемые в данной схеме объекты. Эта настройка не дает пользователю права создавать объекты - здесь устанавливается только значение по умолчанию.
- **TEMPORARY TABLESPACE** - Табличное пространство, в котором находятся временные сегменты, используемые в процессе сортировки транзакций.
- **QUOTA** - Позволяет пользователю сохранять объекты в указанном табличном пространстве, занимая там место вплоть до определенного в квоте общего размера.

К слову сказать, в чем мы далее и убедимся. Для того, чтобы запросы пользователей могли создавать временные сегменты в табличном пространстве **TEMP**, им не нужны квоты на дисковое пространство. Попробуем создать пользователя! Запускайте **SQL\*Plus** с пользователем **SYS** или **SYSTEM** пароли администраторов смотрите в [ware 5!](#) Из всего выше сказанного, запишем вот такую конструкцию:

```
CREATE USER DUMMY IDENTIFIED BY DUMB
DEFAULT TABLESPACE USERS QUOTA 100M ON USERS
TEMPORARY TABLESPACE TEMP QUOTA 10M ON TEMP
/
```

Здесь мы создаем пользователя (схему) **DUMMY** с паролем **DUMB** и позволяем ему резвится на 100 Мб пространства **USERS** и еще немного выделяем из пространства **TEMP**. Получаем в результате:

```
SQL> CREATE USER DUMMY IDENTIFIED BY DUMB
2 DEFAULT TABLESPACE USERS QUOTA 100M ON USERS
3 TEMPORARY TABLESPACE TEMP QUOTA 10M ON TEMP
4 /
```

Пользователь создан.

Ок! Пользователь (схема) создан. Наверное, можно уже подключиться и начать создавать объекты! Пробуем!

```
CONNECT DUMMY/DUMB@PROBA
```

Именное такой синтаксис подключения можно использовать, он еще называется строка коннекта и расписывается вот так:

```
---- [Username]/[Password]@[Sevice] -----
```

Получаем:

```
SQL> CONNECT DUMMY/DUMB
ERROR:
ORA-01045: user DUMMY lacks CREATE SESSION privilege; logon denied
```

Предупреждение: Вы больше не соединены с ORACLE.

Опа! Не повезло! Создание пользователя - это еще не все! Теперь ему нужно разрешить самое основное - создавать сессию с сервером. Сделать это можно командой **GRANT**. Она достаточно объемная и мы ей займемся чуть позже, а пока восстановим подключение:

```
SQL> CONNECT SYS/MANAGER@PROBA
Соединено.
```

Даем пользователю право создавать сессию с сервером:

```
SQL> GRANT CREATE SESSION TO DUMMY
2 /
```

Привилегии предоставлены.

Пробуем подключиться:

```
SQL> CONNECT DUMMY/DUMB
Соединено.
```

Вот теперь можно немного перевести дух. Итак, мы создали пользователя, определили ему табличные пространства, назначили квоты на них. И даже позволили создавать сессию с сервером. Давайте убедимся, что пользователь создан и чувствует себя нормально. Производим переконнект на админа БД:

```
SQL> CONNECT SYS/MANAGER@PROBA
Соединено.
```

Дадим такой запрос к представлению **DBA\_USERS**:

```
SELECT USERNAME, USER_ID, PASSWORD, ACCOUNT_STATUS, DEFAULT_TABLESPACE,
TEMPORARY_TABLESPACE, PROFILE
FROM DBA_USERS
```

---

```
WHERE USERNAME = 'DUMMY'
/
```

Получаем:

```
SQL> SELECT USERNAME, USER_ID, PASSWORD, ACCOUNT_STATUS, DEFAULT_TABLESPACE,
2 TEMPORARY_TABLESPACE, PROFILE
3 FROM DBA_USERS
4 WHERE USERNAME = 'DUMMY'
5 /
```

| USERNAME | USER_ID | PASSWORD         | ACCOUNT_STATUS | DEFAULT_TABLESPACE | TEMPORARY_TABLESPACE | PROFILE |
|----------|---------|------------------|----------------|--------------------|----------------------|---------|
| DUMMY    | 64      | E888ADB4D5FFE1B2 | OPEN           | USERS              | TEMP                 | DEFAULT |

Кто знаком с криптографией, может на досуге раскусить - **E888ADB4D5FFE1B2** или хотя бы провести аналогию с **DUMB**! Итак, все с нашей схемой в порядке! Осталось только разрешить пользователю создавать объекты БД.

Разрешаем:

```
GRANT CREATE TABLE TO DUMMY
/
GRANT CREATE PROCEDURE TO DUMMY
/
GRANT CREATE TRIGGER TO DUMMY
/
GRANT CREATE VIEW TO DUMMY
/
GRANT CREATE SEQUENCE TO DUMMY
/
```

Получаем:

```
SQL> GRANT CREATE TABLE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT CREATE PROCEDURE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT CREATE TRIGGER TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT CREATE VIEW TO DUMMY
2 /
```

---

Привилегии предоставлены.

```
SQL> GRANT CREATE SEQUENCE TO DUMMY
2 /
```

Привилегии предоставлены.

Да, так как оператор **GRANT** это **DDL**, то **COMMIT** вызывается не явно! В данном случае мы разрешили пользователю, создавать такие основные объекты БД как - **TABLE, PROCEDURE, TRIGGER, VIEW, SEQUENCE**. Для начала этого достаточно. А что делать, если пользователю будет необходимо изменять эти объекты? Тогда нужно добавить еще немного прав, на изменение (**ALTER**) вот так:

```
GRANT ALTER ANY TABLE TO DUMMY
/
GRANT ALTER ANY PROCEDURE TO DUMMY
/
GRANT ALTER ANY TRIGGER TO DUMMY
/
GRANT ALTER PROFILE TO DUMMY
/
```

Получаем:

```
SQL> GRANT ALTER ANY TABLE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT ALTER ANY PROCEDURE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT ALTER ANY TRIGGER TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT ALTER PROFILE TO DUMMY
2 /
```

Привилегии предоставлены.

Вот теперь он может не только создавать эти объекты, но и изменять их! А, что если пользователю необходимо будет удалить какой-либо объект или удалить записи из таблиц? Тогда нужно добавить права на удаление объектов БД вот так:

```
GRANT DELETE ANY TABLE TO DUMMY
/
GRANT DROP ANY TABLE TO DUMMY
```

---

```
/
GRANT DROP ANY PROCEDURE TO DUMMY
/
GRANT DROP ANY TRIGGER TO DUMMY
/
GRANT DROP ANY VIEW TO DUMMY
/
GRANT DROP PROFILE TO DUMMY
/
```

Получаем:

```
SQL> GRANT DELETE ANY TABLE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT DROP ANY TABLE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT DROP ANY PROCEDURE TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT DROP ANY TRIGGER TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT DROP ANY VIEW TO DUMMY
2 /
```

Привилегии предоставлены.

```
SQL> GRANT DROP PROFILE TO DUMMY
2 /
```

Привилегии предоставлены.

Уфф! Ну вот теперь кажется все! Пользователь действительно полноценный и может работать! Помните в [шаге 6](#) мы с вами это уже проделывали, но тогда я не вдавался в подробности, так как было не до того! А, вот теперь давайте разберемся более детально и продолжим далее.

---

## Шаг 102 - Oracle - Пользователь (схема) и ее модификация

Создание пользователя мы с вами освоили. Давайте рассмотрим как уже созданного пользователя можно видоизменять. Допустим, вы хотите поменять ему пароль. Например, при создании БД пользователи **SYS** и **SYSTEM** получают пароли по умолчанию, которые просто необходимо заменить сразу же после создания БД!!! Иначе любой злоумышленник знакомый с БД **Oracle** может нанести вам большой ущерб!!! Для того, чтобы изменить пароль необходимо использовать команду **ALTER** - это так же довольно обширная команда и рассматривать все, что она может мы пока не будем. Сразу стоит отметить, что **Oracle** не допускает пароли типа **123456** или **543kolobok**. Например, попробуйте следующее:

```
ALTER USER DUMMY IDENTIFIED BY 123456
/
```

Получаем:

```
ALTER USER DUMMY IDENTIFIED BY 123456
*
ошибка в строке 1:
ORA-00988: отсутствует или неверен пароль (пароли)
```

Или:

```
ALTER USER DUMMY IDENTIFIED BY 543kolobok
/
```

Получаем:

```
ALTER USER DUMMY IDENTIFIED BY 543kolobok
*
ошибка в строке 1:
ORA-00988: отсутствует или неверен пароль (пароли)
```

То есть пароли в **Oracle** должны начинаться с буквы, а не с цифры! Но в середине или в конце цифры применять можно! Я к стати до сих пор не могу понять, почему они так сделали? Хотя оставим это на усмотрение **Oracle Inc.** :) Давайте изменим пароль для схемы **DUMMY** с **DUMB** на **PIONER4**:

```
ALTER USER DUMMY IDENTIFIED BY PIONER4
/
```

Получаем:

```
SQL> ALTER USER DUMMY IDENTIFIED BY PIONER4
2 /
```

Пользователь изменен.

Вот теперь все верно! Пароль нашего пользователя изменен и мы можем в этом убедиться:

```
CONNECT DUMMY/PIONER4@PROBA
```

Получаем:

```
SQL> CONNECT DUMMY/PIONER4@PROBA
Соединено.
```

Есть контакт! Таким же образом я рекомендую изменить пароли по умолчанию у схем **SYS** и **SYSTEM**. А, пока вернем коннект пользователю **SYSTEM**:

```
SQL> CONNECT SYSTEM/MANAGER@PROBA
Соединено.
```

С помощью команды **ALTER** - можно так же менять квоты пользователя в табличных пространствах. Например, урезать **DUMMY** за плохое поведение квоту на табличном пространстве **USERS** до 50 Мб:

```
ALTER USER DUMMY
QUOTA 50M ON USERS
/
```

Получаем:

```
SQL> ALTER USER DUMMY
2 QUOTA 50M ON USERS
3 /
```

Пользователь изменен.

Теперь **DUMMY** не выйдет за пределы 50 Мб. Чего ему вполне достаточно для нашего примера. Рассмотрим следующее понятие применимое к схеме БД, а именно профиль (**Profile**). С помощью профилей можно ограничить количество ресурсов системы и БД доступных для пользователя, а так же управлять ограничениями налагаемыми паролями. Если пользователю не назначен профиль по умолчанию, то будет использовать профиль **DEFAULT** (почти каламбур!).

Давайте создадим свой профиль и назначим его нашему пользователю:

```
CREATE PROFILE TODUMMY LIMIT
PASSWORD_LIFE_TIME 180;
/
```

Получаем:

```
SQL> CREATE PROFILE TODUMMY LIMIT
2 PASSWORD_LIFE_TIME 180;
```

Профиль создан.

Созданный нами профиль ограничивает срок действия пароля до 180 дней. Давайте, определим этот профиль для пользователя **DUMMY**:

---



```
ALTER USER DUMMY
PROFILE TODUMMY
/
```

Получаем:

```
SQL> ALTER USER DUMMY
2 PROFILE TODUMMY
3 /
```

Пользователь изменен.

Теперь срок действия пароля **PIONER4** в схеме **DUMMY** будет всего 180 дней. А, после этого БД не будет принимать регистрации с данным паролем. При создании профилей используется ряд ограничивающих ресурсов. Чуть позже мы их опишем. Допустим, если у вас есть необходимость заблокировать определенного пользователя системы, это возможно сделать, применив все тот же **ALTER USER**:

```
ALTER USER DUMMY ACCOUNT LOCK
/
```

Получим заблокированного пользователя:

```
SQL> ALTER USER DUMMY ACCOUNT LOCK
2 /
```

Пользователь изменен.

И соответственно разблокировать пользователя:

```
ALTER USER DUMMY ACCOUNT UNLOCK
/
```

Получим заблокированного пользователя:

```
SQL> ALTER USER DUMMY ACCOUNT UNLOCK
2 /
```

Пользователь изменен.

Давайте рассмотрим так же такое понятие неразрывно связанное с пользователями БД, как роль (**role**). По своей сути роль это некая группа, в которой объединяются несколько привилегий. При использовании ролей можно изменять уровни привилегий для нескольких пользователей одновременно, что упрощает процесс администрирования в БД имеющей несколько сот пользователей. Для примера создадим роль **DMROLE**:

```
CREATE ROLE DMROLE
/
```

Получаем:

---

```
SQL> CREATE ROLE DMROLE
2 /
```

Роль создана.

Пока это "пустая" роль. Теперь назначаем ей ряд привилегий с помощью оператора **GRANT**, например **ALTER SESSION**:

```
GRANT ALTER SESSION TO DMROLE
/
```

Получаем:

```
SQL> GRANT ALTER SESSION TO DMROLE
2 /
```

Привилегии предоставлены.

А вот теперь отпишем эту роль для нашего пользователя **DUMMY** конечно же с помощью оператора **GRANT**:

```
GRANT DMROLE TO DUMMY
/
```

Получаем:

```
SQL> GRANT DMROLE TO DUMMY
2 /
```

Привилегии предоставлены.

Вот теперь схема **DUMMY** может использовать привилегию **ALTER SESSION**. И конечно же, самое главное - "Удаление пользователя"! Удалить пользователя и все объекты его схемы достаточно просто:

```
SQL> DROP USER DUMMY CASCADE
2 /
```

Пользователь удален.

Ключевое слово **CASCADE** означает удалить все связанное со схемой (пользователем). Что то, как то грустно все у нас кончается. :) Таким образом, теперь для вас я думаю, стала яснее картина создания схемы (пользователя) и работа с ней. Стоит отметить к слову, что назначение ролей пользователю БД не всегда оправдано, но как я уже сказал значительно облегчает администрирование. Думаю, вы уже заметили, что мы с вами работаем пока только через **SQL\*Plus**. Есть и более продвинутые инструменты для администрирования БД **Oracle**, такое мощное средство как, например **Enterprise Manager**. Но это тема для отдельного рассказа. Да и пока его использование нам не столь необходимо. А вот вам задание создайте собственного пользователя и несколько объектов в нем и расскажите мне, что у вас получилось! Дерзайте! :)

---

## Шаг 103 - Oracle - Оператор GRANT

Не знаю, чем у вас закончилась история с нашим новым пользователем **DUMMY**, а у меня он все же остался. Если кто-то из вас создал своего пользователя, то можете воспользоваться своим. А, вот сейчас давайте поговорим о том, как могут взаимодействовать разные схемы БД. И как это все возможно осуществить. Запускайте **SQL\*Plus** и подключайтесь пользователем **DUMMY** (если вы его все-таки пристрелили, реанимируйте его согласно [шагу 101](#)). А теперь, находясь в схеме **DUMMY** дайте такой запрос:

```
SELECT * FROM SALESREPS
/
```

Получаем:

```
SQL> SELECT * FROM SALESREPS
2 /
SELECT * FROM SALESREPS
 *
ошибка в строке 1:
ORA-00942: таблица или представление пользователя не существует
```

Неудача "ORA-00942: таблица или представление пользователя не существует"! Говорит само за себя. Теперь попробуем:

```
SELECT * FROM MILLER.SALESREPS
/
```

Получаем:

```
SQL> SELECT * FROM MILLER.SALESREPS
2 /
SELECT * FROM MILLER.SALESREPS
 *
ошибка в строке 1:
ORA-01031: привилегий недостаточно
```

В чем же причина? Да просто у пользователя **DUMMY** нет прав производить чтение из таблицы схемы **MILLER**! Как его предоставить? Очень просто. Подключаемся к схеме **MILLER**:

```
SQL> CONNECT MILLER/KOLOBOK@PROBA
Соединено.
```

А теперь записываем следующее:

```
SQL> GRANT SELECT ON SALESREPS TO DUMMY
2 /
```

Привилегии предоставлены.

Меняем подключение на **DUMMY**:

```
SQL> CONNECT DUMMY/DUMB@PROBA
Соединено.
```

Снова повторяем запрос вот так, чтобы было меньше столбцов:

```
SELECT NAME FROM MILLER.SALESREPS
/
```

Получаем в результате:

```
SQL> SELECT NAME FROM MILLER.SALESREPS
2 /
```

```
NAME
```

```

Вася Пупкин
Маша Распутина
Филип Киркоров
Света Разина
Наташа Королева
Игорь Николаев
Крис Кельми
Игорь Петров
Дима Маликов
Маша Сидорова
Максим Галкин
```

11 строк выбрано.

Теперь результат операции **GRANT SELECT ON SALESREPS TO DUMMY** виден на практике. Давайте более подробно рассмотрим операторы **DDL** - **GRANT** (предоставить) и **REVOKE** (отменить). Эти операторы нельзя использовать непосредственно в **PL/SQL**. Они предназначены для возможности выполнения других операторов **SQL**. Например, чтобы выполнить над таблицей **Oracle** некоторую операцию - **INSERT** или **DELETE**, необходимо иметь полномочия предоставляемые оператором **GRANT**. Существуют привилегии двух различных видов: объектные и системные. Объектная привилегия (**object privilege**) разрешает выполнение определенной операции над конкретным объектом (например над таблицей). В то время как системная привилегия (**system privilege**) разрешает выполнение операций над целым классом объектов. Существует множество системных привилегий, соответствующих практически всем возможным операциям **DDL**. Например, системная привилегия **CREATE TABLE**, позволяет ее обладателю создавать таблицы. А, вот системная привилегия **CREATE ANY TABLE** дает возможность создавать таблицы в других схемах. Давайте кратко насколько это, возможно остановимся на операторе **GRANT**. Синтаксис для предоставления пользователям или ролям системных полномочий и ролей:

```

GRANT --- system_privilege --- TO --- user --- WITH ADMIN OPTION -----

 --- role -----
 --- role -----
 --- PUBLIC -----
```

- **system\_privilege** - предоставляемое системное полномочие.
- **role** - предоставляемая роль.

- **TO** - определяет пользователей или роли, которым предоставляются системные полномочия.
- **PUBLIC** - указывает что, системные полномочия определяемые администратором предоставляются всем пользователям.
- **WITH ADMIN OPTION** - позволяет получившему системные полномочия или роль предоставлять их в дальнейшем другими пользователям или ролям. Такое решение в частности включает и возможность изменение или удаления роли.

Давайте посмотрим какие системные полномочия могут предоставляться. Основных операций в языке **DDL** три - это **CREATE, ALTER, DROP**.

Группа **ALTER**:

- **ALTER DATABASE** - Позволяет изменять саму БД.
- **ALTER USER** - Позволяет изменять пользователя и его параметры (пароль, профиль, роль и т.д.)
- **ALTER PROFILE** - Позволяет изменять профили.
- **ALTER TABLESPACE** - Позволяет изменять табличные пространства.

Для любого объекта - **ANY**:

- **ALTER ANY PROCEDURE** - Разрешает изменение любой хранимой функции процедуры или пакета в любой схеме.
- **ALTER ANY ROLE** - Разрешает изменение любой роли БД.
- **ALTER ANY SEQUENCE** - Разрешает изменение любой последовательности в БД.
- **ALTER ANY TABLE** - Разрешает изменение любой таблицы или вида в схеме БД.
- **ALTER ANY TRIGGER** - Позволяет разрешать, запрещать компилировать любой триггер в любой схеме БД.
- **ALTER ANY INDEX** - Разрешает изменение любого индекса в любой схеме.

Группа **CREATE**:

Позволяет создавать в любой схеме соответствующий объект:

```
CREATE ANY PROCEDURE;
CREATE ANY SEQUENCE;
CREATE ANY TABLE;
CREATE ANY TRIGGER;
CREATE ANY VIEW;
CREATE ANY INDEX;
```

Позволяет создавать в конкретной схеме соответствующий объект:

```
CREATE PROCEDURE;
CREATE SEQUENCE;
CREATE TABLE;
CREATE TRIGGER;
CREATE VIEW;
CREATE INDEX;

CREATE SESSION
CREATE ROLE;
```

---

Удаление объектов в любой схеме, а так же очистка таблиц:

```
DELETE ANY TABLE;

DROP ANY PROCEDURE;
DROP ANY SEQUENCE;
DROP ANY TABLE;
DROP ANY TRIGGER;
DROP ANY VIEW;
DROP ANY INDEX;
```

Удаление объектов в схеме:

```
DROP PROCEDURE;
DROP SEQUENCE;
DROP TABLE;
DROP TRIGGER;
DROP VIEW;
DROP INDEX;
```

И еще полезные системные привилегии:

```
EXECUTE ANY PROCEDURE - Выполнить любую процедуру.
GRANT ANY PRIVILEGE;
GRANT ANY ROLE;
INSERT ANY TABLE - Вставка в любую таблицу.
LOCK ANY TABLE;
SELECT ANY TABLE - Чтение любой таблицы.
SELECT ANY SEQUENCE; - Чтение любой последовательности.
```

Вот далеко не полный список системных привилегий, которые предоставляются оператором **GRANT**. Для начала я думаю хватит. А дальше все зависит от вас. Давайте теперь рассмотрим предоставление объектных привилегий. Здесь все выглядит вот так:

```

----- GRANT --- object_privilege --- ON -- schema.object --- TO --- User -----
----- --- ALL -----, ----- --- Role ----->
----- --- PRIVILEGES - -- (COLUMN) ----- --- PUBLIC ---

-----> WITH ADMIN OPTION -----
```

**object\_privilege** - предоставляемая привилегия - одна из:

```
:ALTER
:SELECT
:UPDATE
:DELETE
:INSERT
:EXECUTE (только для процедур функций и пакетов)
:INDEX (только для таблиц)
:REFERENCES (только для таблиц).
```

**COLUMN** - определяет столбец таблицы или вида, на который распространяется предоставляемая привилегия.

**ON** - определяет объект (таблицу, вид, и т.д.)

**TO** - указывает кому предоставляется привилегия.

**WITH ADMIN OPTION** - позволяет имеющему эту привилегию предоставлять их в дальнейшем другими пользователями или ролям.

Как с работать с этим типом мы с вами уже пробовали в начале этого шага! Можете, например добавить еще что-нибудь к вышеизложенному примеру. И наконец, давайте рассмотрим как привилегии изымаются или удаляются. Для этого необходимо применять оператор **REVOKE**. Его синтаксис аналогичен первым двум операторам за небольшим исключением:

```

----- REVOKE --- object_privilege --- ON -- schema.object --- FROM --- User -----
----- --- ALL -----, ----- --- Role ----->
----- --- PRIVILEGES - --- PUBLIC ---

-----> CASCADE CONSTRAINTS -----
```

Например, чтобы изъять привилегию на выборку из таблицы **SALESREPS** для схемы **DUMMY** введите следующее находясь в схеме **MILLER**:

```
REVOKE SELECT ON SALESREPS FROM DUMMY
/
```

Получим примерно следующее:

```
SQL> REVOKE SELECT ON SALESREPS FROM DUMMY
2 /
```

Привилегии изъяты.

Вот таким образом применяя операторы **GRANT** и **REVOKE**, можно строить взаимоотношение схем и строить политику доступа к объектам БД. Попробуйте создать в новом пользователе несколько объектов и разрешить обращаться к ним из схемы **MILLER**. Если что не получится пишите!

## Шаг 104 - Oracle - Таблица - Ключевой блок БД

Ранее мы с вами рассматривали блоки сервера **Oracle** и упоминали такое фундаментальное понятие как - **ТАБЛИЦА**. По своей сути это, пожалуй, основной строительный модуль БД. Он является вместилищем всех данных, которые содержит ваша БД. Что же такое таблица. Давайте разберемся с ней подробнее. Но, для начала немного теории. Я думаю, что это не будет скучно. А, в дальнейшем очень поможет нам разобраться во многих понятиях. Первое и самое основное - что есть База Данных?

Звучит примерно следующим образом: **база данных** - это совокупность данных, организованных с определенной целью.

Организована - означает, что указанная совокупность включает данные, которые сохраняются, имеют определенный формат (отформатированы) к ним может быть обеспечен доступ (доступны) и они могут быть представлены потребителю информации в приемлемом виде (репрезентативны). Естественно БД, должна обеспечивать, целостность данных (**integrity**) т.е. согласованность отдельных фрагментов данных и их корректность. Понятие согласованность (**consistency**) означает, что все порции данных в БД должны быть единообразно смоделированы и включены в систему. (Что, далеко не часто выполняется проектировщиками БД). Многие из этих понятий для баз данных были сформулированы в 1970 году Е.Ф. Коддом (E.F.Codd), в так называемой реляционной модели (**relational model**). В то время было очень много концепций хранения и работы с данными. В основном использовались иерархические (**IMS**) и сетевые (**IDMS**). А, до них, БД строились на базе файлов с последовательным доступом или как их называли "плоских" файлов (**flat files**). Вот тогда и появилась СУРБД **DB2**, сыгравшая свою ключевую роль во всем этом. Давайте присмотримся внимательнее к сути реляционной модели.

Согласно определению, реляционная модель имеет два главных свойства:

1. Базовые порции данных представляют собой - отношение (**relations**).
2. Операции над таблицами затрагивают только отношения - реляционные выражения (**relation closure**).

Вот здесь мы и подходим наверное к самой сути. Что же такое отношения? По своей природе это математическая концепция, описывающая как соотносятся между собой элементы двух множеств. Так вот корни (которые мы иногда теряем :) ) реляционной модели лежат в области математики. Как собственно и все что, хоть как-то связано с программированием и компьютерной тематикой. А вот для нашей предметной области отношение - это не что иное, как таблица с некоторыми специальными свойствами. Ура! Наконец подошли к нашей основной теме! Еще не слишком скучно вдумываться во все, что я излагаю? Думаю, что нет! :) Реляционная модель предусматривает организацию данных исключительно в виде таблиц (и никак иначе, хотя уже существуют, совершенно новые концепции хранения данных)! БД **Oracle** (как впрочем и во многих других) **ТАБЛИЦА** - базовый языковой объект (**lingua franca**). Посмотрим на рисунок.



Table  
(Lingua franca)

| Field | Column |  |  |
|-------|--------|--|--|
| Row 1 |        |  |  |
| Row 2 |        |  |  |
| Row 3 |        |  |  |
| Row 4 |        |  |  |
| Row 5 |        |  |  |
| Row 6 |        |  |  |
| Row 7 |        |  |  |
| Row 8 |        |  |  |

Таблица Базы Данных

Рис. 1

Таблица представляет собой множество (думаю можно сказать двумерное) именованных атрибутов. Основные из которых столбцы (**columns**) и записи (**rows**) или строки. Как правило отдельный столбец (**column**) называют полем (**field**) таблицы. Пересечение строки и столбца как в двухмерной системе координат (**X,Y**), образует ячейку (**cell**) таблицы. Набор допустимых значений столбца - домен (**domain**) характеризуется определенным типом данных, например символьным или целым и т.д. Строки же представляют собой данные. Все это можно описать, например, в языке **C++** или таком новом концепте как **C#**, как двумерные массивы. Столбцы **columns** и записи **rows**, это не что иное как **МАССИВЫ** данных. Если это четко представлять, то трудностей в работе с таблицами не будет! Ладно, еще немного и я скачусь к описанию работы с таблицами в **ADO.NET**, а это уже злостное отвлечение от темы! :) Давайте пока посмотрим на рисунок.

| Nomer | Name    | Famil | Otches.  |
|-------|---------|-------|----------|
| 1     | Иванов  | Иван  | Иванович |
| 2     | Сидоров | Петр  | Петрович |

Стандартная таблица БД

Рис. 2

Здесь представлена классическая, форма организации таблицы БД в 1НФ. Что такое 1НФ я еще расскажу. А, пока давайте рассмотрим требования предъявляемые к таблицам БД реляционной моделью данных:

1. Данные в ячейках таблицы должны быть структурно не делимы! Каждая ячейка может содержать только одну порцию данных. Это свойство часто определяется как принцип информационной неделимости - недопустимо, чтобы в ячейках таблицы содержалось более одной порции данных. Еще это называют информационным кодированием (**information coding**).
2. Данные в одном столбце должны быть одного типа.
3. Каждая строка должна быть уникальной (недопустимо дублирование строк).
4. Строки (записи) должны размещаться в таблице в произвольном порядке.
5. Столбцы должны иметь уникальные имена.

Давайте раскроем некоторые из этих понятий. Начнем с начала. Наиболее частая ошибка, допускаемая при проектировании и наполнении таблиц! Например следующее:

Создают таблицу, в которой присутствует поле(!) ФИО! Т.е. одна ячейка (**cell**) таблицы содержит - фамилию, имя и отчество! Это просто не допустимо с точки зрения правил построения реляционных таблиц. Эти данные должны содержаться как минимум в трех(!) полях.

Далее второй пункт. Так же почти постоянно нарушается. Например записывают в одной ячейке улица дом, квартира! Такие данные как правило содержат как символьные, так и числовые данные! Что, никак не должно смешиваться в одной ячейке таблицы!

Третий пункт, по большому счету, дублирование записей не самая большая беда. Но если постоянно упускать это из виду, то ничего хорошего не будет!

Четвертый и пятый пункты по моему просто очевидны и думаю заострять на них особое внимания не будем. Вот собственно часть из того, что касается таблиц БД. Далее мы с вами продолжим эту тему.

---

## Шаг 105 - Oracle - Реляционная модель, Правила КОДА (E.F. CODD)

Раз уж мы углубились в дебри реляционной модели данных, давайте уже разберемся со всем этим до конца, вы хорошо все это запомните и мы пойдем дальше. Помимо таблиц и их свойств, реляционная модель имеет еще и собственные операции. Не вдаваясь глубоко в формулировки реляционной математики, отметим, что эти операции позволяют выполнять некоторые действия над подмножествами столбцов строк, объединять (сливать) таблицы. Важно отметить, что все эти операции используют таблицы в качестве исходных данных, а что самое интересное - результат выполнения этих операций так же является таблицами. Все это относится, как вы уже думаю догадались, к языку **SQL**. Основными операторами этого языка, обеспечивающими манипулирование данными и доступ к ним, являются **SELECT, INSERT, UPDATE, DELETE**. Вот четыре основных, кирпичика языка **DML**. А, вот для определения данных и структурированного доступа к ним служат базовые операторы языка **DDL** - **CREATE, ALTER** и **DROP**. Если присмотреться, то эти трое уж очень напоминают, тех четверых. С той лишь разницей, что первые манипулируют с содержимым объектов БД, а вторые с собственно с самими объектами. Хотя может быть объединение в одном языке функций определения и управления очень привлекательно. Вместо двух языков, мы получаем один хотя и более сложный. Что в конечном итоге приводит к тому, что и администратор БД и пользователь применяют одно и то же средство! Вот собственно так и определяется реляционная модель БД. В конце 70х - начале 80х, произошло следующее событие. Была предпринята попытка внедрить средства языка **SQL** в не реляционные системы, а затем их попросту назвали реляционными! Стали они от этого реляционными или нет - покажет время. А основным критерием по сей день служат двенадцать правил сформулированных Коддом (за которыми так и закрепилось название - правила Кодда). Давайте я приведу сами эти правила, а вы попытайтесь осмыслить каждое из них и сделать к ним свой комментарий. Хотя это довольно не простая задача! :) Итак двенадцать негрятят тьфу ! Правил Кодда:

1. **ЯВНОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ.** Информация должна быть представлена в виде данных, хранящихся в ячейках.
2. **ГАРАНТИРОВАННЫЙ ДОСТУП К ДАННЫМ.** К каждому элементу данных должен быть обеспечен доступ с помощью комбинации имени таблицы, первичного ключа (вот тут спорный вопрос, но пусть!) строки и имени столбца.
3. **ПОЛНАЯ ОБРАБОТКА НЕОПРЕДЕЛЕННЫХ ЗНАЧЕНИЙ.** Неопределенные значения **NULL** отличные от любого определенного значения, должны поддерживаться для всех типов данных при выполнении любых операций. (Давний спор всего сообщества по поводу тройственной логики!)
4. **ДОСТУП К ОПИСАНИЮ БД В ТЕРМИНАХ РЕЛЯЦИОННОЙ МОДЕЛИ.** Словарь данных БД должен сохраняться в форме таблицы и СУБД должна поддерживать доступ к нему при помощи стандартных, языковых средств доступа к таблицам.
5. **ПОЛНОТА ПОДМНОЖЕСТВА ЯЗЫКА.** Язык управления данными и язык определения данных должны поддерживать все операции доступа и быть единственным средством, такого доступа кроме возможно, операций нижнего уровня.
6. **ВОЗМОЖНОСТЬ ОБНОВЛЕНИЯ ПРЕДСТАВЛЕНИЙ.** Все представления подлежащие обновлению должны быть доступны для этого.
7. **НАЛИЧИЕ ВЫСОКОУРОВНЕВЫХ ОПЕРАЦИЙ УПРАВЛЕНИЯ ДАННЫМИ.** Операции вставки, удаления, обновления должны применяться к таблице в целом.

8. **ФИЗИЧЕСКАЯ НЕЗАВИСИМОСТЬ ДАННЫХ.** Прикладные программы не должны зависеть от используемых способов хранения данных на носителях и методов обращения к ним.

9. **ЛОГИЧЕСКАЯ НЕЗАВИСИМОСТЬ ДАННЫХ.** Прикладные программы не должны зависеть от логических ограничений.

10. **НЕЗАВИСИМОСТЬ КОНТРОЛЯ ЦЕЛОСТНОСТИ.** Все необходимое для поддержания целостности данных, должно храниться в словаре данных.

11. **ДИСТРИБУТИВНАЯ НЕЗАВИСИМОСТЬ.** Реляционная БД должна быть переносимой и способной к распространению.

12. **СОГЛАСОВАНИЕ ЯЗЫКОВЫХ УРОВНЕЙ.** Допускается использование низкоуровневого языка доступа, где элемент доступа запись.

Но, что самое интересное, существует и правило 0 (ноль)! Оно звучит примерно таким образом:

**ДЛЯ ТОГО, ЧТОБЫ СИСТЕМУ МОЖНО БЫЛО КВАЛИФИЦИРОВАТЬ КАК РЕЛЯЦИОННУЮ СУБД, ОНА ДОЛЖНА ИСПОЛЬЗОВАТЬ ДЛЯ УПРАВЛЕНИЯ БАЗОЙ ДАННЫХ ИСКЛЮЧИТЕЛЬНО РЕЛЯЦИОННЫЕ ФУНКЦИИ!**

Вот такие умозаключения. Если есть мнения, то готов их выслушать. Хотя это тема для длительной дискуссии. :)

## Шаг 106 - Oracle - Модели взаимосвязей объектов и проектирование

Думаю любой администратор БД (среди вас есть такие? :) ), не раз за свою практику сталкивался, с моделированием взаимосвязей между объектами. Этот вопрос достаточно широкий и сложный, но интересный! Как правило, первым шагом в проектировании БД является логическое проектирование. Как показывает опыт, недостаточно продуманное логическое проектирование выполненное в спешке приводит к плачевным результатам. В большинстве случаев приходится реконструировать уже работающую систему, что-то в ней изменяя или перестраивая. Но, как правило, полностью избавиться от недостатков не удастся! По этому самое разумное, что следует сделать на стадии проектирования вашей БД - это ОЧЕНЬ ХОРОШО продумать ЛОГИЧЕСКУЮ(!) структуру основных компонентов вашей БД, а уже после этого переходить непосредственно к физическому и программному оформлению БД. Логическое проектирование играет решающую роль в обеспечении целостности данных. Как правило, производительность системы может определяться на стадии физического проектирования. А вот целостность данных практически полностью определяется принципами организации структуры системы, сформированными на стадии логического проектирования! К слову сказать, именно при процессе логического проектирования можно при достаточном продумывании заложить возможность известной свободы в выборе средств физической реализации системы, что позволит в дальнейшем совершенствовать ее в процессе эксплуатации.

В основе при начале логического проектирования лежит - объект (**entity**) - представляющий собой некоторый элемент (сущность). Применительно к конкретной предметной области. Например, это может быть - "товар", если рассматривать самую популярную область применения БД такую как создание складского учета. И теперь термин объект - это некоторое понятие определяющее, например, количество банок пива "Старый мельник" - на складе. Одна банка - это "объект" складского учета и он представлен в базе данных. Взаимосвязь (отношение - **relationship**) между объектами БД - это ассоциативная связь между объектами. Например поставщик **A** - такого-то числа отгрузил 100 кг. муки или покупатель **B** - купил оптом сто ящиков пива "Старый мельник". Здесь возможно построить связь. Атрибуты (**atributes**) - это характеристики объекта. Например, атрибутом объекта банка пива может служить срок хранения, температура хранения, объем, стоимость и т.д. Обычно говорится, что атрибут принимает некоторое конкретное значение из домена (**domen**) атрибута, множества допустимых значений. Вот именно значения этих атрибутов и будут использоваться в дальнейшем в реляционной модели. Эти определения как бы абстрагированы от конкретной предметной области. Вот именно здесь мы подходим к моделированию взаимосвязей БД, которую в 76-х была предложена П.П. Ченом (P.P. Chen). Это так называемая методика диаграмм взаимосвязей между объектами (**Entity - Relationship - Diagram ERD**). Так же она известна под названием семантическая модель данных (**semantic data model**), так как она учитывает семантику (смысл) данных по отношению к той предметной области, в которой будет использоваться проектируемая система. По большому счету методика **ERD** предшествует реляционному моделированию. Так, например после построения диаграмм **ERD** результаты могут быть непосредственно преобразованы в реляционную модель, а последняя в физическую модель.

Диаграмма **ERD** представляет объекты в виде прямоугольников. Наименования атрибутов объекта приведены внутри прямоугольника, а наименование объекта снаружи. Между прямоугольниками проведены стрелки, которые представляют тип взаимосвязи между объектами. Вот наконец и самое интересное! Существует три основных типа взаимосвязи между объектами:

- один к одному.
- один ко многим. (Наиболее часто применяется!)
- многие ко многим.

Давайте рассмотрим их по порядку. Взаимосвязь один-к-одному, отображает такой характер отношений между объектами, когда каждому значению одного объекта соответствует только одно значения другого, и на оборот. Как правило, такой вид связи применяется довольно редко. **ERD** - диаграмма такой связи представлена на рисунке:

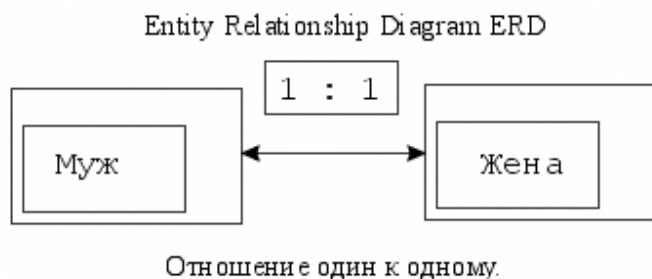


Рис. 1

Ярким примером такой связи может быть муж и жена. Мужу соответствует одна жена, а жене то же один муж. (Варианты полигамии, полиандрии и групповых браков мы не будем рассматривать, в следствии того, что участники таких "семей" - как правило сами не в состоянии определить кто к кому и как относится :))) ) Есть еще один вариант представления взаимосвязи один к одному, такой например, как взаимосвязь подтипов (**subtype**). Такого рода соотношения объектов являются одним из фундаментальных понятий в объектно ориентированном (ОО) анализе и проектировании или моделировании. Вот вам и ООП, по сути дела, это построение иерархии классов или объектов или экземпляров класса-объекта. Кому как больше нравится. Нифига себе до чего добрались, так недалеко и до фундаментальных основ ООП дотопать! А, по чему бы и нет! :) Хватит отвлекаться, смотрим следующий рисунок:



Рис. 2

В данном конкретном случае квадрат является частным случаем семейства прямоугольных! Направление стрелки на линии связи указывает путь наследования (**inheritance**). Немаловажным моментом при использовании связей один-к-одному являются вот такие вопросы:

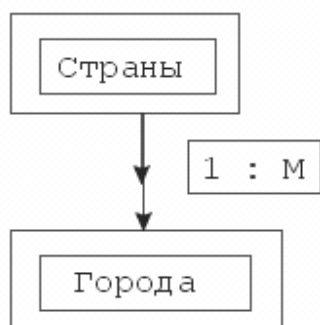
- А, нельзя ли просто объединить эти два объекта?
- Не представляют ли эти объекты в контексте приложения один и тот же объект?

- Существуют ли какие-либо серьезные аргументы в пользу того, чтобы держать эти объекты в системе отдельно?

Вот если два раза нет и один раз нет, тогда все нормально, а если что-то не совпало, то оно того не стоит! :)

Следующим рассмотрим наиболее часто встречающийся тип взаимосвязи один-ко-многим! Смотрим рисунок:

Entity Relationship Diagram ERD



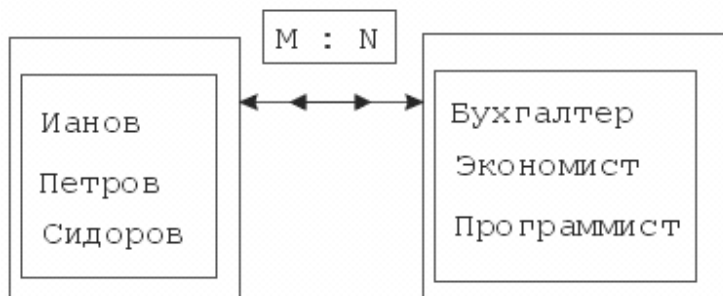
Отношение один ко многим

Рис. 3

Объект "Страна" - связан со множеством объектов "Город". Хотя в некоторых, странах имена городов совпадают, это решается при построении таблицы с составными первичными ключами или специальными идентификаторами городов. Хотя это уже мелочи.

И, наконец, давайте рассмотрим взаимосвязь, многие-ко-многим. Смотрим рисунок:

Entity Relationship Diagram ERD



Отношение многие ко многим.

Рис. 4

Вообще я бы настоятельно рекомендовал Вам при проектировании БД по возможности стараться не применять такой тип связи. И вот почему! Реляционная модель не в состоянии

непосредственно реализовать взаимосвязь "многие-ко-многим"! Задумайтесь над этим! Вследствии этого для обеспечения атомарности данных взаимосвязи типа многие-ко-многим следует заменять несколькими взаимосвязями один-ко-многим. Да, количество объектов БД увеличивается, но правила Кодда соблюдаются! Вот так всегда приходится выбирать из двух зол меньшее и еще не известно, что было лучше! Далее диаграммы **ERD** преобразуются в реляционную модель с помощью так называемых **CASE** систем (**Computer Assisted Software Engineering**) или системы автоматизированного проектирования программного обеспечения. Примером такого ПО может служить **Disigner/2000** от фирмы **Oracle**. Вот собственно кратко, что касается БД и ее проектирования! Но это очень, **ОЧЕНЬ**, важный момент!!!

---



## Шаг 107 - Oracle - Реляционная модель - НОРМАЛИЗАЦИЯ

Дальнейшее совершенствование реляционной модели данных, приводит нас к такому фундаментальному понятию как - НОРМАЛИЗАЦИЯ. Если говорить честно, то когда я пытался это понять с первого раза не получилось! :) Но, я все же попытаюсь, изложить это как можно яснее! Итак - нормализация данных. Как правило, нормализация выполняется после создания приближенной модели (на основе **ERD**) и предназначена для повышения уровня структурной организации данных. В основе нормализации лежит все тот же математический аппарат, базирующийся на концепции функциональной зависимости.

Звучит примерно так: Один столбец или множество столбцов **Y** функционально зависят от одного или множества столбцов **X**, если данное множество значений для **X** определяет единственное множество значений для **Y**. Утверждение "**Y**" функционально зависит от "**X**" равносильно утверждению "**X**" определяет "**Y**", которое записано в форме **X->Y**. Что, поделать, математика и есть математика! Отсюда исходит определение, что основная цель нормализации - избавить реляционную таблицу от зависимостей не связанных с первичными ключами! Если говорить проще, то это приведение к концепту связи типа один-ко-многим, которая является самой фундаментальной в реляционной модели данных и занимает, как правило, 80% всех видов связей в таблицах БД. Что ж, давайте попробуем разобрать это все на практическом примере! Сразу ясно не будет, но я думаю вы справитесь! Итак, возьмем за основу складской учет. Собственно, нам нужно будет взять некие данные и привести их к виду 3НФ (третья нормальная форма данных). Допустим мы имеем накладную на отпущенный товар вот такого вида:

| Накладная № 123     |                 |                                     |             |                 |
|---------------------|-----------------|-------------------------------------|-------------|-----------------|
| Дата                | Покупатель      | Адрес                               |             |                 |
| 10.01.2001          | ООО "Суперлупс" | г. Кукуевск ул. Большая Трубная д.6 |             |                 |
| Отпущен товар       | Количество      | Ед. Изм.                            | Цена за ед. | Общая стоимость |
| Банка стеклянная    | 100             | шт.                                 | 3,45        | 345             |
| Стакан граненый     | 34              | шт.                                 | 1,34        | 45,56           |
| Бутылка "чебурашка" | 367             | шт.                                 | 0,45        | 165,15          |
| Вода минеральная    | 40              | бутылка                             | 7,85        | 314             |
| Водка "Столичная"   | 25              | бутылка                             | 10,50       | 262,5           |
| Пиво "Амур ДВ"      | 40              | банка                               | 4,56        | 182,4           |

Рис. 1

На первом этапе мы приведем ее к 1НФ (первая нормальная форма - к слову скажу на своей практике, я в 90% случаев видел таблицы именно в 1НФ, с чем это связано расскажу ниже!) Определение 1НФ записывается так:

1НФ - требует, чтобы каждое поле таблицы базы данных было не делимым и не содержало повторяющихся групп. Неделимость поля означает, что содержащиеся в нем значения не должны делиться на более мелкие. Повторяющимися считаются поля, которые содержат одинаковые по смыслу значения.

Вернемся к нашей накладной, приводя ее к 1НФ сделаем заметку на то, что далее потребуются, анализ продаж по городам. Поэтому из поля "Адрес" выделим часть данных "Город" в отдельное поле. Учтем, что каждый покупатель может закупить в один день различное количество товаров. Для исключения повторений, фиксируем факт отпуска товара в отдельной записи. В результате получаем нашу накладную в 1НФ:

| ОТПУСК ТОВАРОВ СО СКЛАДА |
|--------------------------|
| Дата                     |
| Покупатель               |
| Город                    |
| Адрес                    |
| Товар                    |
| Ед Измерения             |
| Цена за ед               |
| Отпущено ед              |
| Общая стоимость          |
| Номер накладной          |

Рис. 2

Вторая нормальная форма 2НФ - требует, чтобы все поля таблицы зависели от первичного ключа, т.е. чтобы первичный ключ, однозначно определял запись и не был избыточен.

Идем далее по примеру. Для приведения к 2НФ, выделим поля, которые входят в первичный ключ. Дата и номер накладной не подходят. Более применим первичный ключ "Товар". При этом исходим из того, что по одной накладной может быть отпущено одно наименование конкретного товара. Т.е. нет ситуации когда, один и тот же товар, занимает две строки в накладной, исключая повторение. Получаем четыре поля в составе первичного ключа следующего вида:

| ОТПУСК ТОВАРОВ СО СКЛАДА |
|--------------------------|
| Дата                     |
| Покупатель               |
| Номер накладной          |
| Товар                    |
| Город                    |
| Адрес                    |
| Ед Измерения             |
| Цена за ед               |
| Отпущено ед              |
| Общая стоимость          |

Рис. 3

Такой тип ключа избыточен и не нужен. Поле номер накладной однозначно определяет дату и покупателя. И не может быть никакой иной даты и никакого иного покупателя. Сделаем поле товар и номер накладной как первичный ключ исключив лишнее:

| ОТПУСК ТОВАРОВ СО СКЛАДА |
|--------------------------|
| Номер_накладной          |
| Товар                    |
| Дата                     |
| Покупатель               |
| Город                    |
| Адрес                    |
| Ед_Измерения             |
| Цена за ед               |
| Отпущено_ед              |
| Общая_стоимость          |

Рис. 4

Вот теперь первое требование 2НФ выполнено. Но нет независимости поля от части первичного ключа. Ведь, там два поля! В нашем случае поле "Единица измерения" и "Цена за единицу измерения" зависят от значения поля "Товар", входящего в первичный ключ. Тогда давайте выделим эти поля в самостоятельную таблицу "Товары" и определим связь один-ко-многим! Вот теперь это похоже на 2НФ:

Выделение таблицы "Товар"

| ОТПУСК ТОВАРОВ СО СКЛАДА |
|--------------------------|
| Номер_накладной          |
| Товар (FK)               |
| Дата                     |
| Покупатель               |
| Город                    |
| Адрес                    |
| Отпущено_ед              |
| Общая_стоимость          |

| ТОВАРЫ       |
|--------------|
| Товар        |
| Ед_Измерения |
| Цена за ед   |

2 НФ

Рис. 5

Казалось бы и хватит, но идем дальше! Анализируя структуру нашего склада можно заметить, что значение поля "Покупатель" не зависит от пары "Номер накладной" и "Товар", а зависит только от поля "Номер накладной" по этому выделяем "Город" и "Адрес" в таблицу "Покупатели"! Вот так:

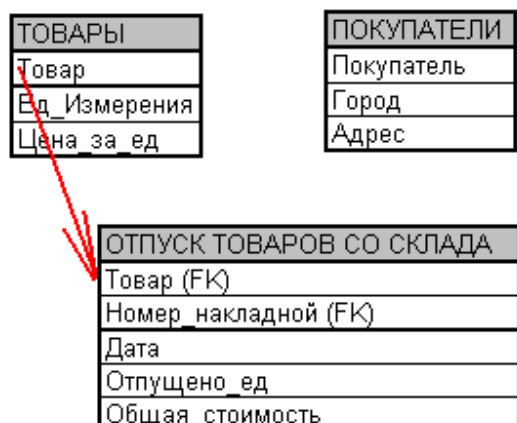


Рис. 6 Выделение таблицы "Покупатели"

Ведем анализ таблицы "Отпуск товаров со склада" далее. Хорошо видно, поле "Дата" зависит только от значения поля "Номер накладной" по этому выдели дату и номер накладной в самостоятельную таблицу "Накладные".

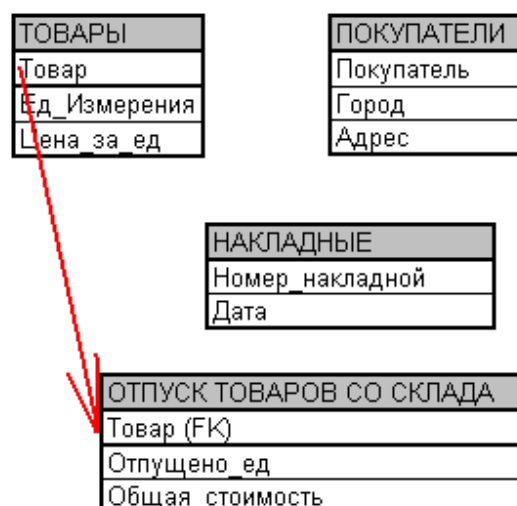


Рис. 7 Выделение таблицы "Накладные"

Определим связи, один покупатель может встречаться во многих накладных, по этому таблицы "Покупатели" и "Накладные" имеют связь один-ко-многим по полю "Покупатель" и соответственно одной накладной может соответствовать несколько товаров. Таблицы "Накладные" и "Отпуск товаров со склада" так же имеют связь один ко многим по полю "Номер накладной". Как это видно из рисунка:

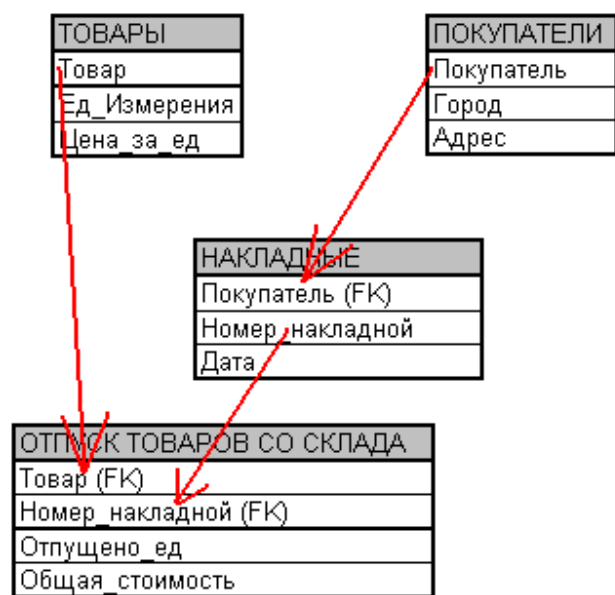


Рис. 8 Связи между таблицами в ЗНФ

Ура! Вот мы, наконец и подходим к триумфу, ой не к ЗНФ! А, она требует чтобы, в таблице не имелось транзитивных зависимостей между не ключевыми полями, т.е. чтобы значение любого поля не входящего в первичный ключ не зависело от значения другого поля, так же не входящего в первичный ключ! Надеюсь теперь хоть что-то понятно? Продолжим наш анализ, из которого так же видно, что таблица "Отпуск товаров со склада" имеет зависимость значения поля "Общая стоимость" от значения поля "Количество". Значение поля "Общая стоимость" может вычисляться как значение поля "Количество" умноженное на значение поля "Цена за единицу" из таблицы "Товары". Исключаем поле "Общая стоимость" из таблицы "Отпуск товаров со склада" вот теперь мы пришли к совершенной ЗНФ! Что, собственно и требовалось получить:



Рис. 9 Нормализованная БД (ЗНФ)

Вот мы и получили нормализованный складской учет! Но! У нормализации есть и свои недостатки! Прежде всего это большее количество сущностей БД. Чем это может грозить? Представьте себе нормализованную БД, масштаба крупного предприятия содержащую, сотни таблиц и тысячи связей между ними! Сопровождение и поддержка такой БД, превращается в достаточно не простую задачу, а если на предприятии текучка кадров программистов, то просто не возможно представить как все это можно осмыслить! Например, известны случаи эволюционного развития систем БД предприятий, функционирование которых впоследствии признавалось вышедшим за границы понимания! (!!!) Представили? Еще одним недостатком, можно определить трудности построения запросов к таким БД, так как необходимо связывать несколько таблиц! А границы понимания не безграничны особенно у человека! Так, что делайте выводы сами! Стоит ли идти по пути нормализации или свалить все в одну максимум две таблицы, что за частую все и делают, зато не нужно долго думать! Но я все же советую использовать хотя бы 2НФ - это не так сложно! Добавлю еще кое что в заключении. По определению, нормальной формы Бойса-Кодда (НФБК) - отсутствие инверсной частичной зависимости или ни первичный ключ, ни какая-либо его часть не должны зависеть от не ключевого атрибута. Эту форму еще шутливо называют **3S** - нормальной формой. В теории нормализации рассматриваются НФ и более высокого порядка 4НФ, 5НФ и т.д. но они представляют собой чисто теоретический интерес и используются крайне редко! 7НФ и 8НФ встречаются только в диссертациях, может кто-то из вас ее напишет! :) Но главной задачей администратора БД, является достичь наибольшего уровня нормализации БД при наименьшем количестве объектов в ней! Вот вам не тривиальная задачка! А продвинуться до уровня 4НФ (**multivalued dependence**), а еще круче 5НФ (**join dependence**), это уже высший пилотаж! Пробуйте! Все в ваших руках!

---

## Шаг 108 - Oracle - ТАБЛИЦЫ БД определение отношений

Надеюсь из прошлых, почти полностью теоретических, изложений стало немного яснее, что же все-таки есть реляционные данные и все, что с ними связано. Давайте сейчас попробуем просмотреть все это практически. Наша с вами учебная БД в схеме **miller** содержит, пять таблиц. Все они в принципе отвечают требованиям ЗНФ. Но, когда я их создавал, я не связал столбцы этих таблиц между собой с помощью стандартных средств. А, вот сейчас давайте мы с вами это сделаем. И так для примера организуем связь, которая чаще всего рекомендована к применению, типа один-ко-многим. Ярким примером для построения такой связи служит две из наших пяти таблиц это **CUSTOMERS** и **SALESREPS**. Оператор **CREATE TABLE** их **DDL** определений записан следующим образом:

Таблица **CUSTOMERS**:

```
CREATE TABLE CUSTOMERS
(
 CUST_NUM INTEGER PRIMARY KEY,
 COMPANY VARCHAR2(30),
 CUST_REP INTEGER,
 CREDIT_LIMIT NUMBER,
)
```

Таблица **SALESREPS**:

```
CREATE TABLE SALESREPS
(
 EMPL_NUM INTEGER PRIMARY KEY,
 NAME VARCHAR2(30),
 AGE INTEGER,
 REP_OFFICE INTEGER,
 TITLE VARCHAR2(20),
 HIRE_DATE DATE NOT NULL,
 MANAGER INTEGER,
 QUOTA NUMBER,
 SALES NUMBER
)
```

При просмотре данных, этих таблиц почти сразу видно, что столбец таблицы **SALESREPS** - **EMPL\_NUM** есть отношение один-ко-многим столбца **CUST\_REP** для таблицы **CUSTOMERS**. Для определения связи между таблицами воспользуемся оператором **ALTER TABLE** и запишем вот такую конструкцию:

```
ALTER TABLE CUSTOMERS ADD
FOREIGN KEY (CUST_REP) REFERENCES SALESREPS (EMPL_NUM)
```

Получаем:

```
SQL> ALTER TABLE CUSTOMERS ADD
```

```
2 FOREIGN KEY (CUST_REP) REFERENCES SALESREPS (EMPL_NUM)
3 /
```

Таблица изменена.

Все, связь между столбцами таблиц установлена! Все достаточно просто. Теперь действует ограничение ссылочной целостности и нарушить его нам с вами не позволят! Можно убедиться в этом. Столбец таблицы **SALESREPS** - **EMPL\_NUM** содержит следующее множество значений 101 .. 110 и отдельно 120. Попробуйте что-нибудь вроде:

```
INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
VALUES(2155, NULL, 150, 34.567)
/
```

После ввода, получаем:

```
SQL> INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
2 VALUES(2155, NULL, 150, 34.567)
3 /
INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
*
ошибка в строке 1:
ORA-02291: нарушено ограничение целостности (MILLER.SYS_C003548) - исходный
ключ не найден
```

Естественно ошибка **ORA-02291**! А все потому, что множество 101 .. 110 и отдельно 120 не содержит числа 150! И по этому в данном случае не допустимо! Вот и получилось жесткое отношение один-ко-многим! Так же, можно и удалить связь, между столбцами таблиц применив оператор **DROP**. Но, нужно узнать имя ссылочной целостности в системе. Сейчас мы его знаем благодаря ошибке. А что если, в процессе работы нужно удалить ссылочную целостность, а потом снова восстановить ее! Для этого обратимся к представлению в вашей схеме **USER\_CONSTRAINTS**. Оно содержит все имена ваших ограничений. Дадим такой запрос:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'CUSTOMERS'
/
```

Получаем:

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
2 WHERE TABLE_NAME = 'CUSTOMERS'
3 /
```

| CONSTRAINT_NAME | CONSTRAINT_TYPE |
|-----------------|-----------------|
| SYS_C003506     | P               |
| SYS_C003548     | R               |

Там где поле **CONSTRAINT\_TYPE** содержит значение **R** и есть наше ограничение (по моему от **REFERENCES**, точно не помню!) Получаем имя ограничения - **SYS\_C003548** (тоже номер был и в ошибке помните?). Вот теперь давайте от него избавимся:



```
ALTER TABLE CUSTOMERS DROP CONSTRAINT SYS_C003548
/
```

Получаем:

```
SQL> ALTER TABLE CUSTOMERS DROP CONSTRAINT SYS_C003548
2 /
```

Таблица изменена.

Вот теперь ограничение снято. Повторим наш предыдущий запрос и посмотрим, что содержит **USER\_CONSTRAINTS** сейчас:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
WHERE TABLE_NAME = 'CUSTOMERS'
/
```

Получаем:

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
2 WHERE TABLE_NAME = 'CUSTOMERS'
3 /
```

| CONSTRAINT_NAME | CONSTRAINT_TYPE |
|-----------------|-----------------|
| SYS_C003506     | P               |

Хорошо видно, что осталось только ограничение первичного ключа таблицы **CUSTOMERS** имеющее имя **SYS\_C003506**. Кроме того, таблица может содержать ограничение на саму себя например все с той же таблицей **SALESREPS** можно проделать следующее:

```
ALTER TABLE SALESREPS ADD
FOREIGN KEY (MANAGER) REFERENCES SALESREPS (EMPL_NUM)
/
```

Получаем:

```
SQL> ALTER TABLE SALESREPS ADD
2 FOREIGN KEY (MANAGER) REFERENCES SALESREPS (EMPL_NUM)
3 /
```

Таблица изменена.

Теперь таблица, как бы это лучше сказать - "самоограничилась", хотя это не всегда оправдано, но вполне применимо и может использоваться! Можете сами с этим всем поработать и определить приоритеты, при проектировании БД, оптимизации и определении ссылочных целостностей таблиц! Но, слишком не увлекайтесь, границы сознания не бесконечны и не стоит выходить за границы понимания, а уж во всяком случае выпускать за них свою БД. :)

## Шаг 109 - Oracle - ТАБЛИЦЫ БД основные операторы определения

Было уже много сказано о таблицах БД. Но, что поделать они самая основная ее часть и разобрать их до конца я думаю стоит. Для начала определимся с операторами создания. Естественно это **CREATE TABLE** и то как он устроен. Если бы я привел вам всю конструкцию **CREATE TABLE**, то это заняло бы шагов так 20-ть или 30-ть! Если у кого есть неподдельный интерес можете заглянуть в **Oracle 8i DBA HandBook**, я кстати с ней не расстаюсь! Там описание **CREATE TABLE** занимает более 10-ти страниц. Но пока остановимся на самом главном! Итак, оператор **CREATE TABLE** и его синтаксис краткий:

```
----- CREATE TABLE -- shema . tablename -----
----- (column - column datatype -- DEFAULT expr -- column constraint) -----
----- table constraint -----
----- TABLESPACE tablespace -----
----- STORAGE storage options -----
```

Где:

- **column datatype** - тип данных столбцов таблицы.
- **DEFAULT expr** - значение по умолчанию.
- **column constraint** - ограничение столбца.
- **table constraint** - ограничение таблицы.
- **TABLESPACE tablespace** - табличное пространство хранения.
- **STORAGE storage options** - параметры хранения таблицы.

Вот таким образом определяется оператор **CREATE TABLE**. Но таблицы так же можно и изменять. С помощью оператора **ALTER TABLE** его краткий синтаксис таков:

```
----- ALTER TABLE -- shema . tablename -----
----- ADD (column datatype -- DEFAULT expr -- column constraint) -----
----- MODIFY (column datatype -- DEFAULT expr -- column constraint) -----
----- DROP COLUMN (column) -----
----- STORAGE storage options -----
```

Где:

- **ADD** - опция добавления элемента таблицы.
  - **column datatype** - тип данных столбцов таблицы.
  - **DEFAULT expr** - значение по умолчанию.
  - **column constraint** - ограничение столбца.
- **MODIFY** - опция изменения элемента таблицы.
  - **column datatype** - тип данных столбцов таблицы.
  - **DEFAULT expr** - значение по умолчанию.
  - **column constraint** - ограничение столбца.
- **DROP COLUMN column** - удаление столбца.
- **STORAGE storage options** - параметры хранения таблицы.

Для получения информации о таблицах, находящихся в БД при их создании и определении возможно с помощью системных представлений таких как:

## 1. ALL\_ALL\_TABLES

Запросим его описание:

```
DESC ALL_ALL_TABLES
```

Получаем:

```
SQL> DESC ALL_ALL_TABLES
Name Type Nullable Default Comments

OWNER VARCHAR2(30) Y Owner of the table
TABLE_NAME VARCHAR2(30) Y Name of the table
TABLESPACE_NAME VARCHAR2(30) Y Name of the tablespace containing the table
CLUSTER_NAME VARCHAR2(30) Y Name of the cluster, if any, to which the table
belongs
IOT_NAME VARCHAR2(30) Y Name of the index-only table, if any, to which the
overflow
 or mapping table entry belongs
PCT_FREE NUMBER Y Minimum percentage of free space in a block
PCT_USED NUMBER Y Minimum percentage of used space in a block
.
.
.
MONITORING VARCHAR2(3) Y Should we keep track of the amount of modification?
CLUSTER_OWNER VARCHAR2(30) Y Owner of the cluster, if any, to which the table
belongs
DEPENDENCIES VARCHAR2(8) Y Should we keep track of row level dependencies?
```

## 2. DBA\_TABLES

Вот его описание:

```
DESC DBA_TABLES
```

Получаем:

```
SQL> DESC DBA_TABLES
Name Type Nullable Default Comments

OWNER VARCHAR2(30) Owner of the table
TABLE_NAME VARCHAR2(30) Name of the table
TABLESPACE_NAME VARCHAR2(30) Y Name of the tablespace containing the table
CLUSTER_NAME VARCHAR2(30) Y Name of the cluster, if any, to which the table
belongs
IOT_NAME VARCHAR2(30) Y Name of the index-only table, if any, to which
the overflow or mapping table entry belongs
PCT_FREE NUMBER Y Minimum percentage of free space in a block
PCT_USED NUMBER Y Minimum percentage of used space in a block
.
.
.
```

|               |                |                                                          |
|---------------|----------------|----------------------------------------------------------|
| CLUSTER_OWNER | VARCHAR2(30) Y | Owner of the cluster, if any, to which the table belongs |
| DEPENDENCIES  | VARCHAR2(8) Y  | Should we keep track of row level dependencies?          |

### 3. USER\_TABLES

Вот его описание:

```
DESC USER_TABLES
```

Получаем:

```
SQL> DESC USER_TABLES
Name Type Nullable Default Comments

TABLE_NAME VARCHAR2(30) Name of the table
TABLESPACE_NAME VARCHAR2(30) Y Name of the tablespace containing the table
CLUSTER_NAME VARCHAR2(30) Y Name of the cluster, if any, to which the table
belongs
IOT_NAME VARCHAR2(30) Y Name of the index-only table, if any, to which the
overflow
or mapping table entry belongs
PCT_FREE NUMBER Y Minimum percentage of free space in a block
PCT_USED NUMBER Y Minimum percentage of used space in a block
.
.
.
MONITORING VARCHAR2(3) Y Should we keep track of the amount of modification?
CLUSTER_OWNER VARCHAR2(30) Y Owner of the cluster, if any, to which the table
belongs
DEPENDENCIES VARCHAR2(8) Y Should we keep track of row level dependencies?
```

Все они очень похожи. Но есть и свои особенности у каждого из них. Вообще работа с системными представлениями, может дать почти полную информацию о вашем экземпляре, о чем я по моему уже не однократно говорил. Так же полезны для работы с таблицами представления **ALL\_COL\_COMMENTS**, **ALL\_COL\_PRIVS**, **ALL\_CALL\_TYPES**, **DBA\_TAB\_PRIVS**, **USER\_TAB\_COLUMNS** и т.д. И последнее удаление таблиц выполняется естественно оператором **DDL - DROP**:

```
----- DROP TABLE -- shema . tablename -----
```

Вот наверное пока все, что касается таблиц и работы с реляционными объектами БД. Но, с таблицами мы пока не расстаемся насовсем, так как еще многое будет связано с ними напрямую. А, вам пока задача: создать собственную реляционную модель и определить данные для построения ЗНФ и внести это все в Ваш сервер **Oracle** в собственную схему, например биллинг сотовой компании. Слабо? :)

## Шаг 110 - PL/SQL - Триггеры БД, теория

Думаю, самое время заняться еще одним типом именованных блоков **PL/SQL**, а именно триггеры таблиц БД. Сам по себе триггер БД, является именованным блоком **PL/SQL** и после компиляции хранится, в соответствующих словарях данных вашей БД. Но он имеет ряд особенностей. Процедуры или функции, которые мы с вами уже разбирали могут вызывать или быть вызваны другими процедурами, при этом им могут быть переданы параметры. Триггер - не может быть вызван из другой процедуры БД и не принимает никаких параметров при вызове. Само название говорит о том, что этот блок **PL/SQL** - срабатывает при определенном событии, а именно при запуске операций **DML** - **INSERT, UPDATE, DELETE**. Существуют так же так называемые системные триггеры, которые срабатывают на события самой БД. Но о них чуть позже. В основном триггеры используются для:

1. Реализации сложных ограничений целостности данных, которые невозможно осуществить через описательные ограничения, устанавливаемые при создании таблиц.
2. Организации всевозможных видов аудита. Например, слежения за изменениями в какой-либо важной таблице БД.
3. Автоматического оповещения других модулей о том, что делать в случае изменения информации содержащейся в таблице БД.
4. Для реализации так называемых "бизнес правил".
5. Для организации каскадных воздействий на таблицы БД.

В принципе по ходу работы можете придумать еще что-нибудь! Вообще триггеры очень удобная и полезная вещь в БД. Так же триггеры имеют определенные правила активации (**firing**), а именно:

1. До момента сработки одного из операторов **DML** - **INSERT, UPDATE, DELETE**.
2. После момента сработки одного из операторов **DML** - **INSERT, UPDATE, DELETE**.

Синтаксис команды для создания триггера, следующий:

```
--- CREATE [OR REPLACE] TRIGGER имя_триггера -----
--- BEFORE | AFTER активизирующее_событие ON ссылка_на_таблицу ---
--- FOR EACH ROW [WHEN условие_срабатывания] -----
--- тело_триггера -----
```

Где:

- **имя\_триггера** - собственно имя вашего триггера.
- **активизирующее\_событие** - указывает момент активации триггера **BEFORE** до срабатывания оператора **DML**, **AFTER** после срабатывания оператора **DML**.
- **ссылка\_на\_таблицу** - собственно таблица, для которой создан триггер.
- **FOR EACH ROW** - если указано активируется от воздействия на строку если нет, то после любого оператора **DML**.
- **условие\_срабатывания** - если **TRUE** триггер срабатывает, если **FALSE** нет.
- **тело\_триггера** - собственно тело триггера.

Для полноты картины определимся с таким фактом, что триггеры имеют собственное пространство имен (**namespace**). Что это означает, само понятие **namespace** - применимо вообще во многих языках программирования. В пределах одного пространства имен не может быть двух функций или процедур с одинаковым именем! Так вот, так как пространство имен у

триггеров свое, то может иметь место ситуация, когда какой-либо триггер имеет тоже имя, что и процедура или функция в пределах одной схемы (не путать с пространством имен это разные вещи!). Но двух триггеров с одинаковым именем не бывает! Так же триггер, может иметь имя совпадающее с именем таблицы, для которой он создан. Но лучше этого не делать, а дать триггеру имя указывающее на то, что он производит! Вообще, по моему мнению, такое понятие как имя для триггера по большому счету архаично! Ведь триггер нельзя "позвать" из процедуры или функции - это запрещено! Так нафига ему имя! Можно было сделать что-то вроде универсальной цифровой маркировки. Хотя имя дает триггеру осмысленность, что улучшает чтение кода БД! Теперь давайте посмотрим на типы триггеров и моменты срабатывания и что все это значит. Итак:

| Категории      | Значение                     | Комментарии                                                                                                                                                                                                                                                         |
|----------------|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Оператор       | INSERT,<br>UPDATE,<br>DELETE | Определяет какой оператор <b>DML</b> вызывает активацию ( <b>firing</b> ) триггера.                                                                                                                                                                                 |
| Момент времени | BEFORE,<br>AFTER             | Определяет момент активации триггера: до или после выполнения оператора.                                                                                                                                                                                            |
| Уровень        | Строка или оператор          | Если триггер является строковым он активируется один раз для каждой из строк, на которую воздействует оператор вызывающий срабатывания триггера (опция <b>FOR EACH ROW</b> ). Если триггер является операторным то он активируется один раз до или после оператора. |

Если посмотреть внимательнее, то значения заданные для оператора, момента времени и уровня, определяют тип триггера. Всего получается 12 возможных типов - 3 оператора, 2 момента времени, 2 уровня. Количество триггеров, для отдельной таблицы в принципе не ограничивается в версии **Oracle 8i** и выше, но делать их слишком много нет смысла. Так же триггер может срабатывать от нескольких операторов **DML**, если это необходимо. Вот собственно, основная теоретическая часть по триггерам БД. Далее попробуем это на практике.

**Шаг 111 - PL/SQL - Триггеры таблиц БД, операторный триггер**

Кое, что о триггерах мы уже знаем. В теории. Давайте попробуем на практике. Запускаем наш "веселый" **SQL\*Plus!** Для демонстрации нам потребуется вспомогательная табличка, вот такого вида:

```
CREATE TABLE MILLER.ADT
(
 USAL VARCHAR2(50),
 TISP DATE
)
/
```

Создадим ее:

```
SQL> CREATE TABLE MILLER.ADT
2 (
3 USAL VARCHAR2(50),
4 TISP DATE
5)
6 /
```

Таблица создана.

Теперь попробуем с помощью нее и нового триггера организовать некий аудит в системе доступа к таблице **MILLER.CUSTOMERSS**. Для этого создадим простой операторный триггер:

```
CREATE OR REPLACE TRIGGER testTrg
 AFTER INSERT OR DELETE OR UPDATE ON customers

DECLARE

BEGIN

INSERT INTO MILLER.ADT(USAL, TISP)
 VALUES(USER, SYSDATE);

END testTrg;
/
```

Откомпилируем его:

```
SQL> CREATE OR REPLACE TRIGGER testTrg
2 AFTER INSERT OR DELETE OR UPDATE ON customers
3
4 DECLARE
5
6 BEGIN
7
8 INSERT INTO MILLER.ADT(USAL, TISP)
9 VALUES(USER, SYSDATE);
```

```
10
11 END testTrg;
12 /
```

Триггер создан.

Триггер создан и будет реагировать на события **INSERT**, **DELETE**, **UPDATE** возникающие при работе с таблицей **MILLER.CUSTOMERSS** после того как они произойдут! Давайте для того, чтобы проверить это сделаем следующее. Добавим запись в таблицу **MILLER.CUSTOMERS** и затем посмотрим на содержимое таблицы **MILLER.ADT**. Вот так:

```
INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
 VALUES (2200, 'MyCompany', 107, 555.5643)
/
```

Видим:

```
SQL> INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
2 VALUES (2200, 'MyCompany', 107, 555.5643)
3 /
```

1 строка создана.

Запись добавлена, замечательно, а что же в таблице аудита **MILLER.ADT**? Смотрим:

```
SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
FROM ADT
/
```

Получаем:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

| USAL   | TISM                 |
|--------|----------------------|
| MILLER | 17/03/2004, 15:25:12 |

Как видно я применил встроенную в **PL/SQL** функцию **TO\_CHAR** с шаблоном вывода даты и времени такого вида: **'DD/MM/YYYY, HH24:MI:SS'**. И действительно 17-го марта в 15:25:12 я добавил запись в таблицу **MILLER.CUSTOMERS**. Хотя это не совсем и очевидно, но тем не менее. А, что если мы попробуем дать оператор отката транзакции **ROLLBACK**! Обратите внимание - я не давал оператор **COMMIT**, но триггер сработал! Собственно это и очевидно! Итак откатимся:

```
ROLLBACK
/
```

Видим:

```
SQL> ROLLBACK
```



2 /

Откат завершен.

Посмотрим в нашу табличку **ADT**:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

строки не выбраны

Все откат вернул все на круги своя. Давайте снова добавим запись уже закрепив все оператором **COMMIT** и пойдем дальше. Итак:

```
INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
VALUES (2200, 'MyCompany', 107, 555.5643)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
2 VALUES (2200, 'MyCompany', 107, 555.5643)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим табличку **ADT**:

```
SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
FROM ADT
/
```

Получаем:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

| USAL   | TISM                 |
|--------|----------------------|
| MILLER | 17/03/2004, 15:29:03 |

Вот все получилось! Давайте теперь попробуем изменить запись в **MILLER.CUSTOMERS**, вот таким образом:

```
UPDATE CUSTOMERS
SET cust_num = 2222
WHERE cust_num = 2200
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE CUSTOMERS
2 SET cust_num = 2222
3 WHERE cust_num = 2200
4 /
```

1 строка обновлена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Посмотрим, что в **MILLER.ADT**:

```
SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
FROM ADT
/
```

Получаем:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

| USAL   | TISM                 |
|--------|----------------------|
| MILLER | 17/03/2004, 15:29:03 |
| MILLER | 17/03/2004, 15:31:01 |

Вот теперь добавление записи и ее изменение зафиксировано как факт и никуда не денешься! Давайте удалим теперь лишнюю запись в **MILLER.CUSTOMERS**. Таким вот образом:

```
DELETE FROM CUSTOMERS
WHERE cust_num = 2222
/

COMMIT
/
```

Фиксация обновлений завершена.

Получаем:

```
SQL> DELETE FROM CUSTOMERS
2 WHERE cust_num = 2222
3 /
```

1 строка удалена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Все удалилось, пробуем смотреть **MILLER.ADT**:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

| USAL   | TISM                 |
|--------|----------------------|
| MILLER | 17/03/2004, 15:29:03 |
| MILLER | 17/03/2004, 15:31:01 |
| MILLER | 17/03/2004, 15:31:41 |

Все три действия с нашей учебной табличкой **MILLER.CUSTOMERS** были безупречно зафиксированы триггером **testTrg**. Давайте посмотрим, как определен наш триггер в словаре данных, а именно в представлении **USER\_TRIGGERS**, дадим вот такой запрос:

```
SELECT TRIGGER_NAME, TRIGGER_TYPE, TABLE_NAME, TRIGGERING_EVENT
FROM USER_TRIGGERS
WHERE TRIGGER_NAME = 'TESTTRG'
/
```

Получаем:

```
SQL> SELECT TRIGGER_NAME, TRIGGER_TYPE, TABLE_NAME, TRIGGERING_EVENT
2 FROM USER_TRIGGERS
3 -- WHERE TRIGGER_NAME = 'testTrg'
4 /
```

| TRIGGER_NAME | TRIGGER_TYPE    | TABLE_NAME | TRIGGERING_EVENT           |
|--------------|-----------------|------------|----------------------------|
| TESTTRG      | AFTER STATEMENT | CUSTOMERS  | INSERT OR UPDATE OR DELETE |

Обратите внимание, что в условии **WHERE**, я пишу **'TESTTRG'**, а не **'testTrg'**! Хорошо видно, что весь наш с вами триггер расписан и ясно, что и для чего! Так же если есть необходимость на некоторое время прекратить вызов определенного вами триггера, но не удалять его тело полностью, примените команду:

```
ALTER TRIGGER TESTTRG DISABLE
```

Получаем:

```
SQL> ALTER TRIGGER TESTTRG DISABLE
2 /
```

Триггер изменен.

Если теперь снова добавить запись в **MILLER.CUSTOMERS**, то в **MILLER.ADT** ничего нового не появится! Давайте проверим:

```
INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
VALUES (2200, 'MyCompany', 107, 555.5643)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO CUSTOMERS(cust_num, company, cust_rep, credit_limit)
2 VALUES (2200, 'MyCompany', 107, 555.5643)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Снова смотрим содержимое **MILLER.ADT**:

```
SQL> SELECT USAL, TO_CHAR(TISP,'DD/MM/YYYY, HH24:MI:SS') TISM
2 FROM ADT
3 /
```

| USAL   | TISM                 |
|--------|----------------------|
| MILLER | 05/03/2004, 15:29:03 |
| MILLER | 05/03/2004, 15:31:01 |
| MILLER | 05/03/2004, 15:31:41 |

Три записи и ничего нового! Триггер "спит"! Удалим лишнюю запись из учебной таблички:

```
DELETE FROM CUSTOMERS
WHERE cust_num = 2200
/

COMMIT
/
```

Фиксация обновлений завершена.

Удалено:

```
SQL> DELETE FROM CUSTOMERS
 2 WHERE cust_num = 2200
 3 /
```

1 строка удалена.

```
SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.

И конечно триггер удаляется с помощью оператора **DROP TRIGGER**, вот таким образом:

```
DROP TRIGGER TESTTRG
/
```

Получаем:

```
SQL> DROP TRIGGER TESTTRG
 2 /
```

Триггер удален.

Вот и все, кирдык! Мучились, мучились, а в конце концов все убили! Чего у нас такие истории с несчастливыми окончаниями, не пойму! Но я думаю, что для вас это совсем не так! Можете кстати пока удалить и табличку **MILLER.ADT** далее она нам не понадобится! Как это делать вы уже знаете или нет? Работайте!

---

## Шаг 112 - PL/SQL - Триггеры - строковые и операторные

Продолжаем разбирать триггеры. Каждый из них, как и любой другой объект БД, после создания хранится в словаре данных в виде **Р-кода**. Ранее до версии **Oracle 7.3** триггеры хранились в словаре данных, в виде исходного кода. И каждые раз при вызове компилировались, а затем исполнялись. В более старших версиях **Oracle**, триггеры хранятся уже в скомпилированном виде. В результате, так же как и модули и подпрограммы, могут автоматически становиться недостоверными. Но, становясь недостоверным, триггер компилируется при следующей его активации. Активация триггеров, как вы уже знаете происходит при выполнении операторов **DML**. Порядок активации триггеров в большинстве случаев таков:

1. Выполняется операторный триггер **BEFORE** (при его наличии)
2. Для каждой строки, на которую воздействует оператор:
  - a. Выполняется строковый триггер **BEFORE** (при его наличии).
  - b. Выполняется собственно оператор.
  - c. Выполняется строковый триггер **AFTER** (при его наличии).
3. Выполняется операторный триггер **AFTER** (при его наличии).

Давайте разберемся со всем этим более детально, чтобы четко представлять, всю картину работы триггеров таблиц БД. Но для начала, создадим пару табличек и наполним их данными. В дальнейшем будем работать с ними как учебными в этом разделе. В следствии того, что я не хочу пока менять что-либо в наших основных пяти учебных табличках. Итак, создаем первую таблицу:

```
CREATE TABLE MILLER.TSTTRIG
(
 ID NUMBER PRIMARY KEY,
 NM VARCHAR2(50),
 ROD VARCHAR2(50),
 INRW DATE
)
```

Получаем:

```
SQL> CREATE TABLE MILLER.TSTTRIG
2 (
3 ID NUMBER PRIMARY KEY,
4 NM VARCHAR2(50),
5 ROD VARCHAR2(50),
6 INRW DATE
7)
8 /
```

Таблица создана.

Заполняем ее поля такими данными:

```
INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
VALUES (7369, 'SMITH', 'CLERK', TO_DATE('17-2-2000', 'DD-MM-YYYY'))
/
```

```
INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 VALUES (7370, 'JONES', 'MANAGER', TO_DATE('2-4-2001', 'DD-MM-YYYY'))
/

INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 VALUES (7371, 'MILLER', 'SALESMAN', TO_DATE('20-3-2003', 'DD-MM-YYYY'))
/

INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 VALUES (7372, 'SCOTT', 'ANALYST', TO_DATE('09-12-2001', 'DD-MM-YYYY'))
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 2 VALUES (7369, 'SMITH', 'CLERK', TO_DATE('17-2-2000', 'DD-MM-YYYY'))
 3 /

1 строка создана.

SQL> INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 2 VALUES (7370, 'JONES', 'MANAGER', TO_DATE('2-4-2001', 'DD-MM-YYYY'))
 3 /

1 строка создана.

SQL> INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 2 VALUES (7371, 'MILLER', 'SALESMAN', TO_DATE('20-3-2003', 'DD-MM-YYYY'))
 3 /

1 строка создана.

SQL> INSERT INTO TSTTRIG (ID, NM, ROD, INRW)
 2 VALUES (7372, 'SCOTT', 'ANALYST', TO_DATE('09-12-2001', 'DD-MM-YYYY'))
 3 /

1 строка создана.

SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.

Итак, одна табличка готова, создаем еще одну:

```
CREATE TABLE MILLER.TSTSV
(
 ID NUMBER PRIMARY KEY,
```

---

```
IDD VARCHAR2(50),
ROD VARCHAR2(50),
CONS NUMBER
)
/
```

Получаем:

```
SQL> CREATE TABLE MILLER.TSTSV
2 (
3 ID NUMBER PRIMARY KEY,
4 IDD VARCHAR2(50),
5 ROD VARCHAR2(50),
6 CONS NUMBER
7)
8 /
```

Таблица создана.

Заполняем ее поля такими данными:

```
INSERT INTO TSTSV (ID, IDD, ROD, CONS)
VALUES (1, 'SMITH', 'CLERK', 7369)
/

INSERT INTO TSTSV (ID, IDD, ROD, CONS)
VALUES (2, 'JONES', 'MANAGER', 7370)
/

INSERT INTO TSTSV (ID, IDD, ROD, CONS)
VALUES (3, 'MILLER', 'SALESMAN', 7371)
/

INSERT INTO TSTSV (ID, IDD, ROD, CONS)
VALUES (4, 'SCOTT', 'ANALYST', 7372)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO TSTSV (ID, IDD, ROD, CONS)
2 VALUES (1, 'SMITH', 'CLERK', 7369)
3 /
```

1 строка создана.

```
SQL> INSERT INTO TSTSV (ID, IDD, ROD, CONS)
2 VALUES (2, 'JONES', 'MANAGER', 7370)
3 /
```

---



1 строка создана.

```
SQL> INSERT INTO TSTSV (ID, IDD, ROD, CONS)
2 VALUES (3, 'MILLER', 'SALESMAN', 7371)
3 /
```

1 строка создана.

```
SQL> INSERT INTO TSTSV (ID, IDD, ROD, CONS)
2 VALUES (4, 'SCOTT', 'ANALYST', 7372)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Строим зависимость таблицы **MILLER.TSTSV** от таблицы **MILLER.TSTTRIG** по полям **CONS** и **ID** соответственно:

```
ALTER TABLE MILLER.TSTSV ADD
FOREIGN KEY (CONS) REFERENCES MILLER.TSTTRIG (ID)
/
```

Получаем:

```
SQL> ALTER TABLE MILLER.TSTSV ADD
2 FOREIGN KEY (CONS) REFERENCES MILLER.TSTTRIG (ID)
3 /
```

Таблица изменена.

Убедимся в том, что связь двух таблиц работает. Дадим такой оператор:

```
INSERT INTO TSTSV (ID, IDD, ROD, CONS)
VALUES (5, NULL, NULL, 7380)
/
```

Получаем соответственно:

```
SQL> INSERT INTO TSTSV (ID, IDD, ROD, CONS)
2 VALUES (5, NULL, NULL, 7380)
3 /
INSERT INTO TSTSV (ID, IDD, ROD, CONS)
*
```

ошибка в строке 1:

ORA-02291: нарушено ограничение целостности (MILLER.SYS\_C003552) - исходный  
ключ не найден

---

Все верно! Нарушена ссылочная целостность! Так же давайте видоизменим нашу табличку из прошлых шагов **MILLER.ADT**, помните? Мы в нее записывали события! Прделаем следующее - добавим в нее еще два поля. Вот так:

```
ALTER TABLE MILLER.ADT ADD WDO VARCHAR2(50)
/

ALTER TABLE MILLER.ADT ADD PRIM VARCHAR2(50)
/
```

Получаем следующее:

```
SQL> ALTER TABLE MILLER.ADT ADD WDO VARCHAR2(50)
2 /
```

Таблица изменена.

```
SQL> ALTER TABLE MILLER.ADT ADD PRIM VARCHAR2(50)
2 /
```

Таблица изменена.

Вот теперь все готово для наших дальнейших действий. Чтобы понять в чем основное отличие операторного триггера от строкового проделаем следующие действия. Создадим для таблицы **MILLER.TSTTRIG** два операторных триггера по одному на каждое из двух действий **BEFORE** и **AFTER** и сработку установим на событие **UPDATE**. Вот так:

```
CREATE OR REPLACE TRIGGER BFOTST
BEFORE UPDATE ON TSTTRIG

DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
VALUES(USER, SYSDATE, 'Update', 'Before Statement trigger');

END BFOTST;
/

CREATE OR REPLACE TRIGGER AFTTST
AFTER UPDATE ON TSTTRIG

DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
VALUES(USER, SYSDATE, 'Update', 'After Statement trigger');

END AFTTST;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER BFOTST
 2 BEFORE UPDATE ON TSTTRIG
 3
 4 DECLARE
 5
 6 BEGIN
 7
 8 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
 9 VALUES(USER, SYSDATE, 'Update', 'Before Statement trigger');
10
11 END BFOTST;
12 /
```

Триггер создан.

```
SQL> CREATE OR REPLACE TRIGGER AFTTST
 2 AFTER UPDATE ON TSTTRIG
 3
 4 DECLARE
 5
 6 BEGIN
 7
 8 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
 9 VALUES(USER, SYSDATE, 'Update', 'After Statement trigger');
10
11 END AFTTST;
12 /
```

Триггер создан.

Операторные триггеры созданы. Теперь создадим строковые по тому же принципу, но отличаться они будут наличием конструкции **FOR EACH ROW**. Вот таким образом:

```
CREATE OR REPLACE TRIGGER BFOTSTR
BEFORE UPDATE ON TSTTRIG
FOR EACH ROW

DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
 VALUES(USER, SYSDATE, 'Update', 'Before Row trigger');

END BFOTSTR;
/

CREATE OR REPLACE TRIGGER AFTTSTR
AFTER UPDATE ON TSTTRIG
FOR EACH ROW
```

---

```
DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
 VALUES(USER, SYSDATE, 'Update', 'After Row trigger');

END AFTTSTR;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER BFOTSTR
 2 BEFORE UPDATE ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 BEGIN
 8
 9 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
10 VALUES(USER, SYSDATE, 'Update', 'Before Row trigger');
11
12 END BFOTSTR;
13 /
```

Триггер создан.

```
SQL> CREATE OR REPLACE TRIGGER AFTTSTR
 2 AFTER UPDATE ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 BEGIN
 8
 9 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
10 VALUES(USER, SYSDATE, 'Update', 'After Row trigger');
11
12 END AFTTSTR;
13 /
```

Триггер создан.

Строковые триггеры созданы. Перед тем как проверить отличия в их действиях, очистим таблицку **MILLER.ADT** от прошлых данных:

```
DELETE FROM MILLER.ADT
/
COMMIT
/
```

---

А вот теперь попробуем ввести такой оператор, для таблички **MILLER.TSTTRIG**:

```
UPDATE MILLER.TSTTRIG
SET ROD = 'SPOOKY'
WHERE ID IN (7369, 7370)
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE MILLER.TSTTRIG
2 SET ROD = 'SPOOKY'
3 WHERE ID IN (7369, 7370)
4 /
```

2 строк обновлено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Изменено две строки! Так как условие оператора **UPDATE** соответствует в нашем случае двум записям. Верно? Смотрим содержимое таблички **MILLER.ADT**:

```
SQL> SELECT * FROM MILLER.ADT
2 /
```

| USAL   | TISP       | WDO    | PRIM                     |
|--------|------------|--------|--------------------------|
| MILLER | 17.03.2004 | Update | Before Statement trigger |
| MILLER | 17.03.2004 | Update | Before Row trigger       |
| MILLER | 17.03.2004 | Update | After Row trigger        |
| MILLER | 17.03.2004 | Update | Before Row trigger       |
| MILLER | 17.03.2004 | Update | After Row trigger        |
| MILLER | 17.03.2004 | Update | After Statement trigger  |

6 rows selected

Думаю, очень хорошо видно, что операторный триггер сработал только два раза на **BEFORE** и **AFTER**, а вот строковый четыре раза! Именно потому, что изменили две строки! И каждый из них среагировал дважды - в результате имеем четыре сработки двух триггеров! Это нужно очень хорошо представлять и применять каждый из данного типа триггеров, именно тогда когда в этом есть необходимость! Для полноты картины вернем полям таблицы **MILLER.TSTTRIG** прежние значения и очистим таблицу **MILLER.ADT**:

```
UPDATE MILLER.TSTTRIG
SET ROD = 'CLERK'
WHERE ID = 7369
/
```

```
UPDATE MILLER.TSTTRIG
SET ROD = 'MANAGER'
WHERE ID = 7370
/

COMMIT
/

DELETE FROM MILLER.ADT
/

COMMIT
/
```

Получим:

```
SQL> UPDATE MILLER.TSTTRIG
 2 SET ROD = 'CLERK'
 3 WHERE ID = 7369
 4 /
```

1 строка обновлена.

```
SQL> UPDATE MILLER.TSTTRIG
 2 SET ROD = 'MANAGER'
 3 WHERE ID = 7370
 4 /
```

1 строка обновлена.

```
SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.

```
SQL> DELETE FROM MILLER.ADT
 2 /
```

18 строк удалено.

```
SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.

Вот пока все, что касается типов триггеров, с которыми в основном вам придется работать! Можете проделать все еще раз сначала, для лучшего запоминания! :)

---

## Шаг 113 - PL/SQL - Триггеры - псевдозаписи - ЧАСТЬ I

Продолжаем наше движение на пути познания триггеров. Одним из интересных моментов при создании строковых триггеров является наличие двух псевдозаписей **:old** и **:new**. Строковый триггер срабатывает один раз для каждой строки. При этом внутри триггера можно обращаться к строке обрабатываемой в данный момент времени. Делать это можно как вы уже поняли, применяя псевдозаписи. По своей сути **:old** и **:new** вообще-то записями в полном понимании не являются. Например, как рассмотренном нами ранее - **таблица%ROWTYPE**. Нельзя например, записать:

```
CREATE OR REPLACE TRIGGER DLTSTR
BEFORE DELETE ON TSTTRIG
FOR EACH ROW

m_ROW TSTTRIG%ROWTYPE;

DECLARE

BEGIN

m_ROW = :old; -- Вызовет ошибку компиляции!

.
.
.
```

Но при этом верно утверждение что **:old** и **:new** это **активизирующая\_таблица%ROWTYPE**, где **активизирующая\_таблица** - это таблица, для которой создан триггер.

Но обращение в нашем случае типа:

```
CREATE OR REPLACE TRIGGER DLTSTR
BEFORE DELETE ON TSTTRIG
FOR EACH ROW

a TSTTRIG.ID%TYPE;
b TSTTRIG.NM%TYPE;
c TSTTRIG.ROD%TYPE;

DECLARE

BEGIN

-- Верно синтаксически!
a = :old.ID;
c = :new.ROD;
b = :old.NM;

.
.
.
```

Ошибок компиляции не вызовет! Думаю понятно, что обращение к псевдозаписям **:old** и **:new** должно производиться через имена полей, по этому они и называются псевдозаписями! Естественно, что все вышесказанное применимо только к строковым триггерам! Обращение к **:old** и **:new** в операторных триггерах вызовет ошибку компиляции! Давайте опишем некоторые положения для псевдозаписей **:old** и **:new** применимо к операторам **DML**:

| Активизирующий оператор | :OLD                                                             | :NEW                                                            |
|-------------------------|------------------------------------------------------------------|-----------------------------------------------------------------|
| INSERT                  | Не определена во всех полях содержится NULL значения             | Значения, которые будут введены после выполнения оператора.     |
| UPDATE                  | Исходные значения содержащиеся в строке перед обновлением данных | Новые значения которые будут введены после выполнения оператора |
| DELETE                  | Исходные значения содержащиеся в строке перед ее удалением       | Не определена во всех полях содержится NULL значения            |

Очень хорошо видно, что псевдозапись **:old** не определена для оператора **INSERT**, а **:new** для оператора **DELETE**! Хотя их применение в триггерах не вызовет ошибки, а значения которые вы получите будут - **NULL**! Например, есть очень эффективный трюк с заполнением ключевых полей с использованием триггера! Попробуйте выполнить вот такой **INSERT** для таблицы **MILLER.TSTTRIG**:

```
INSERT INTO TSTTRIG (NM, ROD, INRW)
VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
/
```

Получите примерно вот это:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW)
2 VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
3 /
INSERT INTO TSTTRIG (NM, ROD, INRW)
*
ошибка в строке 1:
ORA-01400: невозможно вставить NULL в ("MILLER"."TSTTRIG"."ID")
```

Естественно нельзя в таблицу добавить запись, не добавляя значение в поле **ID**, которое является первичным ключом таблицы **MILLER.TSTTRIG**! Как бы решить эту проблему так, чтобы голова не болела, какой номер следующий в первичном ключе и как бы не ошибиться в следующий раз! А очень просто, применив последовательность и триггер **BEFORE INSERT**! Сначала создадим последовательность для получения уникальных значений для таблицы **MILLER.TSTTRIG**. Последовательность напомним - это такой объект БД, который выдает уникальные значения для первичных ключей таблиц БД! Создаем ее:

```
CREATE SEQUENCE TRG
START WITH 8000
INCREMENT BY 1
/
```

Начальное значение 8000 и шаг 1-ка:



```
SQL> CREATE SEQUENCE TRG
 2 START WITH 8000
 3 INCREMENT BY 1
 4 /
```

Последовательность создана.

А, вот теперь создаем триггер с применением псевдозаписи **:new** - вот так:

```
CREATE OR REPLACE TRIGGER INSIDTRG
BEFORE INSERT ON TSTTRIG
FOR EACH ROW

DECLARE

BEGIN

SELECT TRG.NEXTVAL
 INTO :NEW.ID
FROM DUAL;

END INSIDTRG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER INSIDTRG
 2 BEFORE INSERT ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 BEGIN
 8
 9 SELECT TRG.NEXTVAL
 10 INTO :NEW.ID
 11 FROM DUAL;
 12
 13 END INSIDTRG;
 14 /
```

Триггер создан.

А вот теперь снова попробуем дать наш прошлый **INSERT**:

```
INSERT INTO TSTTRIG (NM, ROD, INRW)
VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
/

COMMIT
/
```

---

Получаем:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW)
2 VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Ух ты! Прокатило! Но как? Ведь поле **ID** таблицы **MILLER.TSTTRIG** в нашем случае не получает значения! Да все просто, сработал триггер и отправил значение **8000** в поле **ID** таблицы **MILLER.TSTTRIG** при помощи оператора:

```
SELECT TRG.NEXTVAL
INTO :NEW.ID
FROM DUAL;
```

В данном случае **:NEW.ID** вполне справедливо! **FROM DUAL** - это некая псевдотаблица внутри **Oracle** для получения отвлеченных значений для **SELECT**. Ее описание выглядит следующим образом:

```
SQL> DESC DUAL
Имя Пусто? Тип

DUMMY VARCHAR2(1)
```

На всякий случай посмотрим содержимое таблички **MILLER.TSTTRIG**:

```
SQL> SELECT * FROM TSTTRIG
2 /
```

| ID   | NM     | ROD      | INRW       |
|------|--------|----------|------------|
| 7369 | SMITH  | CLERK    | 17.02.2000 |
| 7370 | JONES  | MANAGER  | 02.04.2001 |
| 7371 | MILLER | SALESMAN | 20.03.2003 |
| 7372 | SCOTT  | ANALYST  | 09.12.2001 |
| 8000 | BLAKE  | MANAGER  | 08.05.1999 |

Вот и вся хитрость! Кстати этот фокус с триггером для заполнения ключевых полей очень пригодиться вам в будущем! Можете пользоваться! А теперь для полного понимания попробуем изменить триггеры из [прошлого шага](#) и рассмотреть псевдозаписи **:old** и **:new** более детально! Итак, создадим и изменим триггера:

```
CREATE OR REPLACE TRIGGER DLTSTR
BEFORE DELETE ON TSTTRIG
FOR EACH ROW
```

```
DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
VALUES(USER, SYSDATE, NULL, 'DELETE '||:old.ROD);

END BFOTSTR;
/

CREATE OR REPLACE TRIGGER BFOTSTR
BEFORE UPDATE ON TSTTRIG
FOR EACH ROW

DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
VALUES(USER, SYSDATE, :OLD.ROD, 'UPDATE TO '||:new.ROD);

END BFOTSTR;
/

CREATE OR REPLACE TRIGGER AFTTSTR
AFTER UPDATE ON TSTTRIG
FOR EACH ROW

DECLARE

BEGIN

INSERT INTO ADT(USAL, TISP, WDO, PRIM)
VALUES(USER, SYSDATE, :new.ROD, :old.ROD);

END AFTTSTR;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER DLTTSTR
2 BEFORE DELETE ON TSTTRIG
3 FOR EACH ROW
4
5 DECLARE
6
7 BEGIN
8
9 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
10 VALUES(USER, SYSDATE, NULL, 'DELETE '||:old.ROD);
11
12 END BFOTSTR;
```

---

13 /

Триггер создан.

```
SQL> CREATE OR REPLACE TRIGGER BFOTSTR
 2 BEFORE UPDATE ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 BEGIN
 8
 9 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
10 VALUES(USER, SYSDATE, :OLD.ROD, 'UPDATE TO '||:new.ROD);
11
12 END BFOTSTR;
13 /
```

Триггер создан.

```
SQL> CREATE OR REPLACE TRIGGER AFTTSTR
 2 AFTER UPDATE ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 BEGIN
 8
 9 INSERT INTO ADT(USAL, TISP, WDO, PRIM)
10 VALUES(USER, SYSDATE, :new.ROD, :old.ROD);
11
12 END AFTTSTR;
13 /
```

Триггер создан.

Триггер **DLTTSTR** срабатывает на удаление строки из таблицы **MILLER.TSTTRIG**, а два триггера **BFOTSTR**, **AFTTSTR** из [прошлого шага](#) теперь вставляют в таблицу **MILLER.ADT** старые (:old) и новые (:new) значения для полей активизирующей таблицы. Очистим таблицу **MILLER.ADT** для чистоты эксперимента:

```
DELETE FROM MILLER.ADT
/

COMMIT
/
```

Снова изменим, значения полей таблички **MILLER.TSTTRIG**:

```
UPDATE MILLER.TSTTRIG
SET ROD = 'SPOOKY'
```

---

```
WHERE ID IN (7369, 7370)
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE MILLER.TSTTRIG
2 SET ROD = 'SPOOKY'
3 WHERE ID IN (7369, 7370)
4 /
```

2 строк обновлено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое - **MILLER.ADT**:

```
SQL> SELECT WDO, PRIM FROM MILLER.ADT
2 /
```

| WDO     | PRIM                     |
|---------|--------------------------|
| Update  | Before Statement trigger |
| MANAGER | UPDATE TO SPOOKY         |
| SPOOKY  | MANAGER                  |
| CLERK   | UPDATE TO SPOOKY         |
| SPOOKY  | CLERK                    |
| Update  | After Statement trigger  |

6 строк выбрано.

Видно как сработал операторный триггер первая и последняя строки и строковые триггеры, показали **:old** и **:new** значения полей для нашей таблички **MILLER.TSTTRIG**. Думаю теперь ясно, как работать с псевдозаписями **:old** и **:new**! Думаю пришла пора заставить их делать что-то более полезное. Давайте, реализуем бизнес - правило, одно из ключевых применений строчных триггеров. И кое что еще! Недаром вначале я делал две таблицы и связывал их! Давайте для начала удалим связь **MILLER.TSTTRIG** и **MILLER.TSTSV** для этого нужно узнать ее имя, как это делать смотрите ["Шаг 100 - PL/SQL - Уровни строгости - Прагма RESTRICT REFERENCES"](#). В моем случае она зовется **SYS\_C003552** удалим ее:

```
SQL> ALTER TABLE MILLER.TSTSV DROP CONSTRAINT SYS_C003552
2 /
```

Таблица изменена.

Теперь будем писать осмысленные триггеры реализующие бизнес правила для таблиц **MILLER.TSTTRIG** и **MILLER.TSTSV**! Сначала удалим все наши учебные триггеры больше они нам не понадобятся (оставим только **INSIDTRG** он нам нужен):

```
SQL> DROP TRIGGER AFTTST
2 /
```

Триггер удален.

```
SQL> DROP TRIGGER AFTTSTR
2 /
```

Триггер удален.

```
SQL> DROP TRIGGER BFOTST
2 /
```

Триггер удален.

```
SQL> DROP TRIGGER BFOTSTR
2 /
```

Триггер удален.

```
SQL> DROP TRIGGER DLTSTR
2 /
```

Триггер удален.

Все, от мусора избавились, пора заняться делом! Чтобы не превышать лимит по объему материала продолжим в следующем шаге! Не уходите далеко!

---

## Шаг 114 - PL/SQL - Триггеры - псевдозаписи - ЧАСТЬ II (бизнес правила)

На чем мы там остановились? Да! Бизнес правила. Итак, давайте создадим некий костяк, который будет определять взаимоотношения наших двух таблиц. Начнем с добавления записи в таблицу **MILLER.TSTTRIG**. Предположим, что каждое добавление записи должно отражаться сразу в двух таблицах БД. Как это сделать? Написать триггер БД, который реализует данное правило. Для работы нам понадобится еще одна последовательность, для таблицы **MILLER.TSTSV** - создадим ее:

```
CREATE SEQUENCE SV
 START WITH 5
 INCREMENT BY 1
/
```

Получаем:

```
SQL> CREATE SEQUENCE SV
2 START WITH 5
3 INCREMENT BY 1
4 /
```

Последовательность создана.

Теперь создадим триггер для таблицы **MILLER.TSTTRIG**, который одновременно будет менять содержимое таблицы - **MILLER.TSTSV** реализуя наше бизнес правило для **MILLER.TSTTRIG**. Создаем:

```
CREATE OR REPLACE TRIGGER AFTINSTTRIG
 AFTER INSERT ON TSTTRIG
 FOR EACH ROW

DECLARE

BEGIN

INSERT INTO MILLER.TSTSV(MILLER.TSTSV.ID, MILLER.TSTSV.IDD, MILLER.TSTSV.ROD,
MILLER.TSTSV.CONST)
 VALUES(SV.NEXTVAL, :NEW.NM, :NEW.ROD, :NEW.ID);

END AFTINSTTRIG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER AFTINSTTRIG
2 AFTER INSERT ON TSTTRIG
3 FOR EACH ROW
4
5 DECLARE
6
7 BEGIN
```

```
8
9 INSERT INTO MILLER.TSTSV(MILLER.TSTSV.ID, MILLER.TSTSV.IDD, MILLER.TSTSV.ROD,
MILLER.TSTSV.CONC
)
10 VALUES(SV.NEXTVAL, :NEW.NM, :NEW.ROD, :NEW.ID);
11
12 END AFTINSTTRIG;
13 /
```

Триггер создан.

Триггер срабатывает после вставки в таблицу **MILLER.TSTTRIG** и так же добавляет данные согласно нашему бизнес правилу! Проверим его работу. Удалите запись в таблице **MILLER.TSTTRIG** с **ID = 8000**! (или с другим, если вы уже что-то меняли!):

```
SQL> DELETE FROM TSTTRIG
2 WHERE ID = 8000
3 /
```

1 строка удалена.

Вот так. А теперь возьмем оператор из [прошлого шага](#) - помните:

```
INSERT INTO TSTTRIG (NM, ROD, INRW)
VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
/

COMMIT
/
```

После добавления:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW)
2 VALUES ('BLAKE', 'MANAGER', TO_DATE('8-5-1999', 'DD-MM-YYYY'))
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое таблиц **MILLER.TSTTRIG** и **MILLER.TSTSV**:

```
SELECT * FROM MILLER.TSTTRIG
/

SELECT * FROM MILLER.TSTSV
/
```

Получаем:



```
SQL> SELECT * FROM MILLER.TSTTRIG
2 /
```

| ID   | NM     | ROD      | INRW       |
|------|--------|----------|------------|
| 7369 | SMITH  | CLERK    | 17.02.2000 |
| 7370 | JONES  | MANAGER  | 02.04.2001 |
| 7371 | MILLER | SALESMAN | 20.03.2003 |
| 7372 | SCOTT  | ANALYST  | 09.12.2001 |
| 8001 | BLAKE  | MANAGER  | 08.05.1999 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID | IDD    | ROD      | CONS         |
|----|--------|----------|--------------|
| 1  | SMITH  | CLERK    | 7369         |
| 2  | JONES  | MANAGER  | 7370         |
| 3  | MILLER | SALESMAN | 7371         |
| 4  | SCOTT  | ANALYST  | 7372         |
|    | 5      | BLAKE    | MANAGER 8001 |

Первый триггер сработал верно, тоже самое будет, если вы добавите еще запись! Таким образом один **INSERT** работает на две таблицы. Триггер типа **AFTER** т.к. поле **:NEW.ID** определяется после вставки! Думаю все догадались, что здесь сработало два триггера (!) второй определил поле **ID** в **MILLER.TSTTRIG**! А вы даже об этом не задумались! Удобно верно? :) Идем дальше - обновление для пары таблиц:

```
CREATE OR REPLACE TRIGGER AFTUPDTTRIG
AFTER UPDATE ON TSTTRIG
FOR EACH ROW
```

```
DECLARE
```

```
BEGIN
```

```
UPDATE MILLER.TSTSV
SET
 MILLER.TSTSV.IDD = :NEW.NM,
 MILLER.TSTSV.ROD = :NEW.ROD,
 MILLER.TSTSV.CONNS = :NEW.ID
```

```
WHERE
 MILLER.TSTSV.CONNS = :OLD.ID;
```

```
END AFTUPDTTRIG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER AFTUPDTTRIG
2 AFTER UPDATE ON TSTTRIG
```

```
3 FOR EACH ROW
4
5 DECLARE
6
7 BEGIN
8
9 UPDATE MILLER.TSTSV
10 SET
11 MILLER.TSTSV.IDD = :NEW.NM,
12 MILLER.TSTSV.ROD = :NEW.ROD,
13 MILLER.TSTSV.CONNS = :NEW.ID
14
15 WHERE
16 MILLER.TSTSV.CONNS = :OLD.ID;
17
18 END AFTUPDTTRIG;
19 /
```

Триггер создан.

Данный триггер следит за изменением полей **NM**, **ROD**, **ID** в таблице **MILLER.TSTTRIG** и отражает изменения в таблице **MILLER.TSTSV**. Например, попробуйте вот этот оператор:

```
UPDATE MILLER.TSTTRIG
SET ROD = 'SPOOKY'
WHERE ID IN (7369, 7370)
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE MILLER.TSTTRIG
2 SET ROD = 'SPOOKY'
3 WHERE ID IN (7369, 7370)
4 /
```

2 строк обновлено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое табличек **MILLER.TSTTRIG** и **MILLER.TSTSV**:

```
SELECT * FROM MILLER.TSTTRIG
/

SELECT * FROM MILLER.TSTSV
/
```

---

Получаем:

```
SQL> SELECT * FROM MILLER.TSTTRIG
2 /
```

| ID NM       | ROD      | INRW       |
|-------------|----------|------------|
| 7369 SMITH  | SPOOKY   | 17.02.2000 |
| 7370 JONES  | SPOOKY   | 02.04.2001 |
| 7371 MILLER | SALESMAN | 20.03.2003 |
| 7372 SCOTT  | ANALYST  | 09.12.2001 |
| 8001 BLAKE  | MANAGER  | 08.05.1999 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID IDD   | ROD      | CONS |
|----------|----------|------|
| 1 SMITH  | SPOOKY   | 7369 |
| 2 JONES  | SPOOKY   | 7370 |
| 3 MILLER | SALESMAN | 7371 |
| 4 SCOTT  | ANALYST  | 7372 |
| 5 BLAKE  | MANAGER  | 8001 |

Изменения отражены в обеих таблицах! Что и должно было произойти при реализации этого бизнес правила! Один **UPDATE** отразился на обеих таблицах. Можете сами проверить действие изменений поля **CONS** зависящее от поля **ID** в этих двух таблицах. Ну и последнее удаление данных. Запишем такой триггер:

```
CREATE OR REPLACE TRIGGER BFRDELTTTRIG
 BEFORE DELETE ON TSTTRIG
 FOR EACH ROW
```

```
DECLARE
```

```
BEGIN
```

```
DELETE FROM MILLER.TSTSV
 WHERE MILLER.TSTSV.CONNS = :OLD.ID;
```

```
END BFRDELTTTRIG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER BFRDELTTTRIG
2 BEFORE DELETE ON TSTTRIG
3 FOR EACH ROW
4
5 DECLARE
6
7 BEGIN
```

```
8
9 DELETE FROM MILLER.TSTSV
10 WHERE MILLER.TSTSV.CON = :OLD.ID;
11
12 END BFRDELTTTRIG;
13 /
```

Триггер создан.

Здесь используется **BEFORE**, то есть "перед тем как уберу за собой"! :) Смотрим как он работает. Удалим запись с номером **8001** в таблице **MILLER.TSTTRIG** (или другой, если вы уже изменяли ее) вот так:

```
SQL> DELETE FROM TSTTRIG
2 WHERE ID = 8001
3 /
```

1 строка удалена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим таблички **MILLER.TSTTRIG** и **MILLER.TSTSV**:

```
SELECT * FROM MILLER.TSTTRIG
/
```

```
SELECT * FROM MILLER.TSTSV
/
```

Получаем:

```
SQL> SELECT * FROM MILLER.TSTTRIG
2 /
```

| ID NM       | ROD      | INRW       |
|-------------|----------|------------|
| 7369 SMITH  | SPOOKY   | 17.02.2000 |
| 7370 JONES  | SPOOKY   | 02.04.2001 |
| 7371 MILLER | SALESMAN | 20.03.2003 |
| 7372 SCOTT  | ANALYST  | 09.12.2001 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID IDD   | ROD      | CONS |
|----------|----------|------|
| 1 SMITH  | SPOOKY   | 7369 |
| 2 JONES  | SPOOKY   | 7370 |
| 3 MILLER | SALESMAN | 7371 |

4 SCOTT ANALYST 7372

Все вернулось на исходную. А триггер **BFRDELTTRIG** провел что-то вроде каскадного удаления. Вот таким образом можно построить, достаточно сложные взаимосвязи таблиц друг с другом. Обеспечить некую правовую политику, для вашей БД! Хотя мой пример достаточно прост думаю для вас теперь более понятно, как строить бизнес правила для таблиц БД. К слову скажу, что менять псевдозапись **:new** в строковом триггере **AFTER** не имеет смысла, так как событие уже обработано. Менять псевдозапись **:new** возможно в строковом триггере **BEFORE**! А вот псевдозапись **:old** никогда не модифицируется, а только считывается. Для закрепления можете сами создать несколько таблиц в вашей схеме и связать их при помощи триггеров. При этом будьте внимательнее при использовании событий **BEFORE** и **AFTER**! Удачи! :)

---

## Шаг 115 - PL/SQL - Триггеры - ЧАСТЬ III - условие WHERE

Как ни странно, но триггеры довольно объемный материал! Продолжаем. Давайте рассмотрим такой момент создания триггера БД, кода в нем применяется условие **WHERE**! С его помощью можно заставить триггер работать - по условию! Само условие **WHERE** в триггере применимо к типу строчных триггеров. Давайте, рассмотрим, как это реализуется на практике. Но предварительно добавим по одному полю в наши таблички из прошлых шагов - вот так:

```
ALTER TABLE TSTTRIG ADD COST NUMBER
/
```

```
ALTER TABLE TSTSV ADD ITOG NUMBER
/
```

Получаем:

```
SQL> ALTER TABLE TSTTRIG ADD COST NUMBER
2 /
```

Таблица изменена.

```
SQL> ALTER TABLE TSTSV ADD ITOG NUMBER
2 /
```

Таблица изменена.

В этих полях мы будем хранить наши данные для последующего использования. Далее давайте добавим данные пока только в таблицу **TSTTRIG**, с помощью операторов **UPDATE** вот так:

```
UPDATE TSTTRIG
 SET COST = 30532
WHERE NM = 'SMITH'
/
```

```
UPDATE TSTTRIG
 SET COST = 80478
WHERE NM = 'JONES'
/
```

```
UPDATE TSTTRIG
 SET COST = 20785
WHERE NM = 'MILLER'
/
```

```
UPDATE TSTTRIG
 SET COST = 10205
WHERE NM = 'SCOTT'
/
```

```
COMMIT
/
```

Получаем:

```
SQL> UPDATE TSTTRIG
2 SET COST = 30532
3 WHERE NM = 'SMITH'
4 /
```

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
2 SET COST = 80478
3 WHERE NM = 'JONES'
4 /
```

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
2 SET COST = 20785
3 WHERE NM = 'MILLER'
4 /
```

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
2 SET COST = 10205
3 WHERE NM = 'SCOTT'
4 /
```

1 строка обновлена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Итак, поле и данные в них добавлены. Давайте посмотрим, как создавать триггер с условием. Запишем вот такую конструкцию:

```
CREATE OR REPLACE TRIGGER WHENTRG
 BEFORE INSERT OR UPDATE OF COST ON TSTTRIG
 FOR EACH ROW
 WHEN (new.COST > 10000)

DECLARE

BEGIN

UPDATE TSTSV
 SET TSTSV.ITOG = :new.COST + :old.COST
WHERE TSTSV.CONC = :old.ID;

END WHENTRG;
/
```

---

Обратите внимание на наличие строки **OF COST ON TSTTRIG** - здесь определяется поле, на которое устанавливаем условие триггера и собственно само условие **WHEN (new.COST > 10000)** - обратите внимание, что псевдозапись **new** записана как - **new**, а не **:new** ! Это важно! В условии псевдозаписи записываются БЕЗ ДВОЕТОЧИЯ! Запомните! Двоеточие применяется только в теле триггера! Условие можно строить и по другому, так как нужно вам. Данный триггер производит довольно глупое действие, но зато наглядное! При вставке или изменении, он просто запоминает сумму чисел в поле **ITOG** таблички **TSTSV**! Это просто пример, но с его помощью вы сможете разобраться как это все работает! Итак, **SQL\*Plus** и компилируем:

```
SQL> CREATE OR REPLACE TRIGGER WHENTRG
 2 BEFORE INSERT OR UPDATE OF COST ON TSTTRIG
 3 FOR EACH ROW
 4 WHEN (new.COST > 10000)
 5
 6 DECLARE
 7
 8 BEGIN
 9
10 UPDATE TSTSV
11 SET TSTSV.ITOG = :new.COST + :old.COST
12 WHERE TSTSV.CONS = :old.ID;
13
14 END WHENTRG;
15 /
```

Триггер создан.

Прекрасно! Теперь поработаем с ним. Давайте теперь изменим одну из записей таблички **TSTTRIG** (запомнив, что ее старое содержимое было равно 10205) вот так:

```
UPDATE TSTTRIG
 SET COST = 20205
WHERE NM = 'SCOTT'
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE TSTTRIG
 2 SET COST = 20205
 3 WHERE NM = 'SCOTT'
 4 /
```

1 строка обновлена.

```
SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.



А, вот теперь давайте посмотрим, что изменилось в полях наших с вами таблиц:

```
SELECT * FROM MILLER.TSTTRIG
/
```

```
SELECT * FROM MILLER.TSTSV
/
```

Видим:

```
SQL> SELECT * FROM TSTTRIG
2 /
```

| ID NM       | ROD      | INRW       | COST  |
|-------------|----------|------------|-------|
| 7369 SMITH  | CLERK    | 17.02.2000 | 30532 |
| 7370 JONES  | MANAGER  | 02.04.2001 | 80478 |
| 7371 MILLER | SALESMAN | 20.03.2003 | 20785 |
| 7372 SCOTT  | ANALYST  | 09.12.2001 | 20205 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID IDD   | ROD      | CONS | ITOG  |
|----------|----------|------|-------|
| 1 SMITH  | CLERK    | 7369 | NULL  |
| 2 JONES  | MANAGER  | 7370 | NULL  |
| 3 MILLER | SALESMAN | 7371 | NULL  |
| 4 SCOTT  | ANALYST  | 7372 | 30410 |

Если применить элементарные знания математики за первый класс, то  $10205 + 20205 = 30410$ . Все верно! Как видим условие дало **TRUE** и триггер сработал. А что, если изменить запись **MILLER**-а вот так:

```
UPDATE TSTTRIG
SET COST = 5342
WHERE NM = 'MILLER'
/
```

```
COMMIT
/
```

Видим:

```
SQL> UPDATE TSTTRIG
2 SET COST = 5342
3 WHERE NM = 'MILLER'
4 /
```

1 строка обновлена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

А теперь давайте посмотрим, что содержат таблицы после изменения:

```
SQL> SELECT * FROM TSTTRIG
2 /
```

| ID NM       | ROD      | INRW       | COST  |
|-------------|----------|------------|-------|
| 7369 SMITH  | CLERK    | 17.02.2000 | 30532 |
| 7370 JONES  | MANAGER  | 02.04.2001 | 80478 |
| 7371 MILLER | SALESMAN | 20.03.2003 | 20785 |
| 7372 SCOTT  | ANALYST  | 09.12.2001 | 20205 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID IDD   | ROD      | CONS | ITOG  |
|----------|----------|------|-------|
| 1 SMITH  | CLERK    | 7369 | NULL  |
| 2 JONES  | MANAGER  | 7370 | NULL  |
| 3 MILLER | SALESMAN | 7371 | NULL  |
| 4 SCOTT  | ANALYST  | 7372 | 30410 |

То же самое! Верно, ведь если снова вспомнить первый класс, то  $5342 < 10000$ . Что дает в условии **WHERE** триггера **FALSE** и он не срабатывает! Что собственно и ожидалось! А что, если запись не изменять, а добавить? Что тогда будет происходить? Давайте посмотрим. Добавим вот такую запись:

```
INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
VALUES ('BOB', 'DUMMY', TO_DATE('9-11-1989', 'DD-MM-YYYY'), 24734)
/
COMMIT
/
```

Получаем (помните? С прошлых шагов, все ранее созданные триггера уже сделали свое дело!):

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
2 VALUES ('BOB', 'DUMMY', TO_DATE('9-11-1989', 'DD-MM-YYYY'), 24734)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое табличек после добавления записи:

```
SQL> SELECT * FROM TSTTRIG
2 /
```

| ID NM       | ROD      | INRW       | COST  |
|-------------|----------|------------|-------|
| 8000 BOB    | DUMMY    | 09.11.1989 | 24734 |
| 7369 SMITH  | CLERK    | 17.02.2000 | 30532 |
| 7370 JONES  | MANAGER  | 02.04.2001 | 80478 |
| 7371 MILLER | SALESMAN | 20.03.2003 | 20785 |
| 7372 SCOTT  | ANALYST  | 09.12.2001 | 20205 |

```
SQL> SELECT * FROM MILLER.TSTSV
2 /
```

| ID IDD   | ROD      | CONS | ITOG  |
|----------|----------|------|-------|
| 5 BOB    | DUMMY    | 8000 | NULL  |
| 1 SMITH  | CLERK    | 7369 | NULL  |
| 2 JONES  | MANAGER  | 7370 | NULL  |
| 3 MILLER | SALESMAN | 7371 | NULL  |
| 4 SCOTT  | ANALYST  | 7372 | 30410 |

Как видим запись **(8000, BOB, DUMMY, 09.11.1989, 24734)** - добавлена в таблицу **TSTTRIG**. Триггеры из прошлых шагов так же добавили запись - **(5, BOB, DUMMY, 8000, NULL)**. Но вот почему поле **ITOG** таблицы **TSTSV** содержит **NULL**? Странно, не правда ли? Ведь условие  $24734 > 10000$  дает **TRUE**! Да, триггер **WHENTRG** свою работу выполнил! Но дело в том, что псевдозапись **:new** для триггеров по **INSERT** не определена, то есть содержит - **NULL**! Отсюда **NULL + 24734 = NULL**! Что собственно и ожидалось! Вспоминаем тройственную логику, если кто забыл - ["Шар 17 - Составные операторы в условии WHERE"](#). Пробуйте! :)

## Шаг 116 - PL/SQL - Триггеры - ЧАСТЬ IV - предикаты

Продолжаем работу с триггерами и всем, что с ними связано! В триггерах БД **Oracle** возможно применение логических операторов - так называемых предикатов. Они имеют следующие определения **INSERTING**, **UPDATING**, **DELETING**. Это некие внутренние переменные среды **Oracle**, которые в зависимости от воздействующего на таблицу оператора **DML** принимают одно из значений **TRUE** или **FALSE**. С их помощью можно значительно сэкономить при написании кода, в чем вы в дальнейшем убедитесь и не плодить слишком большое количество объектов БД.

Кратко их можно описать вот так:

| Предикат  | Принимаемое значение                                                 |
|-----------|----------------------------------------------------------------------|
| INSERTING | TRUE если, активизирующий оператор INSERT. FALSE в противном случае. |
| UPDATING  | TRUE если, активизирующий оператор UPDATE. FALSE в противном случае. |
| DELETING  | TRUE если, активизирующий оператор DELETE. FALSE в противном случае. |

Для полноты понимания давайте все рассмотрим на практическом примере! Так как это самый эффективный способ что-либо запомнить! Итак, создадим некий аудит нашей таблички из [прошлого шага](#) **TSTTRIG**. Создадим таблицу вида:

```
CREATE TABLE MYAUDIT
(
 POLZ VARCHAR2(15),
 VIZM DATE,
 OPER VARCHAR2(20),
 NZAP NUMBER,
 HIST VARCHAR2(50)
)
```

Получаем:

```
SQL> CREATE TABLE MYAUDIT
2 (
3 POLZ VARCHAR2(15),
4 VIZM DATE,
5 OPER VARCHAR2(20),
6 NZAP NUMBER,
7 HIST VARCHAR2(50)
8)
9 /
```

Таблица создана.

В ней мы будем хранить данные, которые будут меняться во время нашего примера. Далее давайте применим на практике предикаты **INSERTING**, **UPDATING**, **DELETING** - для написания одного, но очень эффективного триггера вот такого вида:

```
CREATE OR REPLACE TRIGGER AUDT_TSTTRIG
```

```
BEFORE INSERT OR UPDATE OR DELETE ON TSTTRIG
FOR EACH ROW

DECLARE

TIP VARCHAR2(10);

BEGIN

IF INSERTING THEN
 TIP := 'INSERT';
ELSIF UPDATING THEN
 TIP := 'UPDATE';
ELSIF DELETING THEN
 TIP := 'DELETE';
END IF;

INSERT INTO MYAUDIT(MYAUDIT.POLZ, MYAUDIT.VIZM, MYAUDIT.OPER, MYAUDIT.NZAP,
MYAUDIT.HIST)
 VALUES (USER, SYSDATE, TIP, :new.ID, 'Old Name: '||:old.NM||' New Name: '||:new.NM);

END AUDT_TSTTRIG;
/
```

Получаем после компиляции:

```
SQL> CREATE OR REPLACE TRIGGER AUDT_TSTTRIG
 2 BEFORE INSERT OR UPDATE OR DELETE ON TSTTRIG
 3 FOR EACH ROW
 4
 5 DECLARE
 6
 7 TIP VARCHAR2(10);
 8
 9 BEGIN
10
11 IF INSERTING THEN
12 TIP := 'INSERT';
13 ELSIF UPDATING THEN
14 TIP := 'UPDATE';
15 ELSIF DELETING THEN
16 TIP := 'DELETE';
17 END IF;
18
19 INSERT INTO MYAUDIT(MYAUDIT.POLZ, MYAUDIT.VIZM, MYAUDIT.OPER, MYAUDIT.NZAP,
MYAUDIT.HIST)
20 VALUES (USER, SYSDATE, TIP, :new.ID, 'Old Name: '||:old.NM||' New Name: '||:new.NM);
21
22 END AUDT_TSTTRIG;
23 /
```

Триггер создан.

---

Данный триггер имеет временное действие "ДО"! Попробуем добавить запись в таблицу **TSTTRIG** вот так:

```
INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
 VALUES ('ALFRED', 'MANAGER', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
2 VALUES ('ALFRED', 'MANAGER', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое таблицы **MYAUDIT** применив запрос вида:

```
SQL> SELECT * FROM MYAUDIT
2 /

POLZ VIZM OPER NZAP HIST

MILLER 20.03.2004 INSERT NULL Old Name: "NULL" New Name: ALFRED
```

Здесь в строке **Old Name: "NULL" New Name: ALFRED** я поставил **"NULL"** чисто фигурально, чтобы было понятно. Сразу попутно запоминайте, что псевдозапись **:old** для **DML** оператора **INSERT** триггера типа **BEFORE** не определена! А вот псевдозапись **:new** для поля **ID**, так же еще не получила значения! Почему можете подумать сами! Вспомните для начала как оно вообще формируется? Далее поле **OPER** после сработки условия **.. IF INSERTING THEN ..** получаем, что была операция **INSERT**! Что собственно хорошо видно! Теперь давайте, попробуем изменить запись вот так:

```
UPDATE TSTTRIG
SET NM = 'ALF'
WHERE NM = 'ALFRED'
/

COMMIT
/
```

Получаем:

```
SQL> UPDATE TSTTRIG
2 SET NM = 'ALF'
```

```
3 WHERE NM = 'ALFRED'
4 /
```

1 строка обновлена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Альфред стал инопланетянином Альфом! Помните такого? Не важно! А, важно вот что:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ   | VIZM       | OPER   | NZAP | HIST                              |
|--------|------------|--------|------|-----------------------------------|
| MILLER | 20.03.2004 | INSERT | NULL | Old Name: "NULL" New Name: ALFRED |
| MILLER | 20.03.2004 | UPDATE | 8001 | Old Name: ALFRED New Name: ALF    |

Здесь строка **Old Name: ALFRED New Name: ALF** показывает, что псевдозаписи **:new** и **:old** применительно для оператора **UPDATE**, для триггера типа **BEFORE** определены! Кстати псевдозаписи **:new** так же можно изменить!!! Запоминайте! Здесь для поля **NZAP** получаем 8001, все верно! В данном случае в поле **OPER** после сработки условия **.. IF INSERTING THEN ..** получаем, что была операция **UPDATE**! Что собственно хорошо видно! Теперь давайте, попробуем удалить запись вот так:

```
DELETE FROM TSTTRIG
WHERE NM = 'ALF'
/
```

```
COMMIT
/
```

Получаем:

```
SQL> DELETE FROM TSTTRIG
2 WHERE NM = 'ALF'
3 /
```

1 строка удалена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим содержимое таблицы аудита:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ   | VIZM       | OPER   | NZAP | HIST                              |
|--------|------------|--------|------|-----------------------------------|
| MILLER | 20.03.2004 | INSERT | NULL | Old Name: "NULL" New Name: ALFRED |
| MILLER | 20.03.2004 | UPDATE | 8001 | Old Name: ALFRED New Name: ALF    |
| MILLER | 20.03.2004 | DELETE | NULL | Old Name: ALF New Name: "NULL"    |

Здесь строка **Old Name: ALF New Name: "NULL"** показывает, что псевдозаписи **:new** применительно для оператора **DELETE**, для триггера типа **BEFORE** не определена! Запоминайте! Здесь, для поля **NZAP** получаем **NULL**, это вам так же к слову подумать почему! В данном случае в поле **OPER** после сработки условия **.. IF INSERTING THEN ..** получаем, что была операция **DELETE**! Что собственно хорошо видно! На этом можно было бы поставить точку, но давайте проделаем еще кое-что! Создадим, вот такой триггер:

```
CREATE OR REPLACE TRIGGER AFT_AUDT_TSTTRIG
AFTER INSERT OR UPDATE OR DELETE ON TSTTRIG
FOR EACH ROW

DECLARE

TIP VARCHAR2(10);

BEGIN

IF INSERTING THEN
 TIP := 'INSERT';
ELSIF UPDATING THEN
 TIP := 'UPDATE';
ELSIF DELETING THEN
 TIP := 'DELETE';
END IF;

INSERT INTO MYAUDIT(MYAUDIT.POLZ, MYAUDIT.VIZM, MYAUDIT.OPER, MYAUDIT.NZAP,
MYAUDIT.HIST)
VALUES (USER, SYSDATE, TIP, :new.ID, 'Old Name: '||:old.NM||' New Name: '||:new.NM);

END AFT_AUDT_TSTTRIG;
/
```

Как видно я поменял его имя и сменил контент времени на **AFTER**! Давайте проделаем все еще раз, но при этом триггер **AUDT\_TSTTRIG** заблокируем вот такой командой:

```
ALTER TRIGGER AUDT_TSTTRIG DISABLE
/
```

Получаем:

```
SQL> ALTER TRIGGER AUDT_TSTTRIG DISABLE
2 /
```

Триггер изменен.

А, так же давайте очистим нашу табличку аудита от старых данных:



```
DELETE FROM MYAUDIT
/
```

Видим:

```
SQL> DELETE FROM MYAUDIT
2 /
```

3 строк удалено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Теперь давайте сделаем все сначала. Снова добавляем запись:

```
INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
 VALUES ('ALFRED', 'MANAGER', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
2 VALUES ('ALFRED', 'MANAGER', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Смотрим, что получилось в таблице **MYAUDIT**:

```
SELECT * FROM MYAUDIT
/
```

Видим:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ | VIZM | OPER | NZAP | HIST |
|------|------|------|------|------|
|------|------|------|------|------|

|        |            |        |      |                                   |
|--------|------------|--------|------|-----------------------------------|
| MILLER | 20.03.2004 | INSERT | 8003 | Old Name: "NULL" New Name: ALFRED |
|--------|------------|--------|------|-----------------------------------|

Все почти так же, но теперь псевдозапись **:new** для поля **ID** уже получила значение и его видно! Остальное не изменилось в части действия триггера. Изменим запись:

```
UPDATE TSTTRIG
SET NM = 'ALF'
WHERE NM = 'ALFRED'
/

COMMIT
/
```

Получим:

```
SQL> UPDATE TSTTRIG
2 SET NM = 'ALF'
3 WHERE NM = 'ALFRED'
4 /
```

1 строка обновлена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Снова смотрим табличку **MYAUDIT**:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ   | VIZM       | OPER   | NZAP | HIST                              |
|--------|------------|--------|------|-----------------------------------|
| MILLER | 20.03.2004 | INSERT | 8003 | Old Name: "NULL" New Name: ALFRED |
| MILLER | 20.03.2004 | UPDATE | 8003 | Old Name: ALFRED New Name: ALF    |

Здесь все по старому, как и в прошлый раз. Удалим запись:

```
DELETE FROM TSTTRIG
WHERE NM = 'ALF'
/

COMMIT
/
```

Получаем:

```
SQL> DELETE FROM TSTTRIG
2 WHERE NM = 'ALF'
3 /
```

1 строка удалена.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Снова смотрим табличку **MYAUDIT**:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ   | VIZM       | OPER   | NZAP | HIST                              |
|--------|------------|--------|------|-----------------------------------|
| MILLER | 20.03.2004 | INSERT | 8003 | Old Name: "NULL" New Name: ALFRED |
| MILLER | 20.03.2004 | UPDATE | 8003 | Old Name: ALFRED New Name: ALF    |
| MILLER | 20.03.2004 | DELETE | NULL | Old Name: ALF New Name: "NULL"    |

Здесь все, так же как и прошлый раз. Теперь я думаю, вы не запутаетесь в трех соснах **AFTER**, **BEFORE**, **new**, **old**! Что и когда нужно использовать и как! В ["Шаг 111 - PL/SQL - Триггеры таблиц БД, операторный триггер"](#) я не указал как активировать триггер после его деактивации! Собственно почти так же! Вот такой командой:

```
ALTER TRIGGER AUDT_TSTTRIG ENABLE
/
```

Триггер **AUDT\_TSTTRIG** снова активен:

```
SQL> ALTER TRIGGER AUDT_TSTTRIG ENABLE
2 /
```

Триггер изменен.

А вот теперь давайте добавим еще запись - вот так:

```
INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
VALUES ('MALKOVISH', 'ACTORS', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
/

COMMIT
/
```

Получаем:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW, COST)
2 VALUES ('MALKOVISH', 'ACTORS', TO_DATE('18-12-2002', 'DD-MM-YYYY'), 40967)
3 /
```

1 строка создана.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

А, вот теперь посмотрите на табличку **MYAUDIT**:

```
SQL> SELECT * FROM MYAUDIT
2 /
```

| POLZ   | VIZM       | OPER   | NZAP | HIST                                 |
|--------|------------|--------|------|--------------------------------------|
| MILLER | 20.03.2004 | INSERT |      | Old Name: "NULL" New Name: MALKOVISH |
| MILLER | 20.03.2004 | INSERT | 8004 | Old Name: "NULL" New Name: MALKOVISH |
| MILLER | 20.03.2004 | DELETE |      | Old Name: ALF New Name: "NULL"       |
| MILLER | 20.03.2004 | INSERT | 8003 | Old Name: "NULL" New Name: ALFRED    |
| MILLER | 20.03.2004 | UPDATE | 8003 | Old Name: ALFRED New Name: ALF       |

Хорошо видно как оба триггера отработали! И именно так как мы рассматривали с вами ранее! По сути, эти два триггера заменяют шесть (!) вот вам экономия кода! Хотя иногда не мешает поработать ручками! Советую все это прочесть еще раз и запомнить! Удачи! :)

## Шаг 117 - PL/SQL - Триггеры - ЧАСТЬ V - mutating table

Так как триггеры БД довольно обширная тема, продолжаем разбирать ее далее. Давайте рассмотрим вариант, когда триггер работает с активизирующей его таблицей. Если говорить о работе триггеров таблиц, то как оказывается, триггер может обращаться не ко всем таблицам и столбцам этих таблиц. Для того, чтобы определить к каким таблицам возможен доступ, необходимо понимать, что такое изменяющиеся и ограничивающие таблицы. Изменяющиеся таблица (**mutating table**) - это именно та таблица, которая в данный момент модифицируется оператором **DML**! Для триггера это та таблица, для которой он был создан! Так же и те таблицы, которые обновляются в результате реализации или действия ссылочной целостности. Таблицы выполняющие каскадные удаления - **DELETE CASCADE**, так же являются изменяющимися. Ограничивающая таблица (**constraining table**) - это таблица, информация которой может быть считана при реализации ограничений ссылочной целостности. Вот такие трудности могут быть на пути создания триггеров БД. Давайте рассмотрим пример "неверного" триггера БД. Запишем вот такой триггер для таблицы **TSTTRIG**:

```
CREATE OR REPLACE TRIGGER ERRTRIG
AFTER INSERT ON TSTTRIG
FOR EACH ROW

BEGIN

UPDATE TSTTRIG
SET ROD = LOWER(ROD)
WHERE ID = :old.ID;

END ERRTRIG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER ERRTRIG
2 AFTER INSERT ON TSTTRIG
3 FOR EACH ROW
4
5 BEGIN
6
7 UPDATE TSTTRIG
8 SET ROD = LOWER(ROD)
9 WHERE ID = :old.ID;
10
11 END ERRTRIG;
12 /
```

Триггер создан.

Здесь для простоты примера я произвожу модификацию столбца **ROD** после вставки строки. Пробуем добавить, запись в таблицу **TSTTRIG**:

```
INSERT INTO TSTTRIG (NM, ROD, INRW)
VALUES ('BOB', 'DUMMY', TO_DATE('18-12-2004', 'DD-MM-YYYY'))
```

/

Получаем:

```
SQL> INSERT INTO TSTTRIG (NM, ROD, INRW)
2 VALUES ('BOB', 'DUMMY', TO_DATE('18-12-2004', 'DD-MM-YYYY'))
3 /
INSERT INTO TSTTRIG (NM, ROD, INRW)
 *
```

ошибка в строке 1:

ORA-04091: таблица MILLER.TSTTRIG изменяется, триггер/функция может не заметить это

ORA-06512: на "MILLER.ERRTRIG", line 5

ORA-04088: ошибка во время выполнения триггера 'MILLER.ERRTRIG'

Как видно сразу три типа ошибки вызывает эта операция! Но основная из них это **ORA-04091!** То есть для триггера **MILLER.TSTTRIG** таблица **TSTTRIG** является изменяющейся и модифицировать ее с помощью операторов **DML** он не может! Можете удалить триггер **MILLER.TSTTRIG**, чтобы он нам не мешал. Как это делать я думаю вы знаете! Итак, подведем черту.

**SQL** - операторы в теле триггера не могут:

1. Считывать или модифицировать информацию, любой таблицы изменяющейся в результате выполнения активизирующего оператора. В число таких таблиц входит и сама активизирующая таблица.
2. Считывать или модифицировать информацию столбца первичного ключа, уникальных столбцов и столбцов внешних ключей таблицы, являющейся ограничивающей по отношению к изменяющейся таблице. (уфф .. запутанная формулировка! Но верная!).

Заметим, что эти правила верны для строковых триггеров. Для операторных триггеров они применимы только в тех случаях, когда последние активизируются в результате выполнения операции каскадного удаления информации. Следует так же сказать, что оператор **INSERT** воздействующий только на одну строку (хм .. ну естественно он же ее добавляет - ведь так? :) ) для строковых триггеров **BEFORE** и **AFTER** работающих с этой строкой активизирующая таблица как ни странно не является изменяющейся! Это единственная ситуация когда строковый триггер может считывать или модифицировать информацию активизирующей таблицы. Но! Для таких операторов как:

**INSERT INTO таблица SELECT .....**

активизирующая таблица всегда является изменяющейся, даже если в подзапросе возвращается только одна строка! Вот так бывает много разных ситуаций, про которые не следует забывать, а просто применять их на практике при создании триггеров для таблиц. А, что если вам все же необходимо получать данные в триггере от самой иницирующей таблицы? Тогда можно поступить следующим образом. Создать два триггера, один операторный другой строчный и производить все необходимые действия с их помощью. Давайте попробуем это проиллюстрировать, но здесь вам нужно будет кое-что вспомнить, если забыли обращайтесь к прошлым шагам там все есть! Итак, начнем с написания такого пакета:

**CREATE OR REPLACE PACKAGE TrigTest IS**

```
TYPE m_ID IS TABLE OF TSTTRIG.ID%TYPE
INDEX BY BINARY_INTEGER;

TYPE m_NM IS TABLE OF TSTTRIG.NM%TYPE
INDEX BY BINARY_INTEGER;

V_m_ID m_ID;
V_m_NM m_NM;
V_num BINARY_INTEGER := 0;

END TrigTest;
/
```

Получаем:

```
SQL> CREATE OR REPLACE PACKAGE TrigTest IS
 2
 3 TYPE m_ID IS TABLE OF TSTTRIG.ID%TYPE
 4 INDEX BY BINARY_INTEGER;
 5
 6 TYPE m_NM IS TABLE OF TSTTRIG.NM%TYPE
 7 INDEX BY BINARY_INTEGER;
 8
 9 V_m_ID m_ID;
 10 V_m_NM m_NM;
 11 V_num BINARY_INTEGER := 0;
 12
 13 END TrigTest;
 14 /
```

Пакет создан.

Здесь мы создали две таблицы **V\_m\_ID** и **V\_m\_NM** и переменную **V\_num** для обращения к записям таблиц. Все достаточно не сложно! Теперь создадим такой строчный триггер:

```
CREATE OR REPLACE TRIGGER FIXTRG
BEFORE INSERT OR UPDATE OF NM ON TSTTRIG
FOR EACH ROW

BEGIN

TrigTest.V_num := TrigTest.V_num + 1;
TrigTest.V_m_ID(TrigTest.V_num) := :new.ID;
TrigTest.V_m_NM(TrigTest.V_num) := :new.NM;

END FIXTRG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER FIXTRG
 2 BEFORE INSERT OR UPDATE OF NM ON TSTTRIG
```

---

```
3 FOR EACH ROW
4
5 BEGIN
6
7 TrigTest.V_num := TrigTest.V_num +1;
8 TrigTest.V_m_ID(TrigTest.V_num) := :new.ID;
9 TrigTest.V_m_NM(TrigTest.V_num) := :new.NM;
10
11 END FIXTRG;
12 /
```

Триггер создан.

Этот триггер фиксирует значения двух полей таблицы **TSTTRIG** в наших пакетных таблицах **V\_m\_ID** и **V\_m\_NM**. Далее создаем вот такой операторный триггер:

```
CREATE OR REPLACE TRIGGER OPERTRG
BEFORE INSERT OR UPDATE OF NM ON TSTTRIG

DECLARE

V_CNT NUMBER;
V_COUNT CONSTANT NUMBER := 10;

BEGIN

SELECT COUNT(*) INTO V_CNT
 FROM TSTTRIG;

IF(TrigTest.V_num = V_COUNT) THEN
 TrigTest.V_m_ID(TrigTest.V_num) := V_CNT;
 TrigTest.V_m_NM(TrigTest.V_num) := 'STOP';
 TrigTest.V_num := 0;
END IF;

END OPERTRG;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER OPERTRG
2 BEFORE INSERT OR UPDATE OF NM ON TSTTRIG
3
4 DECLARE
5
6 V_CNT NUMBER;
7 V_COUNT CONSTANT NUMBER := 10;
8
9 BEGIN
10
11 SELECT COUNT(*) INTO V_CNT
```



```
12 FROM TSTTRIG;
13
14 IF(TrigTest.V_num = V_COUNT) THEN
15 TrigTest.V_m_ID(TrigTest.V_num) := V_CNT;
16 TrigTest.V_m_NM(TrigTest.V_num) := 'STOP';
17 TrigTest.V_num := 0;
18 END IF;
19
20
21 END OPERTRG;
22 /
```

Триггер создан.

Здесь саму идею я высосал из пальца, собственно триггер ничего путного не производит, а только фиксирует количество записей в таблице **TSTTRIG** в пакетных таблицах (запутался совсем в таблицах)! Но здесь явно видно, что триггер **OPERTRG** делает запрос к иницилирующей таблице и это ему сходит с рук! Что собственно и требовалось показать! Сами можете придумать что-то более путное, но моя задача состоит в том, чтобы Вы поняли как это все происходит или запутались окончательно! Но, думаю сможете все это разобрать! Теперь давайте попробуем поизменять поля таблицы **TSTTRIG** два три раза:

```
UPDATE TSTTRIG
SET NM = 'ALF'
WHERE NM = 'BOB'
/
```

```
UPDATE TSTTRIG
SET NM = 'BOB'
WHERE NM = 'ALF'
/
```

```
UPDATE TSTTRIG
SET NM = 'BUBER'
WHERE NM = 'SCOTT'
/
```

```
UPDATE TSTTRIG
SET NM = 'SCOTT'
WHERE NM = 'BUBER'
/
```

```
COMMIT
/
```

Получаем:

```
SQL> UPDATE TSTTRIG
2 SET NM = 'ALF'
3 WHERE NM = 'BOB'
4 /
```

---

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
 2 SET NM = 'BOB'
 3 WHERE NM = 'ALF'
 4 /
```

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
 2 SET NM = 'BUBER'
 3 WHERE NM = 'SCOTT'
 4 /
```

1 строка обновлена.

```
SQL> UPDATE TSTTRIG
 2 SET NM = 'SCOTT'
 3 WHERE NM = 'BUBER'
 4 /
```

1 строка обновлена.

```
SQL> COMMIT
 2 /
```

Фиксация обновлений завершена.

А затем напишем вот такой неименованный блок и посмотрим, что он нам выдаст по части содержимого пакетных табличек:

```
SET SERVEROUTPUT ON

DECLARE

BEGIN

FOR V_LOOP IN 1..TrigTest.V_m_NM.COUNT LOOP

DBMS_OUTPUT.enable;
DBMS_OUTPUT.put_line(TO_CHAR(TrigTest.V_m_ID(V_LOOP)));
DBMS_OUTPUT.put_line(TrigTest.V_m_NM(V_LOOP));

END LOOP;

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
```

---

```
SQL> DECLARE
2
3 BEGIN
4
5 FOR V_LOOP IN 1..TrigTest.V_m_NM.COUNT LOOP
6
7 DBMS_OUTPUT.enable;
8 DBMS_OUTPUT.put_line(TO_CHAR(TrigTest.V_m_ID(V_LOOP)));
9 DBMS_OUTPUT.put_line(TrigTest.V_m_NM(V_LOOP));
10
11 END LOOP;
12
13 END;
14 /
8041
ALF
8041
BOB
7372
BUBER
7372
SCOTT
8041
ALF
8041
BOB
7372
BUBER
7372
SCOTT
```

Процедура PL/SQL успешно завершена.

Вот здесь ясно видно, что вся работа триггеров прошла успешно и больше здесь комментировать нечего! Думаю, теперь вам ясно, как избежать ошибок при написании триггеров при использовании изменяющихся таблиц и основные правила работы с ними. Если что-то не совсем ясно, пишите, буду рад ответить!

---

## Шаг 118 - PL/SQL - Триггеры - ЧАСТЬ VI - системные триггеры

Уложились в голове все формулировки предыдущих шагов? Думаю да! Теперь самое время для завершающего этапа в плане изучения триггеров. Осталось рассмотреть, так называемые системные триггеры. Итак! Как было сказано выше триггеры **DML** срабатывают на события **DML** или вместо них! А, именно на операторы **INSERT, UPDATE, DELETE**. Но это еще не все события БД, на основе которых можно писать триггеры. В БД существует два основных вида событий, на которые активизируются системные триггеры. А, вернее сказать на события **DDL** или собственно самой БД. К событиям **DDL** относятся операторы **CREATE, DROP, ALTER**. А вот к событиям базы данных - запуск останов сервера, регистрация отключение пользователя БД, ошибка сервера. Рассмотрим синтаксис создания системного триггера. А именно:

```
----- CREATE OR REPLACE TRIGGER [схема.]имя_триггера -----
----- {BEFORE | AFTER} -----
----- {список_событий_DDL | список_событий_базы_данных} -----
----- ON {DATABASE | [схема.]SCHEMA} -----
----- конструкция_REFERENCING -----
----- [условие_WHEN] -----
----- тело триггера; -----
```

Где:

- **список\_событий\_DDL** - одно или несколько событий **DDL** (разделяемых ключевым словом **OR**)
- **список\_событий\_базы\_данных** - одно или несколько событий БД (разделяемых ключевым словом **OR**)

Приведем в виде таблички события и их обработку для БД:

| Событие     | Разрешенное время выполнения | Описание                                                                                                              |
|-------------|------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| STARTUP     | AFTER                        | Активизируется после запуска экземпляра                                                                               |
| SHUTDOWN    | BEFORE                       | Активизируется при остановке экземпляра. Для заметки, это событие не активизирует триггер, если останов БД аварийный! |
| SERVERERROR | AFTER                        | Активизируется при возникновении ошибки.                                                                              |
| LOGON       | AFTER                        | Активизируется после успешного соединения пользователя с базой данных.                                                |
| LOGOFF      | BEFORE                       | Активизируется в начале отключения пользователя.                                                                      |
| CREATE      | BEFORE, AFTER                | Активизируется до и после создания объекта схемы.                                                                     |
| DROP        | BEFORE, AFTER                | Активизируется до, и после удаления объекта схемы.                                                                    |
| ALTER       | BEFORE, AFTER                | Активизируется до и после изменения объекта схемы.                                                                    |

Сразу следует запомнить, что создание триггеров БД требует системных привилегий администратора БД! И создавать их следует внимательно и без спешки! Иначе последствия могут быть очень печальными! Либо вы больше не сможете получить логин к БД, либо вообще потеряете экземпляр! Возможно все! Все действия с полномочиями схем **SYS** и **SYSTEM** следует выполнять, имея крепкие и прочные знания и нервы! К чему я собственно вас и веду! Итак,

системные триггеры могут создаваться на уровне схемы (**SCHEMA**) или уровне БД (**DATABASE**)! Триггеры БД реагируют на события в БД - **DATABASE**, а схемы - **SCHEMA**, в которой происходит событие. Что ж, готовы! Хотите попробовать создать системный аудит! Кто боится пусть не делает! Итак, запускаем \***SQL/Plus** с правами **SYSTEM**. Создадим табличку для фиксации регистрации пользователей на уровне БД:

```
CREATE TABLE SYSTEM.AUDTBASE
(
 NZAP NUMBER,
 POLZ VARCHAR2(20),
 TMIN DATE,
 OPER VARCHAR2(50)
)
/
```

Получаем:

```
SQL> CREATE TABLE SYSTEM.AUDTBASE
2 (
3 NZAP NUMBER,
4 POLZ VARCHAR2(20),
5 TMIN DATE,
6 OPER VARCHAR2(50)
7)
8 /
```

Таблица создана.

Теперь создадим триггер уровня БД:

```
CREATE OR REPLACE TRIGGER FIXUSERIN
AFTER LOGON ON DATABASE

BEGIN

INSERT INTO SYSTEM.AUDTBASE(NZAP, POLZ, TMIN, OPER)
VALUES(1, USER, SYSDATE, 'UserIsLog(off)');

END FIXUSERIN;
/
```

Получаем:

```
SQL> CREATE OR REPLACE TRIGGER FIXUSERIN
2 AFTER LOGON ON DATABASE
3
4 BEGIN
5
6 INSERT INTO SYSTEM.AUDTBASE(NZAP, POLZ, TMIN, OPER)
7 VALUES(1, USER, SYSDATE, 'UserIsLog(off)');
8
9 END FIXUSERIN;
```

---

10 /

Триггер создан.

Просто, но со вкусом, теперь запустите еще один **\*SQL/Plus** и посмотрите содержимое таблицы в первом сеансе:

```
SELECT NZAP, POLZ, TO_CHAR(TMIN,'DD.MM.YYYY HH24:MI:SS'), OPER FROM SYSTEM.AUDTBASE
/
```

У меня получилось примерно следующее:

```
SQL> SELECT NZAP, POLZ, TO_CHAR(TMIN,'DD.MM.YYYY HH24:MI:SS'), OPER FROM
SYSTEM.AUDTBASE
2 /
```

| NZAP | POLZ   | TO_CHAR(TMIN,'DD.MM.YYYYHH24:M | OPER           |
|------|--------|--------------------------------|----------------|
| 1    | SYS    | 22.03.2003 16:35:50            | UserIsLog(off) |
| 1    | SYS    | 22.03.2003 16:35:50            | UserIsLog(off) |
| 1    | SYS    | 22.03.2003 16:37:38            | UserIsLog(off) |
| 1    | MILLER | 22.03.2003 16:38:44            | UserIsLog(off) |
| 1    | MILLER | 22.03.2003 16:38:44            | UserIsLog(off) |
| 1    | MILLER | 22.03.2003 16:38:46            | UserIsLog(off) |

6 строк выбрано.

У вас может быть по другому, но суть остается той же! Вот такие дела! Уже получается что-то серьезное! Для полноты картины пока удалите триггер **FIXUSERIN** и табличку **AUDTBASE**, только лучше сначала триггер, а потом таблицу! То же можно проделать и для схемы **MILLER** примерно так:

```
CREATE TABLE MILLER.AUDTBASE
(
 NZAP NUMBER,
 POLZ VARCHAR2(20),
 TMIN DATE,
 OPER VARCHAR2(50)
)
/
```

```
CREATE OR REPLACE TRIGGER FIXUSERIN
AFTER LOGON ON SCHEMA
```

```
BEGIN
```

```
INSERT INTO SYSTEM.AUDTBASE(NZAP, POLZ, TMIN, OPER)
VALUES(1, USER, SYSDATE, 'UserIsLog(off));
```

```
END FIXUSERIN;
/
```

Прodelайте все сами и убедитесь, что события будут теперь срабатывать только в вашей конкретной схеме, а не как было ранее. Так же для информации замечу, что триггеры **STARTUP** и **SHUTDOWN** имеют смысл только на уровне БД, хотя их можно создать и в конкретной схеме, но активизироваться они не будут! Кроме того, для системных триггеров существует ряд атрибутивных функций. Помните **INSERTING** и т. д. эти функции имеют тот же смысл! Вот их описание:

| Атрибутивная функция   | Тип данных   | Системное событие для которых применяется | Описание                                                                                                                      |
|------------------------|--------------|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| SYSEVENT               | VARCHAR2(20) | Все события                               | Возвращает системное событие активизировавшее триггер                                                                         |
| INSTANCE_NUM           | NUMBER       | Все события                               | Возвращает номер текущего экземпляра                                                                                          |
| DATABASE_NAME          | VARCHAR2(50) | Все события                               | Возвращает имя текущей БАЗЫ ДАННЫХ.                                                                                           |
| SERVER_ERROR           | NUMBER       | SERVERERROR                               | Принимает один числовой аргумент. Возвращает ошибку на позиции в стеке ошибок, указанной аргументом. Вершина стека позиция -1 |
| IS_SERVERERROR         | BOOLEAN      | SERVERERROR                               | Принимает номер ошибки в качестве аргумента и возвращает TRUE, если указанная ошибка Oracle присутствует в стеке ошибок.      |
| LOGIN_USER             | VARCHAR2(20) | Все события                               | Возвращает идентификатор пользователя активизирующего триггер.                                                                |
| DICTIONARY_OBJ_TYPE    | VARCHAR2(30) | CREATE, DROP, ALTER                       | Возвращает тип объекта словаря, над которым выполнялась операция DDL активизировавшая триггер.                                |
| DICTIONARY_OBJ_NAME    | VARCHAR2(30) | CREATE, DROP, ALTER                       | Возвращает имя объекта словаря, над которым выполнялась операция DDL активизировавшая триггер.                                |
| DICTIONARY_OBJ_OWNER   | VARCHAR2(30) | CREATE, DROP, ALTER                       | Возвращает владельца того объекта словаря, над которым выполнялась операция DDL активизировавшая триггер.                     |
| DES_ENCRYPTED_PASSWORD | VARCHAR2(30) | CREATE USER, ALTER USER                   | Возвращает зашифрованный в стандарте DES пароль создаваемого или изменяемого пользователя.                                    |

Ура! Наконец тему триггеры мы закрываем! Я опустил пока тематику триггеров типа **INSTEAD OF** для представлений, но если таковая необходимость есть, я могу их описать и привести примеры. Или мы еще вернемся к ним позже! А пока можете закреплять материал. Например, вам задание напишите два, три системных триггера уровня схемы, (с уровнем БД советую быть осторожнее) применив некоторые предикаты описанные выше. Например, я считаю очень

интересным атрибутивную функцию **DES\_ENCRYPTED\_PASSWORD** - кто знает алгоритмы **DES** можете потом расшифровать! Пробуйте и спрашивайте, если что-то до конца не ясно! Удачи! :)



## Шаг 119 – Табличные пространства - Системы БД OLTP

Давайте после множества примеров **PL/SQL** - обратимся, к такому понятию БД **Oracle**, как табличное пространство. В [шаге-23](#), мы немного рассмотрели этот вопрос, но только в начальной стадии. Теперь давайте познакомимся с этим понятием более детально. Помните наш запрос:

```
SQL> SELECT tablespace_name, file_name, bytes
2 FROM dba_data_files
3 ORDER BY tablespace_name, file_name
4 /
```

| TABSPACE_NAME  | FILE_NAME                            | BYTES     |
|----------------|--------------------------------------|-----------|
| INDX           | C:\ORACLE\ORADATA\PROBA\INDX01.DBF   | 2097152   |
| OEM_REPOSITORY | C:\ORACLE\ORADATA\PROBA\OEMREP01.DBF | 5242880   |
| RBS            | C:\ORACLE\ORADATA\PROBA\RBS01.DBF    | 26214400  |
| SYSTEM         | C:\ORACLE\ORADATA\PROBA\SYSTEM01.DBF | 146800640 |
| TEMP           | C:\ORACLE\ORADATA\PROBA\TEMP01.DBF   | 2097152   |
| USERS          | C:\ORACLE\ORADATA\PROBA\USERS01.DBF  | 3145728   |

6 rows selected

В данном случае мы видим, что наша с вами система, имеет шесть табличных пространств, каждое из которых содержится в своем файле данных, который в свою очередь расположен в системном каталоге **Oracle Server** - **C:\Oracle\ORADATA\proba**. Иначе, это можно описать как **C:\ORAHOME\ORADATA\SID**. По умолчанию при создании экземпляра БД используется именно такой путь к файлам данных табличных пространств. Если говорить прямо, то табличное пространство, это конкретный файл данных, расположенный в системном каталоге **Oracle**. А, в строгой формулировке - табличное пространство это логическая структура, которая используется для группировки данных с однотипными методами доступа. В табличное пространство может входить, один или несколько файлов данных. Но, как правило, один или несколько файлов данных не могут хранить два табличных пространства, а только одно!

Помните когда мы создавали пользователя (схему), мы с вами указывали где он будет хранить свои данные. Это и есть то самое табличное пространство. И вот здесь мы с вами подходим к очень интересному моменту. При создании любой БД, как правило, оговаривается ее тип, исходя из этого и создается сам экземпляр! Так вот в нашей учебной БД, вы этого не видели. Но, если бы мы пошли по более конструктивному пути, (мы еще это сделаем!) а, именно установили сам сервер **Oracle**, затем создали бы экземпляр и провели его настройку, то вы бы это все увидели. Итак, к чему я это все болтаю, а вот к чему!

БД **Oracle** содержит и поддерживает несколько типов при создании экземпляра. Но основные из них это **OLTP (On-Line Transaction Processing)** и **DSS (Decision Support System)**. Я понимаю, что это немного утомляет когда много теории, но знать это необходимо. Рассмотрим подробнее.

Рассмотрим систему **OLTP**. Данная система характеризуется высоким параллелизмом (большим количеством интерактивных пользователей) и как следствие высокой интенсивностью обновлений (то есть большое количество одновременно открытых транзакций). В таких системах, объем данных может значительно изменяться во времени как бы, то пусто, то густо! По этому, при создании табличных пространств (вот к чему я и вел!) и экстенгов в них для систем **OLTP** необходимо задавать максимально возможный размер. Так как табличные пространства с включенной опцией

динамического расширения может привести к тому, что производительность системы может значительно упасть! Вследствие этого возникает парадокс, что для достижения высокой производительности табличные пространства должны быть фиксированной длины и при этом превосходить по объему максимальный размер данных, обрабатываемых в нем. Так же для управления транзакциями необходимо иметь принципиальную возможность выполнения достаточного количества отмен и откатов. Для коротких транзакций небольшие сегменты отката, для длительной и объемной транзакции большие сегменты отката. Так же большее количество журналов регистрации транзакций и более продуманной системы построения экземпляра в целом с применением методов диагностики и настройки. Вообще по большому счету при работе с любой БД, администрирование и настройка БД, это каждодневный и нелегкий труд админа! Такая у него доля! Кстати ярким примером системы **OLTP**, может служить система, применяемая сегодня в супермаркетах, коих у нас великое множество (Эльдорадо (Москва) НК - Сити (Хабаровск) и т.д.). В единицу времени может списываться сотня наименований товаров, со склада с подсчетом общего баланса и в то же время сотни наименований могут быть приняты на учет, так как пришел груз и получатель его оприходует в тот же момент! Собственно это и есть кратко система **OLTP**.

Далее продолжим рассмотрение второго типа экземпляров БД, для более ясного усвоения изложенного! :)

---

## Шаг 120 - Табличные пространства - Системы БД DSS и другие

Продолжаем наш разбор по теме систем хранения информации в БД. Итак, системы **DSS**.

В системах данного типа, как правило, используются довольно большие и старые базы данных, которые обычно используют "только для чтения". По этому основным видом работы с ними является создание запросов на выборку. Как мы с вами упоминали ранее, для чтения блока **Oracle** следует считывать сразу же максимально возможное количество строк. При этом рекомендуется считывать максимально возможное количество смежных блоков, так как **DSS** - запросы, как правило, требуют полного сканирования таблицы БД, а эта операция по самой своей природе является последовательной как вы должны знать. Таким образом при работе с **DSS** экземплярами необходимо для параметров **DB\_BLOCK\_SIZE** и **DB\_FILE\_MULTIBLOCK\_READ\_COUNT** указывать максимально возможные значения. Так же в коде **DSS** приложений не следует использовать связующие переменные, эта рекомендация находится в прямом противоречии с **OLTP**-системами, где требовалось сократить накладные расходы (синтаксический анализ) связанный с работой приложений. Типичным примером **DSS** системы может служить любой портал библиотеки. Например, сайт большой энциклопедии "РОЛ" вот тут <http://www.krugosvet.ru>! Я, вообще поразился, как нужно было набраться терпения и все это сделать! Просто молодцы ребята! Я ей часто пользуюсь и вам рекомендую, очень полезно! Давайте проведем небольшое сравнение особенностей **OLTP** и **DSS** систем.

Для оптимизации работы **OLTP** и **DSS** систем, могут быть использованы параметры уровня блока **INITRANS** и **PCTFREE**. Так для **OLTP** систем значение параметра **PCTFREE**, следует устанавливать относительно не большим, так как каждый блок должен иметь свободное пространство для изменения и роста. Для **DSS** систем работающих на чтение значение параметра **PCTFREE**, следует устанавливать достаточно высоким, так как данные в блоках остаются преимущественно статичными. Для **OLTP** систем значение параметра **INITRANS** следует установить на относительно высоком уровне, поскольку увеличение количества параллельно работающих пользователей приведет к динамическому расширению, слота заголовка транзакции и последующей реорганизации на уровне блоков, которая может распространиться, вплоть до уровня экстентов! Так же для **DSS** систем существует значительная вероятность того, что параллельно работающие пользователи, будут находиться в состоянии ожидания из-из блокировки вызванной множеством транзакций. Следовательно для параметра **INITRANS** нужно установить значение на некотором среднем уровне, если изменения таблиц БД происходят не слишком часто. В противном случае необходимо оставить параметр по умолчанию.

Вот таким образом устроены наиболее часто применяемые типы хранилищ данных в БД **Oracle**. Существуют так же и другие типы, например **OLAP**, **VCDB** и так называемые пакетные типы. Какой тип хранилища выбрать, это ваше право, но укажу на момент анализа поставленной задачи и соответственно выбора типа экземпляра. Не стоит, например для работы со статистикой обтекаемости плоскости истребителя в разных режимах, продувки в аэродинамической трубе применять **DSS** тип хранилища. И т. д. Надеюсь теперь вам стало яснее как внутренне подразделяются экземпляры БД в целом и экземпляры БД **Oracle** в частности. Далее продолжим.

## Шаг 121 - Табличные пространства - создание, изменение, удаление

Теории было достаточно, теперь немного практики. Попробуем создавать табличные пространства и поработать с ними руками. Для ясности понимания я приведу основные команды БД, для манипулирования табличными пространствами, а затем мы все это опробуем на практике. Итак, создание табличного пространства. Для создания, удаления и изменения табличных пространств, пользователь должен иметь следующие привилегии - **CREATE TABLESPACE**, **DROP TABLESPACE**, **ALTER TABLESPACE**. Рассмотрим первую из них:

```
CREATE TABLESPACE -- (табличное_пространство) -- DATAFILE -- (спецификация_файла) -
(авторасширение)-*

--* -- MINIMUM -> EXTENT - (целое число) - К(килобайт)/М(мегабайт) --
-- LOGGING --
-- NOLOGGING --
-- DEFAULT STORAGE -
-- ONLINE --
-- OFFLINE --
-- PERMANENT --
-- TEMPORARY --
-- (конструкция_управления_экстентами) -- (;)
--*

*-- конструкция_максимального_размера --

-- MAXSIZE -- UNLIMITED -----
-- (целое число) - К(килобайт)/М(мегабайт) --

"конструкция_авторасширения"

-- AUTOEXTEND -- OFF
-- NEXT -- (целое число) - К(килобайт)/М(мегабайт) --
-- ON ----- (конструкция_максимального_размера)

"конструкция_управления_экстентами"

----- DICTIONARY -----
----- AUTOALLOCATE -----

-- EXTEND -- MANAGEMENT -- LOCAL ---- UNIFORM -- SIZE -- (целое число) -
К(килобайт)/М(мегабайт) --
```

Рассмотрим основные конструкции. С первого взгляда их здесь много, но это не столь важно, просто я привел полную конструкцию команды из руководства администратора, по этому не пугайтесь все будет понятно. Значок **-\*** и **\*-** в данном случае указывает, что строка продолжается как бы в одну строку. Начнем помаленьку:

- "**конструкция\_авторасширения**" - разрешает или запрещает авто расширение файла данных.  
**ON** разрешает и **OFF** запрещает соответственно.  
- "**конструкция\_максимального\_размера**" - указывает максимальное дисковое пространство, которое может быть выделено файлу данных. **UNLIMITED** - снимает ограничения на выделение дискового пространства файлу данных.

- **LOGGING/NOLGGING** - указывает атрибуты протоколирования, по умолчанию **LOGGING**.
- **DEFAULT STORAGE** - определяет для всех объектов создаваемых в табличном пространстве, значения параметров хранения по умолчанию.
- **ONLINE/OFFLINE** - определяет сразу после создания доступность табличного пространства.
- **PERMANENT/TEMPORARY** - постоянное или временное пространство для хранения объектов БД, по умолчанию **PERMANENT**.
- **"конструкция\_управления\_экстентами"** - определяет способ управления экстентами табличного пространства.
- **DICTIONARY** - установка по умолчанию, определяет управление по таблицам словаря данных.
- **LOCAL** - указывает на локальное управление экстентами.
- **AUTOALLOCATE** - управляется системой и пользователь не может влиять на размер экстента БД.
- **UNIFORM** - указывает, что экстенты имеют фиксированный размер в **SIZE** байт.

Давайте сразу рассмотрим конструкцию изменения табличного пространства, которая почти аналогична конструкции создания за исключением некоторых нюансов:

```
ALTER TABLESPACE -- (табличное_пространство) -- DATAFILE -- (спецификация_файла) -
(авторасширение)-*

--* -- MINIMUM -> EXTENT - (целое число) - К(килобайт)/М(мегабайт) --
-- LOGGING --
-- NOLGGING --
-- DEFAULT STORAGE -
-- ONLINE --

-- NORMAL --
-- TEMPORARY --
-- IMMEDIATE --

-- OFFLINE -- FOR RECOVER --
-- PERMANENT --
-- TEMPORARY --

-- BEGIN -- -- BACKUP --
-- END --

-- READ -- ONLY --
-- WRITE --

-- COALESCE --

-- (конструкция_управления_экстентами) -- (;)
--*

*-- конструкция_максимального_размера --

-- MAXISIZE -- UNLIMITED -----
-- (целое число) - К(килобайт)/М(мегабайт) --

"конструкция_авторасширения"

-- AUTOEXTEND -- OFF
-- NEXT -- (целое число) - К(килобайт)/М(мегабайт) --
-- ON ----- (конструкция_максимального_размера)
```

"конструкция\_управления\_экстентами"

```
----- DICTIONARY -----
----- AUTOALLOCATE -----
```

```
-- EXTEND -- MANAGEMENT -- LOCAL ---- UNIFORM -- SIZE -- (целое число) -
К(килобайт)/М(мегабайт) --
```

"конструкция\_файла\_данных/временного файла"

```
-- DATAFILE --
-- ADD -- TEMPFILE -- (спецификация_файла) - (авторасширение) --*

-- RENAME -- DATAFILE -- * - имя файла *- TO -* имя файла -----*
```

Что появилось нового? Рассмотрим:

- **NORMAL** - исключает все блоки файла данных из **SGA**. По умолчанию.
- **TEMPORARY** - выполняет контрольную точку для всех оперативных файлов данных табличного пространства.
- **IMMEDIATE** - немедленно без выполнения контрольной точки для файла данных.
- **OFFLINE - FOR RECOVER** - переводит в автономный режим для выполнения восстановления с привязкой по времени.
- **BEGIN - BACKUP** - запускает процесс открытого резервного копирования.
- **END** - останавливает процесс открытого резервного копирования. Доступ к табличному пространству при этом не запрещается.
- **COALESCE** - (!) Для каждого файла данных, в табличном пространстве, объединяет все непрерывные свободные экстенты в один непрерывный экстенст большого размера.
- **DATAFILE / ADD - TEMPFILE - (спецификация\_файла) - (авторасширение)** - Добавляет файл данных к табличному пространству, или временный файл.
- **RENAME - DATAFILE - \* - имя файла \*- TO -\* имя файла -\*** - Переименовывает один или несколько файлов данных, перед этим необходимо перевести табличное пространство в автономный режим.

Ну и конечно же удаление табличного пространства производится командой:

```
DROP TABLESPACE -- (табличное_пространство) --(;)
```

Заметьте, что при этом сам файл с физического диска не удаляется! Его необходимо удалить средствами операционной системы! Не забывайте об этом. Вот таким образом происходит управление созданием и работой табличного пространства БД и его файлами данных. Далее, мы рассмотрим все это на примере! Запоминайте!

## Шаг 122 - Табличные пространства на практике

Теории было много, но теория не подкрепленная практикой сама по себе бесполезна! Попробуем создать свое табличное пространство и поэкспериментируем с ним. Начнем с простого (помните команду **CREATE TABLESPACE**?) Запускайте **SQL\*Plus** и вперед:

```
CREATE TABLESPACE TBLSPCMYONE DATAFILE 'C:\ORACLE\ORADATA\PROBA\TEMP\ONE.DAT'
SIZE 100M REUSE AUTOEXTEND ON NEXT 2M MAXSIZE 200M
/
```

Получаем:

```
SQL> CREATE TABLESPACE TBLSPCMYONE DATAFILE 'C:\ORACLE\ORADATA\PROBA\TEMP\ONE.DAT'
2 SIZE 100M REUSE AUTOEXTEND ON NEXT 2M MAXSIZE 200M
3 /
```

Раздел создан.

Мы с вами создали наше первое табличное пространство! Давайте посмотрим как это отразилось на самой системе. Сначала можете посмотреть содержимое каталога **C:\ORACLE\ORADATA\PROBA\TEMP** - и соответственно видим:

```
Volume in drive C has no label.
Volume Serial Number is 24FF-83ED
```

```
Directory of C:\Oracle\ORADATA\PROBA\TEMP
```

```
20.04.2004 20:40 <DIR> .
20.04.2004 20:40 <DIR> ..
20.04.2004 20:40 0 my.txt
20.04.2004 20:35 104 865 792 ONE.DAT
 2 File(s) 104 865 792 bytes
 2 Dir(s) 41 545 981 952 bytes free
```

Эти данные чисто гипотетические у вас может быть другое! У меня данные с моего рабочего сервера **Win2003** и **Oracle 9.0.1.0**, так что принимайте как есть! Главное видно, что файл появился и звать его **ONE.DAT**! Такое расширение я дал не случайно для того, чтобы не путать с основными файлами системы. У них расширение как это не смешно **DBF**. И каталог, я сделал загодя отдельный для того, чтобы не вносить сумятицу рабочий каталог сервера. Теперь давайте сделаем еще одно табличное пространство с другим именем и другими опциями создания:

```
CREATE TABLESPACE TBLSPCMYTWO LOGGING DATAFILE
'C:\ORACLE\ORADATA\PROBA\TEMP\TWO.DBF' SIZE 50M
REUSE AUTOEXTEND ON NEXT 1024K MAXSIZE 250M EXTENT MANAGEMENT LOCAL SEGMENT
SPACE
MANAGEMENT AUTO
/
```

Получаем:

```
SQL> CREATE TABLESPACE TBLSPCMY TWO LOGGING DATAFILE
'C:\ORACLE\ORADATA\PROBA\TEMP\TWO.DBF' SIZE 50M
 2 REUSE AUTOEXTEND ON NEXT 1024K MAXSIZE 250M EXTENT MANAGEMENT LOCAL SEGMENT
SPACE
 3 MANAGEMENT AUTO
 4 /
```

Раздел создан.

А теперь в каталоге уже два файла:

```
Volume in drive C has no label.
Volume Serial Number is 24FF-83ED
```

```
Directory of C:\Oracle\ORADATA\PROBA\TEMP
```

```
20.04.2004 20:40 <DIR> .
20.04.2004 20:40 <DIR> ..
20.04.2004 20:40 0 my.txt
20.04.2004 20:35 104 865 792 ONE.DAT
20.04.2004 20:29 52 436 992 TWO.DBF
 3 File(s) 157 302 784 bytes
 2 Dir(s) 41 545 981 952 bytes free
```

Для полноты картины посмотрим системное представление **dba\_data\_files**. Описание его полей можно получить дав такую команду:

```
DESC dba_data_files
```

Получаем:

```
SQL> DESC dba_data_files
```

| Name            | Type          | Nullable | Default | Comments                                            |
|-----------------|---------------|----------|---------|-----------------------------------------------------|
| FILE_NAME       | VARCHAR2(513) | Y        |         | Name of the database data file                      |
| FILE_ID         | NUMBER        | Y        |         | ID of the database data file                        |
| TABLESPACE_NAME | VARCHAR2(30)  | Y        |         | Name of the tablespace to which the file belongs    |
| BYTES           | NUMBER        | Y        |         | Size of the file in bytes                           |
| BLOCKS          | NUMBER        | Y        |         | Size of the file in ORACLE blocks                   |
| STATUS          | VARCHAR2(9)   | Y        |         | File status: "INVALID" or "AVAILABLE"               |
| RELATIVE_FNO    | NUMBER        | Y        |         | Tablespace-relative file number                     |
| AUTOEXTENSIBLE  | VARCHAR2(3)   | Y        |         | Autoextensible indicator: "YES" or "NO"             |
| MAXBYTES        | NUMBER        | Y        |         | Maximum size of the file in bytes                   |
| MAXBLOCKS       | NUMBER        | Y        |         | Maximum size of the file in ORACLE blocks           |
| INCREMENT_BY    | NUMBER        | Y        |         | Default increment for autoextension                 |
| USER_BYTES      | NUMBER        | Y        |         | Size of the useful portion of file in bytes         |
| USER_BLOCKS     | NUMBER        | Y        |         | Size of the useful portion of file in ORACLE blocks |

Теперь дадим вот такой запрос к этому представлению:

```
SELECT tablespace_name, file_name, status, bytes
FROM dba_data_files
```



```
WHERE tablespace_name LIKE 'TBLSP%'
ORDER BY tablespace_name, file_name
/
```

Получаем:

```
SQL> SELECT tablespace_name, file_name, status, bytes
2 FROM dba_data_files
3 WHERE tablespace_name LIKE 'TBLSP%'
4 ORDER BY tablespace_name, file_name
5 /
```

| TABLESPACE_NAME | FILE_NAME                            | STATUS    | BYTES     |
|-----------------|--------------------------------------|-----------|-----------|
| TBLSPCMYONE     | C:\ORACLE\ORADATA\PROBA\TEMP\ONE.DAT | AVAILABLE | 104857600 |
| TBLSPCMYTWO     | C:\ORACLE\ORADATA\PROBA\TEMP\TWO.DBF | AVAILABLE | 52428800  |

Все верно! У нас есть два табличных пространства, которые мы только что создали! Кстати я провожу все эти действия в схеме **SYS** и поэтому будьте внимательнее при удалении объектов! Или можете дать системные привилегии на создание и удаление табличных пространств **MILLER**-у, но какое из зол меньшее выбирать вам! :-) Теперь давайте переведем одно из табличных пространств в **OFFLINE**:

```
ALTER TABLESPACE TBLSPCMYONE OFFLINE
/
```

Получаем:

```
SQL> ALTER TABLESPACE TBLSPCMYONE OFFLINE
2 /
```

Раздел изменен.

Попробуем дать предыдущий запрос снова:

```
SELECT tablespace_name, file_name, status, bytes
FROM dba_data_files
WHERE tablespace_name LIKE 'TBLSP%'
ORDER BY tablespace_name, file_name
/
```

Получаем:

```
SQL> SELECT tablespace_name, file_name, status, bytes
2 FROM dba_data_files
3 WHERE tablespace_name LIKE 'TBLSP%'
4 ORDER BY tablespace_name, file_name
5 /
```

| TABLESPACE_NAME | FILE_NAME                            | STATUS    | BYTES |
|-----------------|--------------------------------------|-----------|-------|
| TBLSPCMYONE     | C:\ORACLE\ORADATA\PROBA\TEMP\ONE.DAT | AVAILABLE |       |

```
TBLSPCMYTWO C:\ORACLE\ORADATA\PROBA\TEMP\TWO.DBF AVAILABLE 52428800
```

Видите все вроде верно, но поле **BYTES** для табличного пространства **TBLSPCMYONE** пустое! Все верно оно в режиме **OFFLINE**! Убедимся в этом дав такой запрос к системному представлению **DBA\_TABLESPACES**:

```
SELECT TABLESPACE_NAME, PCT_INCREASE, STATUS, CONTENTS, LOGGING
FROM DBA_TABLESPACES
WHERE TABLESPACE_NAME LIKE 'TBLSP%'
ORDER BY TABLESPACE_NAME
/
```

Получаем:

```
SQL> SELECT TABLESPACE_NAME, PCT_INCREASE, STATUS, CONTENTS, LOGGING
2 FROM DBA_TABLESPACES
3 WHERE TABLESPACE_NAME LIKE 'TBLSP%'
4 ORDER BY TABLESPACE_NAME
5 /
```

| TABLESPACE_NAME | PCT_INCREASE | STATUS  | CONTENTS  | LOGGING |
|-----------------|--------------|---------|-----------|---------|
| TBLSPCMYONE     |              | OFFLINE | PERMANENT | LOGGING |
| TBLSPCMYTWO     |              | ONLINE  | PERMANENT | LOGGING |

Вот теперь все понятно, к стати обратите внимания на поля **CONTENTS**, **LOGGING** помните предыдущий шаг? :) Вернем пространство **TBLSPCMYONE** в **ONLINE**:

```
ALTER TABLESPACE TBLSPCMYONE ONLINE
/
```

Получаем:

```
SQL> ALTER TABLESPACE TBLSPCMYONE ONLINE
2 /
```

Раздел изменен.

Теперь дадим такую команду:

```
ALTER TABLESPACE TBLSPCMYTWO OFFLINE
/
```

Получаем:

```
SQL> ALTER TABLESPACE TBLSPCMYTWO OFFLINE
2 /
```

Раздел изменен.

---

Теперь пространство **TBLSPCMYTWO** в отключке! Попробуем что-либо создать на нем в таком режиме! Например табличку:

```
SQL> CREATE TABLE TEST
 2 (
 3 TEST VARCHAR2(100)
 4)
 5 TABLESPACE TBLSPCMYTWO
 6 /
```

```
CREATE TABLE TEST
```

```
*
```

ошибка в строке 1:

ORA-01542: раздел 'TBLSPCMYTWO' явл. автономным, в нем нельзя распределять память

Все привет! **ORA-01542** настигло нас! Вернем его в нормальное состояние и создадим таблицу:

```
ALTER TABLESPACE TBLSPCMYTWO ONLINE
/
```

```
CREATE TABLE SYS.TEST
(
TEST VARCHAR2(100)
)
TABLESPACE TBLSPCMYTWO
/
```

Получаем:

```
SQL> ALTER TABLESPACE TBLSPCMYTWO ONLINE
 2 /
```

Раздел изменен.

```
SQL> CREATE TABLE SYS.TEST
 2 (
 3 TEST VARCHAR2(100)
 4)
 5 TABLESPACE TBLSPCMYTWO
 6 /
```

Таблица создана.

Прекрасно! А, вот теперь попробуйте удалить пространство из словаря данных БД:

```
DROP TABLESPACE TBLSPCMYTWO
/
```

Получаем:

```
SQL> DROP TABLESPACE TBLSPCMYTWO
 2 /
```

---

```
DROP TABLESPACE TBLSPCMY TWO
```

```
*
```

ошибка в строке 1:

ORA-01549: раздел не пуст, используйте опцию INCLUDING CONTENTS

Вот теперь нам заявляют с помощью **ORA-01549**, что раздел не пуст! Можно конечно дать команду типа:

```
DROP TABLESPACE TBLSPCMY TWO INCLUDING CONTENTS
```

```
/
```

Но, чтобы не раздражать пользователя **SYS** (все же он **DBA!**) удалим табличку, а потом файл пространства в каталоге:

```
DROP TABLE TEST
```

```
/
```

```
DROP TABLESPACE TBLSPCMY TWO INCLUDING CONTENTS
```

```
/
```

Получаем:

```
SQL> DROP TABLE TEST
```

```
2 /
```

Таблица удалена.

```
SQL> DROP TABLESPACE TBLSPCMY TWO INCLUDING CONTENTS
```

```
2 /
```

Раздел удален.

Вот и все осталось только стереть файл **TWO.DBF** в каталоге **C:\ORACLE\ORADATA\PROBA\TEMP** средствами ОС и на этом все закончится с удалением табличного пространства из чрева вашего сервера БД! Первое пространство можете тоже удалить или оставить! Решайте сами. Пока закрепляйте материал! ;-)

---

## Шаг 123 - БД Oracle - Табличные пространства, ввод - вывод

В [прошлый раз](#) мы с вами создали пару табличных пространств, и поработали с ними. Кое, что надеюсь, стало уже яснее, а кое что может еще потребовать рассмотрения. Если вы оставили табличные пространства из [прошлого шага](#), то можете работать с ними, если нет, то давайте сделаем следующее. Посмотрим как можно, создав табличное пространство, добавить к нему файл данных, а затем рассмотрим что из этого вытекает! Итак, запускаем **SQL\*Plus** и заходим в ваш экземпляр с правами **DBA** пользователем **SYS** или **SYSTEM**, (можете и **MILLER**'ом, если ваш админ дал вам права на создание табличных пространств) затем запускаем такой **SQL** код: (если хотите, можете использовать табличное пространство из [прошлого шага](#)):

```
CREATE TABLESPACE PROBATBS DATAFILE 'C:\ORACLE\ORADATA\PROBA\PRBONE.DAT'
SIZE 100M REUSE AUTOEXTEND ON NEXT 10M MAXSIZE 200M
/
```

Видим после прохода:

```
SQL> CREATE TABLESPACE PROBATBS DATAFILE 'C:\ORACLE\ORADATA\PROBA\PRBONE.DAT'
2 SIZE 100M REUSE AUTOEXTEND ON NEXT 10M MAXSIZE 200M
3 /
```

Раздел создан.

Мы создали табличное пространство **PROBATBS** величиной 100M и возможностью расширения до 200M. А что если вам по какой-либо причине, скажем, почти полной загрузке этого пространства, добавить еще один раздел! Запустим вот такой скрипт:

```
ALTER TABLESPACE PROBATBS ADD DATAFILE 'C:\ORACLE\ORADATA\PROBA\PRBTWO.DAT'
SIZE 50M REUSE AUTOEXTEND ON NEXT 10M MAXSIZE 100M
/
```

Видим после прохода:

```
SQL> ALTER TABLESPACE PROBATBS ADD DATAFILE 'C:\ORACLE\ORADATA\PROBA\PRBTWO.DAT'
2 SIZE 50M REUSE AUTOEXTEND ON NEXT 10M MAXSIZE 100M
3 /
```

Раздел изменен.

Теперь к вашему табличному пространству **PROBATBS** добавлен еще один кусочек пространства в виде файла данных **PRBTWO.DAT**. Который может расширяться до 100M, что в совокупности составит 300M табличного пространства **PROBATBS**. Для большей убедительности давайте дадим запрос к системному представлению:

```
SELECT tablespace_name, file_name, status, bytes
FROM dba_data_files
WHERE tablespace_name LIKE 'PROB%'
ORDER BY tablespace_name, file_name
/
```

Получаем:

```
SQL> SELECT tablespace_name, file_name, status, bytes
2 FROM dba_data_files
3 WHERE tablespace_name LIKE 'PROB%'
4 ORDER BY tablespace_name, file_name
5 /
```

| TABLESPACE_NAME | FILE_NAME                          | STATUS    | BYTES     |
|-----------------|------------------------------------|-----------|-----------|
| PROBATBS        | C:\ORACLE\ORADATA\PROBA\PRBONE.DAT | AVAILABLE | 104857600 |
| PROBATBS        | C:\ORACLE\ORADATA\PROBA\PRBTWO.DAT | AVAILABLE | 52428800  |

Хорошо видно, что мы получили одно табличное пространство **PROBATBS** состоящее из двух(!) файлов данных. Вот таким образом можно строить и вводить в работу табличные пространства, которые будут вам необходимы в дальнейшем.

И вот здесь мы подходим к тематике, которая напрашивается сама собой. Все табличные пространства это не что иное, как файлы данных на жестких дисках ваших серверов! Так? Значит здесь необходимо продумывать такой вопрос, как настройка всей системы в целом для достижения наибольшей производительности системы. Сама по себе настройка системы ввода-вывода заключается в основном в настройке физической структуры сегментов (таблиц и индексов), которые образуют базу данных. Настройка охватывает табличные пространства состоящие из экстентов, которые в свою очередь состоят из блоков и файлов данных. Все это представляет из себя, в конечном счете, объекты операционной системы! Операции ввода-вывода представляют из себя не что иное, как операции чтения и записи. А вернее это **DML** - операции такие как **SELECT, INSERT, UPDATE, DELETE**. Где **SELECT** - это операция чтения, а **UPDATE, INSERT, DELETE** - это операции записи! Так!? И **DDL** - операции это тоже операции записи! Таким образом, любые операции ввода-вывода требуют настройки не только табличных пространств, экстентов, блоков и файлов данных, но также сегментов отката и журналов регистрации транзакций. В связи с тем, что **DML** - операции пользователей затрагивают все эти объекты.

Думаю, все помнят, я не говорил, что все будет просто! **Oracle** достаточно сложная СУБД, но и в тоже время простая и надежная БД! Далее попробуем во всем этом разобраться! :)

## Шаг 124 - БД Oracle - СИСТЕМА Ввода-Вывода и работа с ней

Давайте попробуем разобраться, как добиться наибольшей производительности при работе и обращениях к БД **Oracle**. Например, начнем с того, как вообще строится сервер БД. Как правило он имеет массив хард дисков (винчестеров, венеков и т. д.!) Они могут быть **IDE** или **SCSI**. В свою очередь, они могут быть в **RAID** массивах или без таковых. Все это только для примера при выборе оборудования. В руководстве **Oracle**, как правило рекомендуют размещать каждое табличное пространство на отдельном диске. Как, например **SYS**, **TEMP**, **INDEX** и т.д. В наших с вами условиях это не всегда получается и приходится исходя из возможностей строить максимум из минимум доступного! При построении такой БД, как правило, неизбежны конфликты при обращении к жесткому диску (**disk contention**). Такого рода конфликт может возникать, когда один или несколько пользователей, работают с одним и тем же жестким диском сервера. Если, к примеру две таблицы большого объема имеют в запросах объединения, то их табличные пространства следует размещать на разных хард драйвах! Хотя я что-то опять размечтался! То же касается и индексов и сегментов отката! Идеальная картинка будет именно тогда когда, все эти разделы (табличные пространства) расположены на разных дисках системы. Кстати у меня эта заветная мечта так и не осуществилась - хотя у меня в системе три сервера, на двух из которых стоит **Oracle**. Кто знает может вам повезет больше! :) Это первое минимальное требование. Но в принципе, если все спланировать и на двух **SCSI** драйвах, то тоже работает и не плохо! Далее, при создании схем не забывайте про то, где будет данный пользователь размещать свои объекты БД. Если что-то пошло не так можно применить следующий ход:

```
ALTER USER <ПОЛЬЗОВАТЕЛЬ> DEFAULT TABLESPACE <ДРУГОЕ ТАБЛИЧНОЕ ПРОСТРАНСТВО>
TEMPORARY TABLESPACE TEMP (или другое на выбор)
```

Собственно кроме определения самого типа приложения (**typing the application**) сразу классифицируйте таблицы по уровням активности. Очень активные таблицы (**hot table**), размещают как можно ближе к дискам **SCSI**, а таблицы типа **warm table** и **cold table**, можете раскидывать по **IDE RAID**-ам. Чем больше **DML** - операций тем, больше фрагментация в табличных пространствах. Тем больше вам головной боли как **DBA**! Далее можно посмотреть в сторону блоков и экстенгов! Как вы уже поняли я думаю в **Oracle** блоки собраны в экстенги, которые образуют физическую среду хранимых табличных пространств. Следовательно чем эффективнее управление экстенгами, тем выше производительность системы в целом! Так же неоправданное увлечение динамическим выделением памяти приводит к большим накладным расходам и снижению производительности операций ввода-вывода. По этому по мере возможности применяйте статическое предварительное выделение памяти для сегментов БД (таблиц, индексов и т.д.). То есть при указании задаваемых табличных пространств с последующим предварительным выделением памяти для таблиц можно с несколько большей гибкостью комбинировать в одном и том же табличном пространстве таблицы с разными требованиями по отношению к экстенгам. Можно привести такой пример:

```
CREATE TABLESPACE TS1
DATAFILE '/data1/file1.dat' SIZE 256M
DEFAULT STORAGE (INITIAL 100M NEXT 100M
MINEXTENTS1);
```

```
CREATE TABLE T1 (a number(9), ..., z number(9))
TABLESPACE TS1;
```

```
CREATE TABLE T2 (a number(9), ..., z number(9))
TABLESPACE TS1;
```

Здесь выделение памяти производится предварительно для всего табличного пространства. Вернее для двух таблиц вместе.

А вот в этом случае:

```
CREATE TABLESPACE TS1
DATAFILE '/data1/file1.dat' SIZE 256M;

CREATE TABLE T1 (a number(9), ..., z number(9))
TABLESPACE TS1
STORAGE (INITIAL 100M NEXT 10M
MINEXTENTS 1);

CREATE TABLE T2 (a number(9), ..., z number(9))
TABLESPACE TS1
STORAGE (INITIAL 100M NEXT 10M
MINEXTENTS 1);
```

Подход несколько иной, так как здесь выделяется память для каждой таблицы отдельно! Второе выделение памяти для таблиц позволяет обеспечить более гибкое управление не только ростом табличного пространства, но и связанным с ним изменением производительности работы. А вот как поступать в каждом случае, это уже решать вам! Основное нужно четко представлять чего вы хотите от вышей БД прежде всего. Какого типа она будет! Думайте! Удачи! :-)))

---



## Шаг 125 - БД Oracle - Оценка табличных пространств - дефрагментация

Настало, наконец, время заняться серьезными вещами! :) Любая операция ввода-вывода, о которой мы говорили ранее, как следствие влечет за собой фрагментацию табличного пространства. Еще ее можно назвать фрагментацией экстентов (**extent fragmentation**). Возникает она при изменении размещения экстентов в табличном пространстве. Может происходить либо за счет фрагментации свободного пространства, либо за счет фрагментации таблиц. Как правило следует избегать размещения в одном разделе таблиц с разными уровнями активности. Экстент можно назвать свободным (**free extents**), если он никогда не выделялся для сегментов табличного пространства, либо был освобожден в результате удаления сегментов. Фрагментации экстентов бывают разных типов. Например, если изолированные наборы свободных экстентов, распределены по всему табличному пространству, то такая фрагментация называется - фрагментация типа "швейцарский сыр" (**Swiss cheese fragmentation**) или пузырьковой фрагментацией (**bubbling**). При наличии смежных свободных экстентов, которые распределены по всему свободному табличному пространству - фрагментация называется - "сотовой фрагментацией" (**honeycomb fragmentation**). Каждый из этих двух типов фрагментации является - "фрагментацией со свободными экстентами" (**free space fragmentation**). Существует так же и - "фрагментация таблиц" (**table fragmentation**), то есть динамическое расширение таблицы за пределы ее исходных экстентов. Из всего вышесказанного ясно, что чрезмерная фрагментация приводит к замедлению вашей БД. Так как для поисков разрозненных участков, например таблицы при проведении запроса к ней может потребовать существенно большего времени, в отличие от таблицы расположенной одним "не равным" экстендом! Надеюсь это понятно!

Давайте углубимся в это еще немного! В жизни это вам пригодится! Итак, создается табличное пространство. В нем создается сегмент имеющий, скажем, один экстент состоящий из **n**-блоков **Oracle**! Ясно? Вот так весь этот механизм строится. Сегмент так же может состоять из нескольких разделов - экстентов. Еще раз повторюсь сами по себе экстенты представляют собой группы смежных блоков **Oracle**. Когда уже существующий экстент уже не может вместить в себя данные - сегмент получает еще один экстент. Вот так! Процесс продолжается до тех пор пока в файле данных не останется больше места, либо пока не будет достигнуто внутреннее максимальное количество экстентов на сегмент! Уловили? Для упрощения управления сегментами лучше, чтобы каждый сегмент данных содержал по одному экстенту! Вот по этому параметр хранения сегмента - **initial** определяющий размер начального экстенга должен быть достаточно большим, чтобы обработать все данные сегмента! Кстати, если сегмент состоит из нескольких экстенгов, то это не дает гарантии, что они расположены рядом друг с другом. Для примера можно сказать, что выполнить операцию **DROP TABLE** для таблицы содержащей, скажем, 3000 экстенгов при хорошем стечении обстоятельств может занять 5-10 мин. А вот попробуйте удалить дефрагментированную таблицу в **60 000** экстенгов, для этого Вам понадобится целый день! Давайте посмотрим, как себя чувствует наша учебная табличка **CUSTOMERS**? Дадим вот такой запрос:

```
SELECT TABLESPACE_NAME,
 OWNER,
 SEGMENT_NAME,
 SEGMENT_TYPE,
 EXTENTS,
 BLOCKS,
 BYTES
FROM DBA_SEGMENTS
WHERE TABLESPACE_NAME = 'USERS' AND
 SEGMENT_NAME = 'CUSTOMERS'
AND OWNER = 'MILLER'
/
```

Получаем:

```
SQL> SELECT TABLESPACE_NAME,
2 OWNER,
3 SEGMENT_NAME,
4 SEGMENT_TYPE,
5 EXTENTS,
6 BLOCKS,
7 BYTES
8 FROM DBA_SEGMENTS
9 WHERE TABLESPACE_NAME = 'USERS' AND
10 SEGMENT_NAME = 'CUSTOMERS'
11 AND OWNER = 'MILLER'
12 /
```

| TABLESPACE_NAME | OWNER  | SEGMENT_NAME | SEGMENT_TYPE | EXTENTS | BLOCKS | BYTES  |
|-----------------|--------|--------------|--------------|---------|--------|--------|
| USERS           | MILLER | CUSTOMERS    | TABLE        | 2       | 16     | 131072 |

Да! результат не веселый! Хотя бы тот факт, что табличка уже размазалась по двум экстендам! Да размерчик у нее что-то больно велик. Так вот для того, чтобы уменьшить "расходы" при работе с сильно фрагментированными таблицами в **Oracle 8i** и старше введены локально управляемые табличные пространства! (**locally managed tablespaces**), в которых информация об использовании экстендов хранится в битовой карте в заголовке файла данных, а не в словаре данных. Вот так и решили проблему. А, теперь давайте посмотрим информацию об экстендах одного сегмента данных. Все той же нашей таблички **CUSTOMERS**. Дадим вот такой запрос:

```
SELECT TABLESPACE_NAME,
 OWNER,
 SEGMENT_NAME,
 SEGMENT_TYPE,
 EXTENT_ID,
 BLOCK_ID,
 BYTES
 BLOCKS
FROM DBA_EXTENTS
WHERE TABLESPACE_NAME = 'USERS'
AND SEGMENT_NAME = 'CUSTOMERS'
AND OWNER = 'MILLER'
/
```

Получаем:

```
SQL> SELECT TABLESPACE_NAME,
2 OWNER,
3 SEGMENT_NAME,
4 SEGMENT_TYPE,
5 EXTENT_ID,
6 BLOCK_ID,
7 BYTES
8 BLOCKS
9 FROM DBA_EXTENTS
```

---

```
10 WHERE TABLESPACE_NAME = 'USERS'
11 AND SEGMENT_NAME = 'CUSTOMERS'
12 AND OWNER = 'MILLER'
13 /
```

| TABLESPACE_NAME | OWNER  | SEGMENT_NAME | SEGMENT_TYPE | EXTENT_ID | BLOCK_ID | BLOCKS |
|-----------------|--------|--------------|--------------|-----------|----------|--------|
| USERS           | MILLER | CUSTOMERS    | TABLE        | 0         | 41       | 65536  |
| USERS           | MILLER | CUSTOMERS    | TABLE        | 1         | 49       | 65536  |

В вашем случае информация может и отличаться, но мой сервер по такому запросу дает именно это! Как можно видеть таблица находится в двух экстентах. Само собой напрашивается вопрос "как же все-таки решать проблему дефрагментации табличных пространств"? А, вот с этим мы далее и разберемся! А, пока усваивайте выше изложенное.

## Шаг 126 - БД Oracle - Дефрагментация и борьба с ней!

Как же все же решить проблему дефрагментации табличного пространства? Существует несколько способов. Мы постепенно рассмотрим каждый и попробуем понять, как это работает и на что действует. А начнем с дефрагментации свободных экстенгов. Свободный экстенг в табличном пространстве представляет собой набор смежных блоков. После удаления сегмента его экстенги помечаются как свободные. Однако они не всегда объединяются с соседними свободными экстенгами. Между ними могут быть барьеры. А, дело в следующем, если значение параметра **pcincrease** по умолчанию для табличного пространства не равно нулю, то фоновый процесс **SMON** - периодически объединяет соседние свободные экстенги. Если же **pcincrease = 0**, то БД не будет объединять свободное место в табличном пространстве. Для слияния соседних свободных экстенгов можно использовать параметр **COALESCE** команды **ALTER TABLESPACE!** В этом случае слияние произойдет независимо от параметра **pcincrease**. Кстати для заметки, процесс **SMON** осуществляет слияние в тех табличных пространствах, в которых значение параметра **pcincrease** по умолчанию не равно нулю! А вот **pcincrease = 1** заставит **SMON** объединять смежные области свободного места в табличном пространстве не оказывая особого влияния на размер следующего экстенга! Вот такие дела! Так что, только в идеале каждый объект БД находится только в одном экстенге и все доступное свободное место на диске расположено в одном большом непрерывном экстенге! Но, это только в идеале! :) Для оценки фрагментации табличного пространства основным показателем является размер самого большого свободного экстенга, выраженный в процентах от общего свободного места (т.е. на сколько БД близка к идеалу). Число полученное для каждого табличного пространства называется "индексом фрагментации свободного места" (**free space fragmentation index - FSFI**). Можно развить этот индекс, уделив внимание другим критериям. Заметьте, что индекс учитывает не общий объем доступного свободного места на диске, а только его структуру! Итак, выглядит это так:

$$FSFI = 100 * \sqrt[4]{\frac{\text{размер самого большого экстенга}}{\text{сумма всех экстенгов}}}$$

Запомните эту формулу, она вам пригодится еще в жизни! :) Наибольшее значение **FSFI** (для идеального табличного пространства содержащего только один файл данных) равно 100 (у меня было и больше!) По мере роста количества экстенгов значение **FSFI** медленно снижается. С уменьшением размера самого большого экстенга значение **FSFI** падает очень быстро! Давайте посмотрим, как это все работает на практике, вот, например, что я получил для своей БД, которая у меня развернута дома:

```
-- *****
-- Letuchiy S.V. @ By Presents *
-- *****
-- *
-- Calculate FSFI index. *
-- *
-- DATA BASE "MONOLIT" *
-- *
-- *****
```

```
SET NEWPAGE 0 PAGESIZE 60
COLUMN FSFI FORMAT 999.99
```

```
SELECT TABLESPACE_NAME,
 SQRT(MAX(BLOCKS)/SUM(BLOCKS))+
 (100/SQRT(SQRT(COUNT(BLOCKS)))) FSFI
```

```

FROM DBA_FREE_SPACE
GROUP BY TABLESPACE_NAME
ORDER BY 1
/

```

Получаем:

```

SQL> -- *****
SQL> -- Letuchiy S.V. @ By Presents *
SQL> -- *****
SQL> -- *
SQL> -- Calculate FSFI index. *
SQL> -- *
SQL> -- *
SQL> -- DATA BASE "MONOLIT" *
SQL> -- *
SQL> -- *****
SQL>
SQL> SET NEWPAGE 0 PAGESIZE 60
SQL> COLUMN FSFI FORMAT 999.99
SQL>
SQL> SELECT TABLESPACE_NAME,
2 SQRT(MAX(BLOCKS)/SUM(BLOCKS))+
3 (100/SQRT(SQRT(COUNT(BLOCKS)))) FSFI
4 FROM DBA_FREE_SPACE
5 GROUP BY TABLESPACE_NAME
6 ORDER BY 1
7 /
TABLESPACE_NAME FSFI

CWMLITE 85.08
DRSYS 101.00
EXAMPLE 101.00
INDX 101.00
ODM 101.00
OEM_REPOSITORY 101.00
PROBATBS 84.91
SYSTEM 101.00
TOOLS 101.00
UNDOTBS1 47.68
USERS 101.00
XDB 101.00

```

12 строк выбрано.

Этот сценарий я использую уже давно, и по этому в нем остался заголовок. Думаю вы не в обиде на мое тщеславие! :) Но главное, чтобы вы поняли, как это определяется и, что делать дальше! Имея значение **FSFI** базы данных можно определить базовую линию. Хотя я думаю вы не так часто, будете сталкиваться с проблемами доступности свободного места в табличном пространстве и имеющего адекватный объем этого свободного места и коэффициент **FSFI** более 30. А вот для того, чтобы определить распределение свободных экстенгов и их размеры, а так же, чтобы определить какие объекты являются барьерами между свободными экстенгами, запустите следующий сценарий:

```
set pagesize 60 linesize 132 verify off
column file_id heading "File|Id"
```

```
select
 'free space' Owner,
 ' ' Object,
 File_ID,
 Block_ID,
 Blocks
from DBA_FREE_SPACE
where Tablespace_Name = 'USERS'
and Owner = 'MILLER'
union
select
 SUBSTR(Owner,1,20),
 SUBSTR(Segment_Name,1,32),
 File_ID,
 Block_ID,
 Blocks
from DBA_EXTENTS
where Tablespace_Name = 'USERS'
and Owner = 'MILLER'
order by 3,4
/
```

Получаем:

```
SQL> set pagesize 60 linesize 132 verify off
SQL> column file_id heading "File|Id"
SQL>
SQL> select
 2 'free space' Owner,
 3 ' ' Object,
 4 File_ID,
 5 Block_ID,
 6 Blocks
 7 from DBA_FREE_SPACE
 8 where Tablespace_Name = 'USERS'
 9 and Owner = 'MILLER'
 10 union
 11
SQL> select
 2 SUBSTR(Owner,1,20),
 3 SUBSTR(Segment_Name,1,32),
 4 File_ID,
 5 Block_ID,
 6 Blocks
 7 from DBA_EXTENTS
 8 where Tablespace_Name = 'USERS'
 9 and Owner = 'MILLER'
 10 order by 3,4
 11 /
```

---

|                    |                           | File |          |        |
|--------------------|---------------------------|------|----------|--------|
| SUBSTR(OWNER,1,20) | SUBSTR(SEGMENT_NAME,1,32) | Id   | BLOCK_ID | BLOCKS |
| -----              |                           |      |          |        |
| MILLER             | BOYS                      | 9    | 9        | 8      |
| MILLER             | BOYS                      | 9    | 17       | 8      |
| MILLER             | SYS_C003505               | 9    | 25       | 8      |
| MILLER             | SYS_C003505               | 9    | 33       | 8      |
| MILLER             | CUSTOMERS                 | 9    | 41       | 8      |
| MILLER             | CUSTOMERS                 | 9    | 49       | 8      |
| MILLER             | SYS_C003506               | 9    | 57       | 8      |
| MILLER             | SYS_C003506               | 9    | 65       | 8      |
| MILLER             | GIRLS                     | 9    | 73       | 8      |
| MILLER             | GIRLS                     | 9    | 81       | 8      |
| MILLER             | SYS_C003507               | 9    | 89       | 8      |
| MILLER             | SYS_C003507               | 9    | 97       | 8      |
| MILLER             | OFFICES                   | 9    | 105      | 8      |
| MILLER             | OFFICES                   | 9    | 113      | 8      |
| MILLER             | SYS_C003511               | 9    | 121      | 8      |
| MILLER             | SYS_C003511               | 9    | 129      | 8      |
| MILLER             | ORDERS                    | 9    | 137      | 8      |
| MILLER             | ORDERS                    | 9    | 145      | 8      |
| MILLER             | SYS_C003512               | 9    | 153      | 8      |
| MILLER             | SYS_C003512               | 9    | 161      | 8      |
| MILLER             | PRODUCTS                  | 9    | 169      | 8      |
| MILLER             | PRODUCTS                  | 9    | 177      | 8      |
| MILLER             | SYS_C003513               | 9    | 185      | 8      |
| MILLER             | SYS_C003513               | 9    | 193      | 8      |
| MILLER             | SALESREPS                 | 9    | 201      | 8      |
| MILLER             | SALESREPS                 | 9    | 209      | 8      |
| MILLER             | SYS_C003515               | 9    | 217      | 8      |
| MILLER             | SYS_C003515               | 9    | 225      | 8      |
| MILLER             | TBLA                      | 9    | 74537    | 8      |
| MILLER             | ADT                       | 9    | 74545    | 8      |
| MILLER             | TSTTRIG                   | 9    | 74553    | 8      |
| MILLER             | SYS_C003550               | 9    | 74561    | 8      |
| MILLER             | TSTSV                     | 9    | 74569    | 8      |
| MILLER             | SYS_C003551               | 9    | 74577    | 8      |
| MILLER             | MYAUDIT                   | 9    | 75961    | 8      |

35 строк выбрано.

Как видно здесь нет записей вида **free space**, но это не так важно, главное что пользователь **MILLER** их пока не имеет, но думаю в дальнейшем мы устроим такую возможность нашему тренировочному пользователю! С оценками **FSFI** мы разобрались, далее будем решать проблемы собственно устранения самой дефрагментации. Так что, попробуйте поработать с вашим табличным пространством, например, создавая и очищая несколько таблиц! Пробуйте!

## Шаг 127 - БД Oracle - Быстрее, выше, сильнее!

В [прошлом шаге](#) я говорил о команде сцепления свободных экстенгов. В следствии того, что табличное пространство это наиболее изменяемый объект БД, от состояния которого в первую очередь зависит скорость работы ваших запросов и получения данных из БД. Все конечно относительно. Кто-то требует высокой скорости получения данных, а для кого-то подождать минуту другую, совсем не страшно. Вот здесь и вытекает момент грамотности построения экземпляра БД. Как бы вы не старались, но ваши таблицы, процедуры и т.д. меняются с течением времени и табличное пространство неизбежно все более и более дефрагментируется. Итак, как же все-таки с этим бороться. Самое простое это использовать коэффициент **FSFI** и, если в результате анализа карты свободных экстенгов, станет ясно, что их слияние полезно для выбранного табличного пространства их можно слить либо вручную, либо используя рабочую лошадку, фоновый процесс **SMON**. Чтобы заставить его это проделать, параметру **pctincrease** необходимо присвоить значение не равное нулю. Например, это можно сделать так:

```
ALTER TABLESPACE MYTABLESPACE DEFAULT STORAGE (pctincrease 1)
/
```

При этом, если в табличном пространстве **MYTABLESPACE** создается объект без указания значения параметра **pctincrease**, то будет использовано его значение по умолчанию. В большинстве случаев небольшие значения этого параметра сравнительно точно соответствуют обычному линейному росту числа строк в вашей БД. По этому было использовано самое маленькое не нулевое значение этого параметра - 1. Замечу, что с помощью конструкции **storage** создаваемого объекта это значение можно переписать. Теперь давайте попробуем проделать объединение свободных экстенгов вручную. Сделать это можно применив параметр **COALESCE** команды **ALTER TABLESPACE**. Вот таким образом:

```
ALTER TABLESPACE MYTABLESPACE COALESCE
/
```

Например, я проделал это с моим табличным пространством **USER**, и вот что получилось:

```
ALTER TABLESPACE USERS COALESCE
/
```

Получил:

```
SQL> ALTER TABLESPACE USERS COALESCE
2 /
```

Раздел изменен.

Как это не смешно весь процесс у меня занял 0.218 секунды. Может быть, у вас будет и больше не знаю, но можете попробовать. Хотя библия Администратора БД **Oracle** утверждает, что в результате этой операции будут объединены свободные соседствующие экстенги. А вот, между которыми стоят барьеры, скорее всего нет! Так, что это не панацея! :) Лично, я чаще поступаю так, сливаю все содержимое БД в файл экспорта (эту большую тему мы будем рассматривать далее!). Затем полностью сношу мое табличное рабочее табличное пространство и создаю его заново. Получаю естественно чистый лист! Затем заливаю импортом все мои объекты одним экстенгом и вуоля! Мои таблички начинают откликаться на запросы просто молниеносно! Это самый радикальный способ, хотя не всегда применимый. Есть еще **Enterprise Manager** - но это очень



объемная тема и мы ее выделим отдельно. Скажу только что, **Enterprise Manager** может дефрагментировать любое табличное пространство на подобие программы **Norton Utilities, Speed Disk** - помните! Это тоже полезная функция для БД.

Так же существует такое понятие, как "сцепление строк". Как правило сцепление снижает производительность запроса, поскольку **ORACLE** вынужден искать одну логическую строку в нескольких физических местах. Для того, чтобы избежать этого явления необходимо следить за параметром **pctfree** при создании сегмента данных и устанавливать правильное значение этого параметра. Для оценки сцепления строк есть команда **analyze**, которая собирает некую статистику о использовании различных объектов БД. Команда **analyze** позволяет так же находить и записывать сцепленные строки таблиц. Например, вот так:

```
ANALYZE TABLE MYTABLE LIST CHAINED ROWS INTO CHAINED_ROWS
/
```

Таблица **CHAINED\_ROWS** создается отдельно при работе с данной командой и принимает значения анализа. Помимо сцепления строк **ORACLE** время от времени перемещает их. Если строка превышает доступное в блоке место она может быть включена в другой блок. Процесс перемещения строки из одного блока в другой называется "миграцией" строк. А сама строка называется "мигрирующей". В ходе самого процесса **ORACLE** динамически управляет местом в нескольких блоках, а так же обращается к свободному списку (**free list** - список блоков доступных для вставок). По своей сути мигрирующая строка не является сцепленной, но влияет на производительность транзакции в целом! Вот, такой достаточно не простой механизм использует БД при размещении объектов в табличных пространствах. А задача **DBA** следить за этим и корректно использовать возможности системы для повышения ее производительности. Что, я собственно и хочу до вас довести. Применяя данные инструменты можно значительно повысить скорость работы вашей БД, но все естественно зависит от вас! Дерзайте!

---

## Шаг 128 - БД Oracle - Работа с табличными пространствами в целом

В [прошлый раз](#) я упустил один момент, давайте еще не надолго вернемся к дефрагментации. Это достаточно обширная тема, но думаю последнее, на что стоит обратить внимание это размер блока **Oracle**. Он содержится в файле **init.ora** в секции **db\_block\_size** и имеет, как правило, оптимальное значение выбранное по умолчанию. Но эффект от увеличения размера блок просто поражает! В большинстве случаев используют блоки двух размеров 2 и 4 Кбт. (Хотя я почти всегда ставлю 8 Кбт!). Переход на больший размер блока может повысить производительность на 50%! И достигается это без значительных затрат! Учтите, что менять секцию **db\_block\_size** просто так нельзя! Для увеличения размера блока БД лучше пересоздать весь экземпляр заново с новым значением! Повышение производительности связано со способом работы сервера **Oracle** с заголовком блока. Как следствие для данных используется больше места, что улучшает возможность обращения к одному и тому же блоку данных, от нескольких пользователей. Удвоение размера блока **Oracle** практически не влияет на его заголовок. Это значит, что в процентном отношении для заголовка расходуется меньше места! Но учтите, что, например, удвоение размера блока **Oracle** так же будет влиять на кэш буфера данных и может вызвать проблемы с управлением памятью на сервере!

Теперь давайте рассмотрим момент, когда табличное пространство необходимо модифицировать в ту или иную сторону. Например, рассмотрим случай когда табличное пространство и связанный с ним файл данных необходимо усесть в размерах! Сделать это можно, например, с помощью команды **ALTER DATABASE**. Но учтите, что нельзя изменить размер файла данных, если пространство, которое вы пытаетесь освободить, в настоящий момент занято объектами БД. Например, если объекты БД занимают объем 200 Мб, а размер файла данных 300 Мб, то можно отсечь только 100 Мб у файла данных! Сама команда будет выглядеть вот так:

```
ALTER DATABASE DATAFILE 'C:\Oracle\Oradata\PROBA\PRBONE.DAT' RESIZE 100M
/
```

При этом учтите, если табличное пространство сильно дефрагментировано, то **Oracle** может выдать ошибку при попытке усесть табличное пространство! Далее давайте посмотрим как можно производить сокращение таблиц и индексов в БД. Но, для начала проделаем следующее. Создадим таблицу **SPEED** в схеме **MILLER**:

```
CREATE TABLE SPEED (
 ID NUMBER,
 DT VARCHAR2(100)
)
/
```

Получаем:

```
SQL> CREATE TABLE SPEED (
2 ID NUMBER,
3 DT VARCHAR2(100)
4)
5 /
```

Таблица создана.

Теперь запишем вот такой блок для того, чтобы наполнить таблицу данными! Для примера:

```
SET SERVEROUTPUT ON

DECLARE

X NUMBER := 0;

BEGIN

 DBMS_OUTPUT.enable;

 FOR X IN 0..5000 LOOP

INSERT INTO SPEED(ID, DT)
 VALUES(X, 'Hello '||TO_CHAR(X));

 END LOOP;

 DBMS_OUTPUT.put_line('Good Job!');

END;
/
```

Получаем:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3 X NUMBER := 0;
4
5 BEGIN
6
7 DBMS_OUTPUT.enable;
8
9 FOR X IN 0..5000 LOOP
10
11 INSERT INTO SPEED(ID, DT)
12 VALUES(X, 'Hello '||TO_CHAR(X));
13
14 END LOOP;
15
16 DBMS_OUTPUT.put_line('Good Job!');
17
18 END;
19 /
Good Job!
```

Время, которое потратил **Oracle** в моем случае составило 2,5 Сек. (Это оценивает **PL/SQL Developer**). Когда **Oracle** записывает данные в сегмент, обновляется так называемая - верхняя отметка (**high - water mark** - высшая точка) сегмента. Верхняя отметка сегмента - это наибольший номер блока сегмента, в котором вы когда-либо хранили данные. Если вы добавили скажем 5000 строк верхняя отметка будет увеличиваться! Дайте к таблице **SPEED** вот такой запрос:

---

```
SELECT COUNT(*) FROM SPEED
/
```

Время на исполнение у меня было 0.016 сек. Хорошо. Запрос прошел все блоки таблицы до верхней отметки. А теперь удалим записи:

```
DELETE FROM SPEED
/
```

```
COMMIT
/
```

Время на удаление чуть больше, уже 0.235 сек! А теперь повторите прошлый запрос:

```
SELECT COUNT(*) FROM SPEED
/
```

Снова 0.016 сек! Но почему? А в следствии того, что при удалении записей из таблицы ее **high - water mark** не снижается и запрос прошел все блоки снова! Вот как! Если не считать удаление таблицы и ее воссоздание, верхняя отметка сегмента переустанавливается только после команды **TRUNCATE TABLE** (к ней мы еще вернемся!) Давайте сделаем следующее. Снова наполним таблицу:

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
2
3 X NUMBER := 0;
4
5 BEGIN
6
7 DBMS_OUTPUT.enable;
8
9 FOR X IN 0..5000 LOOP
10
11 INSERT INTO SPEED(ID, DT)
12 VALUES(X, 'Hello '||TO_CHAR(X));
13
14 END LOOP;
15
16 DBMS_OUTPUT.put_line('Good Job!');
17
18 END;
19 /
Good Job!
```

А теперь дадим команду нашего запроса:

```
SELECT COUNT(*) FROM SPEED
/
```

Время снова примерно 0.017 сек. Хорошо, даем вот такую команду:

---

```
TRUNCATE TABLE SPEED
/
```

Получаем:

```
SQL> TRUNCATE TABLE SPEED
2 /
```

Таблица усечена.

Снова запрос:

```
SELECT COUNT(*) FROM SPEED
/
```

Получаем:

```
SQL> SELECT COUNT(*) FROM SPEED
2 /

COUNT(*)

0
```

Затраченное время 0 сек! Указатель **high - water mark** был перемещен! Что и требовалось доказать! Здесь так же кроется некий подводный камень, при работе с таблицами БД и особенно большими таблицами! Знание этого нюанса думаю в дальнейшем поможет вам справляться с распределением табличного пространства под объекты БД. Найти верхнюю отметку для таблицы **CUSTOMERS** для схемы **MILLER** нашей учебной БД поможет такой сценарий (для того, чтобы все получилось необходимо зайти в экземпляр пользователем **SYS** или **SYSTEM!**):

```
SET SERVEROUTPUT ON
```

```
declare
```

```
VAR1 number;
VAR2 number;
VAR3 number;
VAR4 number;
VAR5 number;
VAR6 number;
VAR7 number;
```

```
begin
```

```
DBMS_OUTPUT.enable;
```

```
SYS.dbms_space.unused_space('MILLER','CUSTOMERS','TABLE',
VAR1,VAR2,VAR3,VAR4,VAR5,VAR6,VAR7);
dbms_output.put_line('OBJECT_NAME = SPACES');
dbms_output.put_line('-----');
dbms_output.put_line('TOTAL_BLOCKS = '||VAR1);
dbms_output.put_line('TOTAL_BYTES = '||VAR2);
```

---

```

dbms_output.put_line('UNUSED_BLOCKS = '||VAR3);
dbms_output.put_line('UNUSED_BYTES = '||VAR4);
dbms_output.put_line('LAST_USED_EXTENT_FILE_ID = '||VAR5);
dbms_output.put_line('LAST_USED_EXTENT_BLOCK_ID = '||VAR6);
dbms_output.put_line('LAST_USED_BLOCK = '||VAR7);
end;
/

```

Здесь используется пакет **SYS.dbms\_space** и его метод **unused\_space**! Получаем:

```

SQL> SET SERVEROUTPUT ON
SQL>
SQL> declare
2
3 VAR1 number;
4 VAR2 number;
5 VAR3 number;
6 VAR4 number;
7 VAR5 number;
8 VAR6 number;
9 VAR7 number;
10 begin
11
12 DBMS_OUTPUT.enable;
13
14 SYS.dbms_space.unused_space('MILLER','CUSTOMERS','TABLE',
15 VAR1,VAR2,VAR3,VAR4,VAR5,VAR6,VAR7);
16 dbms_output.put_line('OBJECT_NAME = SPACES');
17 dbms_output.put_line('-----');
18 dbms_output.put_line('TOTAL_BLOCKS = '||VAR1);
19 dbms_output.put_line('TOTAL_BYTES = '||VAR2);
20 dbms_output.put_line('UNUSED_BLOCKS = '||VAR3);
21 dbms_output.put_line('UNUSED_BYTES = '||VAR4);
22 dbms_output.put_line('LAST_USED_EXTENT_FILE_ID = '||VAR5);
23 dbms_output.put_line('LAST_USED_EXTENT_BLOCK_ID = '||VAR6);
24 dbms_output.put_line('LAST_USED_BLOCK = '||VAR7);
25 end;
26 /
OBJECT_NAME = SPACES

TOTAL_BLOCKS = 16
TOTAL_BYTES = 131072
UNUSED_BLOCKS = 8
UNUSED_BYTES = 65536
LAST_USED_EXTENT_FILE_ID = 9
LAST_USED_EXTENT_BLOCK_ID = 40
LAST_USED_BLOCK = 8

```

Процедура PL/SQL успешно завершена.

Здесь верхняя отметка таблицы (в байтах) представляет собой разницу между значениями **TOTAL\_BYTES** и **UNUSED\_BYTES**. Значение **UNUSED\_BLOCKS** соответствует числу блоков выше высшей точки. **TOTAL\_BLOCKS** это общее количество блоков связанное с данной таблицей!

Улавливаете! Если нужно сжать таблицу и значение **UNUSED\_BLOCKS** не равно нулю, с помощью команды **ALTER TABLE** можно забрать пространство выше верхней отметки. Чтобы освободить занимаемое таблицей пространство можно дать команду:

```
ALTER TABLE MILLER.CUSTOMERS DEALLOCATE UNUSED KEEP 65536
/
```

Получаем:

```
SQL> ALTER TABLE MILLER.CUSTOMERS DEALLOCATE UNUSED KEEP 65536
2 /
```

Таблица изменена.

И действительно зачем ей лишние 8 блоков! У меня это получается  $(16 * 8192) - (8 * 8192) = 65536$ ! Вот так лишнее долой! Кстати, если не указать конструкцию **keep**, то значение параметров сохранения **minextents** и **initial** таблицы останутся прежними. Если использовать **keep**, то можно освобождать свободное пространство из любого экстента! Даже из **initial**, если в других экстентах данных нет! Так, что пользуйтесь возможностью борьбы с неиспользуемым свободным местом табличных пространств! Но, осторожно! Удачи!

---

**Шаг 129 - БД Oracle - Импорт, экспорт данных в теории**

Думаю, что с табличными пространствами мы с вами немного разобрались. Давайте теперь рассмотрим как вообще возможно, извлечь данные из вашей БД, полностью или частично и сохранить их для последующего либо восстановления, либо хранения так на всякий случай. Данная, тема вытекает из раздела резервное копирование и восстановление БД. Способов это делать множество! Но, для начала рассмотрим утилиты импорта, экспорта. На мой взгляд - данное средство, является одним из самых простых и эффективных способов сохранности ваших данных. Если делать регулярные выборки утилитой экспорта, то это почти полная гарантия, что вы ничего не потеряете! Так же, данными средствами, легко переносить объекты БД из одной БД в другую или смене версии БД и т.д. Что, кстати характерно, изучать способы сохранения данных - администратор БД, начинает, когда БД, либо падает либо разваливается полностью! То есть исходя из принципа - "пока жаренный питух в п.. не клюнет!" Хотя этот принцип как правило у нас во всем! :) Ладно, закончим с лирическим отступлением и начнем разбирать данную тему. Собственно сами утилиты импорта экспорта, это два исполняемых модуля (экзешника), которые находятся как правило в каталоге определенным переменной **ORACLE\_HOME**. В вашем случае это может быть например в каталоге - **C:\Oracle\ora81\bin** и имеют имена **exp.exe** и **imp.exe**. Как уже, наверное, ясно одна из них производит выгрузку данных, а другая их загрузку в БД. Если войти в каталог **C:\Oracle\ora81\bin** и дать команду:

**C:\Oracle\ora81\bin\exp.exe help=Y**

То увидите примерно следующее:



```

C:\oracle\ora81\BIN>exp.exe help=y

Export: Release 8.1.5.0.0 - Production on 12:08:20 2004

(c) Copyright 1999 oracle corporation. All rights reserved.

Export яфёрьцхё трь ярёрьхёё/, хёйш ттхёёш юьрэфё EXP,
чр юёюёюц ёыхфёёё тр'ш шь _яюю№чютрехы /ярёю№:

 1ёшьхё: EXP SCOTT/TIGER

т/ ёрьцх юцхёх еяёрти ё№ т/яююхэхшхё Export, чрфртр яюёых юьрэф./ EXP
Ёрчышүэ/х рёуёыхёё/. -ы чрфрэш ярёрьхёёют шёяюю№чещёх ышүхт/х ёютр:

 4оёрьё: EXP KEYWORD=чэрүхэхш шыш KEYWORD=(чэрүхэхш1,...,чэрүхэхшN)
 1ёшьхё: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
 шыш TABLES=(T1:P1,T1:P2), хёйш T1 ты хёё ёрчфыххёюц ёрсышүхц

USERID фююцё с/ё№ яхёт/ь ярёрьхёёю т юьрэфёюц ёёёюх.

 ышү.ёы. фяшё.(4о ёююү.) ышү.ёы. фяшё.(4о ёююү.)

USERID шь _яюю№ч./ярёю№ FULL 1ёёяюёё тёхую їрщыр (N)
BUFFER ёрчьхё сёїхёр фрээ/ї OWNER ёяшёю шьхэ тырфхы№шхт
FILE т/їюфэ/х їрщыр (EXPDAT.DMP) TABLES ёяшёю шьхэ ёрсышү
COMPRESS шьяюёё т юфшэ 1ёёххё(У) RECORDLENGTH фышэр чряшёш тт/т/т
GRANTS 1ёёяюёё яёштыхушц (У) INCTYPE шзёхххёёры№э/ц 1ёёяюёё
INDEXES 1ёёяюёё шэфхёют (У) RECORD юёюхц. шзё.1ёёяюёё (У)
ROWS 1ёёяюёё ёсёю фрээ/ї(У) PARFILE шь їрщыр ярёрьхёёют
CONSTRAINTS 1ёёяюёё яёртшы үхы. (У) CONSISTENT ёюурё. ёт чрээ/ї ёрсышү
LOG яёюё. їрщы т/пофр эр 1ёёрэ STATISTICS рзрышч юс•хёют (ESTIMATE)
DIRECT яё юц яёё№ (N) TRIGGERS 1ёёяюёёшёех/х ёёшүүхё/ (У)
FEEDBACK т/пофшэ эр 1ёёрэ ёюс•хэхш ю уюютэюёёш ьрцф/ї х ёсёю (0)
FILESIZE ьрёшрь№э/ц ёрчьхё ьрцфюую їрщыр ёсёюёр
QUERY їёрчр т/сюёш шёяюю№чехьр фы 1ёёяюёёр яюфёюцхёстр ёрсышү/

 ыыхфёёшх ышүхт/х ёютр яёшьхэш/ ёюю№ю фы ёёрэёяюёё. ёрсы. яёюёёёрэёст
TRANSPORT_TABLESPACE 1ёёяюёёшёехё ьхёрфрээ/х ёёрэёяюёё. ёрсы. яёюёёёрэёст (N)
TABLESPACES ёяшёю ёрсы. яёюёёёрэёст фы ёёрэёяюёёшёютрэш

 1ёёяюёё чрхё'шыё ёёях'ёю схч яёхфёяёхцхэхшц.

```

Не знаю, по каким причинам, но все происходит именно так! Не волнуйтесь скоро мы с вами все приведем в норму! А, для начала давайте определимся с тем что же такое есть ЭКСПОРТ. Итак - утилита экспорта **ORACLE** читает базу данных, включая словарь данных и записывает результаты в двоичный файл(!). Который в свою очередь именуется как файл дампа экспорта (**export dump file**). В данном смысле можно экспортировать всю базу данных, конкретных пользователей (схемы) или конкретные таблицы вашей БД. В процессе экспорта, вы можете определиться есть ли необходимость экспортировать связанную с таблицами информацию словаря данных, такую как привилегии, индексы и ограничения. Созданный утилитой экспорта файл, будет содержать команды необходимые для полного восстановления всех выбранных объектов. Можно осуществлять полный (**complete**) экспорт всех таблиц БД, или только тех таблиц, которые были изменены, со времени последнего экспорта. Во втором случае экспорт будет инкрементальным или кумулятивным. Инкрементальный (**incremental**) экспорт привет к записи всех таблиц изменившихся со времени последнего экспорта, а кумулятивный (**cumulative**) - всех таблиц изменившихся со времени последнего полного экспорта! Но, что еще не маловажно, утилита экспорта предоставляет вам возможность сжимать свободные экстенды сильно фрагментированных сегментов данных. Например, я раньше (да и сейчас) поступал следующим образом. Когда какая-либо схема более менее сформировалась, я уничтожал ее табличное пространство предварительно

все слив в экспорт и затем восстанавливал табличное пространство и заливал данные! Разница, была очень заметна - все запросы к БД выполнялись быстрее! Хотя такой способ не всегда оправдан, но вполне применим! Итак, давайте попробуем получить команду **help** на понятном нам языке, а не вражескими кабалистическими знаками! :) Для этого напишем батовский файл с именем **exp\_out.bat** и вот таким содержимым: (надеюсь создавать файлы в вашей ОС вы умеете!)

```
set nls_lang=russian_cis.ru8pc866
```

```
exp.EXE help=y
```

Запустим его на выполнение и должны получить уже следующее на более понятном нам диалекте:

```
C:\oracle\ora81\BIN>exp_out.bat
C:\oracle\ora81\BIN>set nls_lang=russian_cis.ru8pc866
C:\oracle\ora81\BIN>exp.EXE help=y
Export: Release 8.1.5.0.0 - Production on Пнд Май 3 12:40:29 2004
(c) Copyright 1999 Oracle Corporation. All rights reserved.

Утилита Export подскажет Вам параметры, если ввести команду EXP,
за которой следуют Ваши имя_пользователя/пароль:

Пример: EXP SCOTT/TIGER

Вы также можете управлять выполнением Export, задавая после команды EXP
различные аргументы. Для задания параметров используйте ключевые слова:

Формат: EXP KEYWORD=значение или KEYWORD=(значение1,...,значениеN)
Пример: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
или TABLES=(T1:P1,T1:P2), если T1 является разделенной таблицей

USERID должно быть первым параметром в командной строке.

Ключ.сл. опис.(По умолч.) Ключ.сл. опис.(По умолч.)

USERID имя_польз./пароль FULL экспорт всего файла (N)
BUFFER размер буфера данных OWNER список имен владельцев
FILE выходные файла (EXPDAT.DMP) TABLES список имен таблиц
COMPRESS импорт в один экстенд(Y) RECORDLENGTH длина записи Вв/Выв
GRANTS экспорт привилегий (Y) INCTYPE инкрементальный экспорт
INDEXES экспорт индексов (Y) RECORD отслеж. инкр.экспорт (Y)
ROWS экспорт строк данных(Y) PARFILE имя файла параметров
CONSTRAINTS экспорт правил цел. (Y) CONSISTENT соглас. связанных таблиц
LOG прот. файл вывода на экран STATISTICS анализ объектов (ESTIMATE)
DIRECT прямой путь (N) TRIGGERS экспортируемые триггеры (Y)
FEEDBACK выводит на экран сообщение о готовности каждые x строк (0)
FILESIZE максимальный размер каждого файла сброса
QUERY фраза выборки используемая для экспорта подмножества таблицы

Следующие ключевые слова применимы только для транспорт. табл. пространств
TRANSPORT_TABLESPACE экспортирует метаданные транспорт. табл. пространств (N)
TABLESPACES список табл. пространств для транспортирования

Экспорт завершился успешно без предупреждений.
```

Вот теперь уже кое что проясняется! По крайней мере можно разобрать отдельные фразы! :) Далее, мы с вами продолжим разбирать все это более детально!

---

## Шаг 130 - БД Oracle - Экспорт - команды и управление выводом

Разбираемся далее с экспортом! Функциональные средства экспорта имеют три уровня: полный (**Full**) режим, пользовательский (**User**) режим и табличный (**Table**).

В полном режиме (**Full**) экспортируется вся база данных, т.е. читается весь словарь данных. В файл дампа экспорта записываются команды языка **DDL**, для полного восстановления БД. В своих схемах, этот файл будет содержать определение всех табличных пространств, всех пользователей и все объекты, данные и привилегии.

В пользовательском режиме (**User**) экспортируются объекты пользователя и содержащиеся в них данные. Все привилегии и индексы, созданные пользователем для своих объектов так же экспортируются. Не экспортируются только привилегии и индексы, созданные пользователями не являющимися владельцами этих объектов.

В табличном режиме (**Table**) экспортируется указанная таблица. Ее структура, индексы и привилегии экспортируются совместно с ее данными или без них. Табличный режим позволяет экспортировать так же все таблицы, принадлежащие пользователю (для этого нужно указать владельца схемы, но не указывать название таблицы). Можно так же экспортировать определенный раздел таблицы.

Для выполнения экспорта, я обычно пользуюсь исполняемыми файлами **.bat** - что, иногда приводит в замешательство некоторых моих знакомых, имеющих весьма смутное представление о том, что такое командная строка!

- О, ты все еще работаешь в DOS! Какой ужас!!! - восклицают они, видя на экране моего компьютера **Far Manager** и то как я вообще работаю с **Oracle** посредством командной строки! Ну, что ж вот такой я динозавр! А, меня вполне устраивает! Хотя есть способ выполнить экспорт с применением **OEM (Oracle Enterprise Manager)** - но об этом, чуть позже! Итак, для выполнения экспорта, проще всего написать **.bat** файл вот такого, например содержания:

..\full.bat -> это имя файла для примера, его не нужно помещать внутрь файла для экспорта!

```
set nls_lang=russian_cis.ru8pc866
```

```
exp.EXE USERID=sys/manager@proba FULL=Y FILE=c:\ORACLE\full.dat LOG=C:\ORACLE\full.LOG
```

**.bat** файл такого содержания полностью экспортирует всю вашу БД, в файл дампа с именем **full.dat**! Вы, можете поместить этот файл скажем на **CD** - диск и положить его в сейф! Все ваша БД теперь всегда может быть восстановлена заново!!! Но, не все так просто как кажется! Для того, чтобы быть более уверенным вам нужно запускать этот файл хотя бы раз в неделю или день! В зависимости от вашей потребности или еще чего либо еще! Но, это уже на ваше усмотрение! Итак, сейчас давайте разберем, все опции командной строки экспорта по порядку! Это будет полезно для вас в дальнейшем и вы можете пользоваться этим как справочником! Поехали!

| Ключевое слово | Описание                                                                                                                                                          |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| userid         | Имя пользователя и пароль учетной записи, выполняющей экспорт. Если он следует первым в командной строке <b>exp.exe</b> , то <b>userid</b> писать не обязательно! |
| buffer         | Размер буфера используемого для считывания строк данных. Значение по умолчанию зависит от системы. Обычно > 65536.                                                |

|                       |                                                                                                                                                                                                                                                   |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| file                  | Имя файла дампа экспорта.                                                                                                                                                                                                                         |
| filesize              | Максимальный размер файла дампа экспорта. Если в элементе <b>file</b> перечислено несколько файлов, результат экспорта будет записываться в них на основе их значений <b>filesize</b> .                                                           |
| compress              | Флаг <b>Y/N</b> , показывающий должна ли утилита экспорта сжимать фрагментированные сегменты в единичные экстенды. Этот параметр влияет на то какие конструкции <b>storage</b> будут содержаться в файле экспорта для этих объектов.              |
| grants                | Флаг <b>Y/N</b> , показывающий будут ли экспортироваться полномочия ( <b>GRANTS</b> - привилегии) на объекты БД.                                                                                                                                  |
| indexes               | Флаг <b>Y/N</b> , показывающий будут ли экспортироваться индексы таблиц.                                                                                                                                                                          |
| rows                  | Флаг <b>Y/N</b> , показывающий будут ли экспортироваться строки. Если значение этого параметра равно <b>N</b> в файле экспорта будут созданы только операторы <b>DDL</b> для объектов базы данных.                                                |
| constraints           | Флаг <b>Y/N</b> , показывающий будут ли экспортироваться ограничения на таблицы.                                                                                                                                                                  |
| full                  | Если значение этого параметра равно <b>Y</b> , будет выполнен полный экспорт БД.                                                                                                                                                                  |
| owner                 | Список экспортируемых учетных записей БД. Для этих учетных записей может быть выполнен экспорт в режиме <b>User</b> .                                                                                                                             |
| tables                | Список экспортируемых таблиц БД. для них может быть выполнен экспорт в режиме <b>Table</b> .                                                                                                                                                      |
| recordlength          | Длина записи файла дампа экспорта в байтах. Обычно оставляет значение по умолчанию, если не предполагается переносить файлы экспорта между различными операционными системами.                                                                    |
| inctype               | Тип выполняемого экспорта. Допустимы значения <b>COMPLETE</b> (по умолчанию) <b>COMULATIVE</b> и <b>INCREMENTAL</b> . (Далее мы рассмотрим это подробнее)                                                                                         |
| direct                | Флаг <b>Y/N</b> , показывающий следует ли выполнять прямой ( <b>Direct</b> ) экспорт. При этом процесс обходит кэш буфера, что позволяет получить существенный выигрыш в производительности.                                                      |
| record                | Для инкрементального экспорта флаг <b>Y/N</b> показывает, будет ли запись об экспорте занесена в таблицы словаря данных.                                                                                                                          |
| parfile               | Имя файла параметров передаваемого утилите экспорта. В этом файле могут содержаться все описанные ранее и позже параметры.                                                                                                                        |
| statistic             | Параметр определяющий надо ли записывать в файл экспорта команду <b>analyze</b> для всех экспортируемых объектов. Допустимы значения <b>COMPUTE</b> , <b>ESTIMATE</b> (по умолчанию), и <b>N</b> . Ранее этот параметр назывался <b>ANALYZE</b> . |
| consistent            | Флаг <b>Y/N</b> , показывающий, сохранять ли для всех экспортируемых объектов вариант, согласований по чтению. Это требуется если в ходе процесса экспорта, связанные друг с другом таблицы модифицируются пользователем.                         |
| log                   | Имя файла, в который будет записан журнал экспорта.                                                                                                                                                                                               |
| feedback              | Число строк, по достижению которого на экране будет отображаться процесс экспорта таблицы. Значение по умолчанию, равно 0, означает, что никакой обратной связи не предусматривается, пока вся таблица не будет полностью экспортирована.         |
| point_in_time_recover | Флаг <b>Y/N</b> , дающий <b>ORACLE</b> понять, что экспортируются метаданные для использования при восстановлении табличного пространства с привязкой ко времени (point_in_time). Эта возможность восстановления является                         |

|                      |                                                                                                                                                              |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                      | углубленной.                                                                                                                                                 |
| recovery_tablespace  | Табличные пространства метаданные, которых будут экспортированы с привязкой ко времени.                                                                      |
| query                | Конструкция <b>where</b> , которая будет применяться к каждой таблице при ее экспорте.                                                                       |
| transport_tablespace | Равен <b>Y</b> , если используется режим сменных табличных пространств (pluggable tablespace). Используется совместно с ключевым словом <b>tablespaces</b> . |
| tablespaces          | Табличные пространства, метаданные которых будут экспортированы при перемещении табличного пространства.                                                     |

Вот собственно список параметров утилиты экспорта. Вообще-то, если присмотреться внимательнее, то можно увидеть, что многие параметры конфликтуют друг с другом. Так, например, если записать **.. FULL=Y OWNER=MILLER**, то получите сообщение об ошибке, так как первый параметр означает полный экспорт, а второй пользовательский со схемой **MILLER**! Параметры **transport\_tablespace** и **tablespaces** являются специфическими и применимы только при работе с так называемыми транспортируемыми табличными пространствами. Тогда как параметр **point\_in\_time\_recover** и **recovery\_tablespace** применим только к восстановлению табличных пространств с привязкой ко времени. Следующий немаловажный фактор. Значение **COMPERESS=Y** изменяет параметр **initial** конструкции **storage** для сегментов с несколькими экстендами. Общее выделенное пространство на диске для таких сегментов сжимается в один экстенд. В связи с этим следует отметить две важные особенности:

Во-первых, сжимается выделенное, а не используемое табличное пространство. Если таблица, для которой выделено 30 Мб в трех экстендах по 10 Мб в каждом, сожмется в один пустой экстенд размером 30 Мб(!) Освобождение дискового пространства при этом не происходит!

Во-вторых, если табличное пространство содержит несколько файлов данных, сегмент может выделить место на диске превышающее по размерам самый большой из них! В результате в конструкции **storage** размер экстенда **initial** будет больше размера самого большого файла данных. Поскольку один экстенд не может охватить больше одного файла данных, создание объекта в процессе импорта закончится неудачей!

Вот такие подводные камушки ждут вас на этом тернистом пути! Теперь вы сами можете попробовать, провести экспорт ваших БД и посмотреть, что у вас получится! Далее мы продолжим! :)

## Шаг 131 - БД Oracle - Экспорт - на примерах

В [прошлый раз](#) мы несколько углубились в теорию связанную с командной строкой экспорта. А сейчас, давайте попробуем, поработать с экспортом БД практически. Следует сразу заметить как вы уже, наверное, поняли файл экспорта **Oracle** нельзя использовать для последующего импорта скажем в **MS SQL** сервер! (Да, хотел бы я посмотреть на лицо Гейтса, если бы это было возможно!) Так, что файл экспорта **Oracle** работает только с **Oracle**! Вот такие подробности! Итак, в [прошлый раз](#) мы с вами выполняли полный экспорт вашей БД с содержимым исполняемого (**.bat**) файла примерно следующего содержания:

```
..\bat->

set nls_lang=russian_cis.ru8pc866
exp.EXE USERID=sys/manager@proba FULL=Y FILE=c:\ORACLE\full.dat LOG=C:\ORACLE\full.LOG
```

Для большей наглядности я бы сделал некоторые изменения в строке экспорта вот так:

```
..\bat->

set nls_lang=russian_cis.ru8pc866
exp.EXE USERID=sys/manager@proba FULL=Y FEEDBACK=1000 FILE=c:\ORACLE\full.dat
LOG=C:\ORACLE\full.LOG
```

В этом случае после экспорта 1000 строк таблички будут появляться точки. Так более понятно, что происходит! А, вот так можно "слить" конкретного пользователя скажем нашего **MILLER**-а:

```
..\bat->

set nls_lang=russian_cis.ru8pc866
exp.EXE USERID=miller/kolobok@proba COMPRESS=Y FILE=c:\ORACLE\miller.dat
LOG=C:\ORACLE\miller.LOG
```

А, за одно объединить его разорванные экстенды. Вот фрагмент log-файла после экспорта:

```
.
.
. . экспорт таблицы SALESREPSЭкспортировано 11 строк
. . экспорт таблицы SPEEDЭкспортировано 0 строк
. . экспорт таблицы TBLAЭкспортировано 0 строк
. . экспорт таблицы TSTSVЭкспортировано 6 строк
. . экспорт таблицы TSTTRIGЭкспортировано 6 строк
. экспорт синонимов
. экспорт представлений
. экспорт хранимых процедур
. экспортируются операторы
. экспорт ссылочных ограничений целостности
. экспорт триггеров
. экспортируются индексные типы
. экспортируются битовые, функциональные и расширяемые индексы
. экспорт посттабличных действий
. экспорт материализованных представлений
```

- . экспорт журналов снимков
- . экспорт очередей заданий
- . экспорт обновл. групп и потомков
- . экспортируются размерности
- . экспортируются post-schema процедурные объекты и действия
- . экспорт статистики

Экспорт завершился успешно без предупреждений.

Все вроде прошло гладко! Или вот, например, слить таблички **CUSTOMERS** и **SALESREPS** из схемы **MILLER**:

..\bat->

```
set nls_lang=russian_cis.ru8pc866
exp.EXE USERID=miller/kolobok@proba TABLES=(CUSTOMERS, SALESREPS)
FILE=c:\ORACLE\Tmiller.dat
LOG=C:\ORACLE\Tmiller.LOG
```

Вот еще фрагмент **log**-файла, после экспорта в табличном режиме:

Экспорт сделан с код.таблицами RU8PC866 и AL16UTF16 NCHAR  
сервер использует кодировку CL8MSWIN1251 (возможно перекодирование)

Готовиться экспорт указанных таблиц через обычный маршрут ...

|                     |                         |          |
|---------------------|-------------------------|----------|
| . . экспорт таблицы | CUSTOMERSЭкспортировано | 22 строк |
| . . экспорт таблицы | SALESREPSЭкспортировано | 11 строк |

EXP-00091: Экспортируется недостоверная статистика

EXP-00091: Экспортируется недостоверная статистика

Экспорт завершен успешно с предупреждениями.

Последняя строка свидетельствует о том, что все прошло удачно!!! А, вот теперь благодаря прежде всего вашему терпению, мы с вами и дошли, до повторения пройденного! Вспомните ["Шаг 8 - Пересоздаем пользователя miller"](#)! Там были файлики, как раз с дампом экспорта! Которые я подготовил, для вас! В табличках кстати много недочетов по части русского языка! Я уже потом заметил! Очень извиняюсь! Ну не могу я никак великий и могучий понять до конца!!! :))) Так вот, там был еще **.bat** файл такого содержания:

..\miller.bat->

```
set nls_lang=russian_cis.ru8pc866
imp.EXE USERID=MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

Здесь как вы уже наверное догадались я провожу уже импорт данных в вашу БД из файла дампа **miller.dat** от пользователя **miller** пользователю **miller**! Не совсем ясная ситуация, но я позже все поясню! Один интересный момент я бы хотел отметить особенно! Мне уже много раз приходили письма с вопросом откуда, закачать эти три файла! И, что самое смешное некоторые даже спрашивали: - Может Вы за плату их отсылаете? Странно! :) Но я подумаю! Так вот! Еще раз обращаю внимание - внизу HTML странички есть ссылка "Загрузить проект"! Нажмите ее и получите файлы совершенно бесплатно!!! :) Еще немного дополню, многие как я понял при создании сетевой службы при установке **Oracle (TNS Names** - в нашем случае **proba**) используют отличное имя от нашего - **proba**! И при импорте получают ошибку! Так вот, если вы производите импорт



непосредственно на сервере, а как правило так и происходит, то имя сетевой службы можно вообще не писать - сервер вас поймет! Вот так:

```
set nls_lang=russian_cis.ru8pc866
imp.EXE USERID=MILLER/KOLOBOK FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

Если вы проводите импорт с клиентской машины, то имя сетевой службы нужно записывать и именно, то которое у вас в системе! Дело все в том, что при инсталляции по умолчанию, имя сетевой службы совпадает с **SID** вашего экземпляра БД, но и может быть отличным от него! Как создавать сетевую службу смотрите внимательнее ["Шаг 4 - Настройка с помощью Net8 Easy Config"](#)! Если что-то еще будет не ясно, я наверное сделаю еще пару отдельных шагов на эту тему! Так что не путайтесь и будьте внимательны при работе с экспортом! Удачи!!! :)

---

## Шаг 132 - БД Oracle - Экспорт данных - окончание

Теперь давайте рассмотрим несколько немаловажных, моментов при работе с экспортом. Что такое вообще полный экспорт и инкрементальный и кумулятивный экспорт? Все упирается в параметр экспорта **INCTYPE** используемый совместно с параметром **FULL**. Все это в совокупке позволяет администратору БД экспортировать только те таблицы, которые изменились со времени последнего экспорта. Если изменилась хотя бы одна строка таблицы, то в ходе инкрементального или кумулятивного экспорта будут экспортированы все ее строки. Рассмотрим доступные значения **INCTYPE**:

| Значение<br><b>INCTYPE</b> | Описание                                                                                                                                                                                        |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| COMPLETE                   | Значение по умолчанию, экспортированы будут все таблицы.                                                                                                                                        |
| COMULATIVE                 | Его можно указать если <b>FULL=Y</b> . Экспортированы будут только те таблицы, которые содержат строки изменившиеся со времени последнего полного экспорта любого типа.                         |
| INCREMENTAL                | Его можно указать если <b>FULL=Y</b> . Экспортированы будут только те таблицы, которые содержат строки изменившиеся со времени последнего кумулятивного, полного или инкрементального экспорта. |

Строго говоря, основу стратегии экспорта при резервном копировании формирует полный экспорт. Инкрементальный и кумулятивный экспорт могут быть полезны, если изменились только некоторые таблицы базы данных и размер этих таблиц не велик! Хотя мое мнение, слил все в полный экспорт и голова не болит какие таблицы менялись или нет, главное гарантия, что я полностью восстановлю БД после сбоя! Собственно решать вам использовать данные методы или нет! Следующий вопрос это согласованный экспорт. Так как экспорт к великой радости можно проводить не останавливая экземпляр БД, (хотя можно сделать и на оборот!), а прямо во время работы вашего экземпляра, то возникает вывод, что при записи в таблицу информации из БД в файл дампа экспорта, утилита экспорта считывает только по одной таблице за единицу времени! Таким образом хотя экспорт и начинается в определенный момент времени, каждая таблица считывается в свое, отличающееся от времени начала экспорта. При этом экспортируются данные существующие в таблице на тот момент, когда утилита экспорта начинает считывать именно эту таблицу! Поскольку большинство таблиц связаны между собой и имеют различные размеры, это может привести к несогласованности экспортируемых данных. Как же решить эту проблему!? Есть два способа как с этим справиться! Во-первых, можно запланировать экспорт на то время когда никого нет на рабочих местах - часика эдак в три ночи! Здорово! Хотя думаю это вам не понравится! А, что если БД в работе в режиме 365X24! Тогда есть еще один способ! Использовать параметр **CONSISTENT**. Этот параметр доступен только при полном экспорте. Инкрементальный и кумулятивный в данном случае не прокатывает! Если **CONSISTENT=Y**, то БД будет работать с сегментами отката и отслеживать все изменения сделанные с момента начала экспорта. Затем с помощью сегментов отката можно воссоздать данные существующие на тот момент. В результате экспортируемые данные будут согласованы, но ценой снижения производительности. Вообще по возможности обеспечивайте согласованность данных экспорта, проводя экспорт когда БД, не используется или установлена в режиме **restricted session**. При невозможности используйте экспорт с параметром **CONSISTENT=Y** для модифицируемых таблиц и **CONSISTENT=N** для полной БД. При таком способе не будет слишком низкой производительности, но обеспечите согласованность данных вашей БД. Так же с помощью утилиты экспорта и последующего импорта возможно проводить, работу с табличными пространствами и перемещениями объектов схем и самих табличных пространств. Думаю, в дальнейшем попробовав несколько сценариев, экспорта вы сможете сами

научиться проводить, его так как это вам необходимо, тем более что изложенного вам должно хватить в избытке! Можете попробовать и рассказать мне, что у вас получилось! ;-)

---

## Шаг 133 - БД Oracle - Импорт данных - параметры импорта

Раз существует экспорт - разумно предположить, что должен быть и импорт данных в БД **Oracle**. Давайте разберемся теперь с этим вопросом. Итак, мы экспортировали данные получили двоичный(!) файл с данными, который невозможно подпихнуть в какую-либо БД отличную от **Oracle**. Что ж, хорошо! А, вот теперь давайте попробуем импортировать данные в БД из нашего двоичного файла! В данном конкретном случае утилита импорта считывает файл дампа экспорта и запускает находящиеся в нем команды на исполнение. Кроме всего прочего импорт можно применять для селективного восстановления объектов или пользователей из файла дампа экспорта. При этом учтите тот момент, что импортируя данные из инкрементального или кумулятивного экспорта, нужно сначала импортировать результаты самого последнего инкрементального экспорта, а затем самого последнего полного экспорта. После этого можно импортировать файл самого последнего кумулятивного экспорта и все результаты инкрементальных экспортов, имевших место после него. Витиеватая формулировочка! Но, если прочитать пару раз, то все понятно! Давайте сразу определимся с параметрами импорта для полного понимания, как он работает (которые кстати схожи с параметрами экспорта, но и имеют, кое какие отличия от последнего!). Итак вот они:

| Ключевое слово | Описание                                                                                                                                                                                                            |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| userid         | Имя пользователя и пароль учетной записи, выполняющей импорт. Если он следует первым в командной строке <b>imp.exe</b> , то <b>userid</b> писать не обязательно!                                                    |
| buffer         | Размер буфера используемого для считывания строк данных. Значение по умолчанию зависит от системы. Обычно < 100000.                                                                                                 |
| file           | Имя импортируемого файла для дампа экспорта.                                                                                                                                                                        |
| show           | Флаг <b>Y/N</b> , определяющий нужно ли отображать или исполнять содержание файла дампа экспорта.                                                                                                                   |
| ignore         | Флаг <b>Y/N</b> , определяющий должна ли утилита импорта игнорировать ошибки, возникающие при выполнении команды <b>CREATE</b> . Он используется, если импортируемые объекты уже существуют.                        |
| grants         | Флаг <b>Y/N</b> , показывающий будут ли импортироваться полномочия ( <b>GRANTS</b> - привилегии) на объекты БД.                                                                                                     |
| indexes        | Флаг <b>Y/N</b> , показывающий будут ли импортироваться индексы таблиц.                                                                                                                                             |
| constraints    | Флаг <b>Y/N</b> , показывающий будут ли импортироваться ограничения на таблицы.                                                                                                                                     |
| rows           | Флаг <b>Y/N</b> , показывающий будут ли импортироваться строки. Если значение этого параметра равно <b>N</b> будут выполнены только операторы <b>DDL</b> для объектов базы данных.                                  |
| full           | Если значение этого параметра равно <b>Y</b> , импортируется файл дампа полного экспорта БД.                                                                                                                        |
| fromuser       | Список учетных записей БД, объекты которых должны быть считаны из файла дампа экспорта. (при этом <b>full = N</b> )                                                                                                 |
| touser         | Список учетных записей БД, в которые следует импортировать объекты из файла дампа экспорта. Параметры <b>fromuser</b> и <b>touser</b> не обязательно должны иметь одно и тоже значение. ( <b>full = N</b> при этом) |
| tables         | Список импортируемых таблиц БД.                                                                                                                                                                                     |
| recordlength   | Длина в байтах записи файла дампа экспорта. Обычно оставляет значение по умолчанию, если вы не собираетесь передавать файл экспорта между                                                                           |

|                        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                        | различными операционными системами.                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| inctype                | Тип выполняемого импорта. Допустимы значения <b>COMPLETE</b> (по умолчанию), <b>COMULATIVE</b> и <b>INCREMENTAL</b> .                                                                                                                                                                                                                                                                                                                                                        |
| commit                 | Флаг <b>Y/N</b> , показывающий должен ли импорт завершаться командой <b>COMMIT</b> после ввода каждого массива (размер которого определяется в параметре <b>BUFFER</b> ). Если он равен <b>N</b> , то импорт будет завершаться командой <b>COMMIT</b> после ввода каждой таблицы. Для больших таблиц значение <b>COMMIT = N</b> будет требовать такого же по размеру сегмента отката!                                                                                        |
| parfile                | Имя файла параметров передаваемого утилите импорта. В этом файле могут содержаться все описанные ранее и позже параметры.                                                                                                                                                                                                                                                                                                                                                    |
| indexfile              | Очень мощное средство, (К стати очень полезное!) позволяющее записывать все команды <b>CREATE TABLE</b> , <b>CREATE CLUSTER</b> , <b>CREATE INDEX</b> в файл, а не выполнять их! Все команды кроме <b>CREATE INDEX</b> , будут закомментированы. Затем (внеся незначительные изменения) этот файл можно запустить после завершения импорта с параметром <b>indexes = N</b> . Средство очень полезно для изолирования таблиц и индексов по различным табличным пространствам. |
| charset                | Набор символов, используемы при импорте данных для v5 и v6 (данный параметр больше не употребляется, но сохранен)                                                                                                                                                                                                                                                                                                                                                            |
| point_in_time_recover  | Флаг <b>Y/N</b> , показывающий входит ли импорт в состав процесса восстановления табличного пространства с привязкой ко времени. Далее будет рассматриваться в разделе "Архивация и восстановление" (если у меня хватит сил на все это! :).                                                                                                                                                                                                                                  |
| destroy                | Флаг <b>Y/N</b> , показывающий будут ли выполняться команды <b>CREATE TABLESPACE</b> , обнаруженные в файлах дампа полных экспортов. (Это приведет к уничтожению файла данных в БД, в которую осуществляется импорт или проще сказать он будет полностью переписан!)                                                                                                                                                                                                         |
| log                    | Имя файла, в который будет записан журнал импорта.                                                                                                                                                                                                                                                                                                                                                                                                                           |
| skip_unusable_indexes  | Флаг <b>Y/N</b> , показывающий следует ли в процессе импорта пропускать индексы разделов, помеченные, как неиспользуемые. Чтобы повысить производительность создания индексов, можно пропустить этот этап и создать индексы в последствии в ручную.                                                                                                                                                                                                                          |
| analyze                | Флаг <b>Y/N</b> , показывающий следует ли в процессе импорта выполнять команды <b>analyze</b> , обнаруженные в файлах дампа экспорта.                                                                                                                                                                                                                                                                                                                                        |
| feedback               | Количество строк, по достижению которого на экране будет отображаться процесс импорта таблицы. Значение по умолчанию, равно 0, означает, что никакой обратной связи не предусматривается, пока вся таблица не будет полностью импортирована.                                                                                                                                                                                                                                 |
| tiod_novallidate       | Позволяет в процессе импорта пропустить проверку указанных типов объектов на допустимость. используется при установке картриджей ( <b>cartridge installs</b> ). Можно указать один или несколько типов объектов.                                                                                                                                                                                                                                                             |
| filesize               | Максимальный размер дампа указанный для экспорта. Если в ходе экспорта использовался параметр <b>FILESIZE</b> .                                                                                                                                                                                                                                                                                                                                                              |
| recalculate_statistics | Флаг <b>Y/N</b> , показывающий следует ли генерировать статистику оптимизатора.                                                                                                                                                                                                                                                                                                                                                                                              |
| transport_tablespace   | Флаг <b>Y/N</b> , показывающий следует ли импортировать в БД метаданные транспортируемых табличных пространств.                                                                                                                                                                                                                                                                                                                                                              |
| tablespaces            | Имя табличного пространства или список имен таковых транспортируемых в БД.                                                                                                                                                                                                                                                                                                                                                                                                   |

|           |                                                                                 |
|-----------|---------------------------------------------------------------------------------|
| datafiles | Список файлов данных транспортируемых в БД.                                     |
| tts_owner | Имя или список имен владельцев данных транспортируемых табличных пространствах. |

Здесь достаточно много интересных моментов, взять хотя бы параметр - **indexfile**! Далее давайте попробуем немного поработать с утилитой **imp.exe**. А пока можете более внимательно почитать все вышеизложенное.

---

## Шаг 134 - БД Oracle - Импорт (экспорт) в примерах

Для полноты картины напомним, что некоторые параметры утилиты импорта могут конфликтовать друг с другом. Например, если написать - **FULL = Y** и **FROMUSER = MILLER**, то получите ошибку! Так же параметр **DESTROY** может быть очень полезным для админов БД, которые работают с несколькими БД, на одном сервере. Поскольку в процессе полного экспорта БД, осуществляется запись всего словаря данных. А, в файл дампа экспорта, заносятся определения табличных пространств и файлов данных. При этом, если файл дампа экспорта используется для миграции в другую БД того же сервера у вас могут возникнуть проблемы. Это связано с тем что при импорте результатов полного экспорта, первой БД во вторую будут выполнены команды **CREATE TABLESPACE**, обнаруженные в файле дампа экспорта. Эти команды создадут такие же файлы во второй БД, в результате чего, если не указать параметр **DESTROY = N** файлы первой БД могут быть переписаны и ее данные будут потеряны! Избежать этого можно заранее создав табличные пространства и разделив их каталоги! Тогда ничего страшного не произойдет! А, сейчас давайте вспомним еще раз наш с вами первый импорт при создании пользователя **MILLER**:

```
rem Эта строка предотвращает появление вражеских кабалистических знаков при работе с
командной строкой!
set nls_lang=russian_cis.ru8pc866
```

```
rem А это собственно строка, выполняющая импорт!
imp.EXE USERID=MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

Здесь хорошо видно, что файл дампа экспорта имеет имя **MILLER.DAT**. При этом производится импорт данных, от имени пользователя **MILLER** моей БД, которую я создал для работы с шагами в вашу БД тому же пользователю **MILLER**. В данном случае работают операторы **fromuser** и **touser**. Данную строку можно было переписать и вот так:

```
imp.EXE MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

Так как данные пользователя БД, идут первыми в командной строке и запись **USERID** при этом не требуется. А, если импорт проводится прямо на машине сервера, то можно записать еще проще:

```
imp.EXE MILLER/KOLOBOK FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

В данном случае имя сетевой службы нет необходимости указывать, так как "ухо" сервера вашей БД и так поймет что от него хотят! Если у вас есть желание, то можете попробовать добавить в эту строку что-то еще из параметров [прошлого шага](#) и посмотреть, что будет получаться! Теперь, что касается команды **COMMIT** для выполнения импорта данных. Допустим в вашем дампе экспорта есть таблица на 300 Мбайт данных, это совсем не значит, что необходимо иметь сегмент отката такой же величины! Для того, чтобы уменьшить размеры элементов сегментов отката при выполнении данного импорта определите **COMMIT = Y** и задайте значение параметра **BUFFER**. Теперь команда **COMMIT** будет выполняться после каждого ввода информации объемом **BUFFER**. Например:

```
imp.EXE MILLER/KOLOBOK FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER
```

В данном случае **COMMIT** будет выполняться после ввода каждой таблицы, а в следующем примере:

```
imp.EXE MILLER/KOLOBOK FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER BUFFER = 64000
COMMIT = Y
```

**COMMIT** будет срабатывать каждые 64000 бит. Но учтите при этом, что наличие полей типа **LOB** или **BLOB** в вашем файле экспорта БД, может значительно превысить 64000 бит для одной порции данных импорта по этому не забывайте про это при определении параметра **BUFFER** импортируя данные с использованием больших двоичных объектов! Здесь следует увеличивать данный параметр, то минимально необходимого объема! Если при этом будет возвращена ошибка **IMP-00020**, значит параметр **BUFFER** не достаточно велик. Увеличивайте его значение и повторяйте импорт. Но при этом, если импорт таблицы закончился неудачно, некоторые ее строки все же могут оказаться импортированными. Следует удалить такую "частичную" таблицу перед повторным импортированием! А, что если скажем не удалось сразу импортировать все объекты БД? Что, делать в этом случае? При импорте таблицы создаются в алфавитном порядке! При этом могут возникнуть проблемы из-за существования зависимостей в таблицах БД. Если, например, утилита импорта пытается создать представление таблицы, которой еще не существует, то будет выдано сообщение об ошибке! В данном случае импорт следует повторить, задав параметры **ROWS = N** и **IGNORE = Y**! Это позволит создать только те структуры данных, которые не были созданы при последнем сеансе импорта! Например, вы дали такую команду для строки импорта:

```
imp.EXE MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FULL = Y COMMIT = Y BUFFER = 64000
```

После этого вы получили сообщение об ошибке **ORA-00942**(таблица или представление не существуют)! Что делать дальше? Все просто! Даем вот такую строку:

```
imp.EXE MILLER/KOLOBOK@PROBA FILE=MILLER.DAT FULL = Y IGNORE = Y ROWS = N COMMIT = Y
BUFFER = 64000
```

При этом не забывайте о связи объектов БД! Я вам о них рассказывал ранее! Например, если вы удалили таблицу, а представление созданное от нее осталось в словаре данных, то последующий импорт так же будет неудачным в следствии того, что представлению некуда ссылаться! Не забывайте об этом! Следует быть внимательным при работе с объектами БД и их экспортом для последующего импорта в другую или в эту же БД! Теперь давайте рассмотрим еще один параметр при импорте данных, а именно **INDEXES** и **INDEXFILE**. Эти параметры позволяют "развести" при необходимости таблицы и индексы по разным табличным пространствам. При использовании **INDEXFILE** файл данных при импорте будет прочитан, но не импортирован. Все сценарии создания таблиц будут записаны в файл на выходе. При этом в сценарий, который будет создан, все конструкции **DDL** будут закомментированы и в дальнейшем на его основе изменив параметры **storage** и **tablespace** вы можете переопределить местоположение таблиц и индексов. Так же следует отметить, что использование **INDEXFILE** требует либо определить параметр **FROMUSER**, либо параметр **FULL** значением **Y**! Рассмотрим это на примере. Первое проводим экспорт пользователя **MILLER** вот так:

```
.
.
exp.EXE USERID=MILLER/KOLOBOK FILE=MILLER.DAT OWNER=MILLER
.
```

Получаем файл **MILLER.DAT**. Затем напишите **bat** файл вот с такой командой импорта данного пользователя:

```
.
.
imp.exe USERID=MILLER/KOLOBOK FILE=MILLER.DAT indexfile=milleridx.sql full=y
.
```



На экране должно получиться, что-то вроде:

```
Соединен с: Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9.2.0.1.0 - Production

Экспорт-файл создан EXPORT:V09.02.00 через обычный маршрут
импорт выполнен в кодировке RU8PC866 и AL16UTF16 кодировке NCHAR
импортирующий сервер использует кодировку CL8MSWIN1251 (возможно перекодирование)
. . пропускается таблица "ADT"
. . пропускается таблица "BOYS"
. . пропускается таблица "CUSTOMERS"
. . пропускается таблица "GIRLS"
. . пропускается таблица "MYAUDIT"
. . пропускается таблица "OFFICES"
. . пропускается таблица "ORDERS"
. . пропускается таблица "PRODUCTS"
. . пропускается таблица "SALESREPS"
. . пропускается таблица "SPEED"
. . пропускается таблица "TBLA"
. . пропускается таблица "TSTSV"
. . пропускается таблица "TSTTRIG"

Импорт завершился успешно без предупреждений.
```

Затем отредактируйте как вам нужно файл сценария **milleridx.sql** изменив в нем конструкции **STORAGE** и **TABLESPACE**. Вот примерное содержимое этого файла, которое получилось у меня:

```
.
.
REM ALTER TABLE "MILLER"."CUSTOMERS" ADD PRIMARY KEY ("CUST_NUM") USING
REM INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 STORAGE(INITIAL 81920
REM FREELISTS 1 FREELIST GROUPS 1) TABLESPACE "USERS" LOGGING ENABLE ;
REM CREATE TABLE "MILLER"."GIRLS" ("NM" NUMBER(*,0), "NAME" VARCHAR2(20),
REM "CITY" VARCHAR2(20)) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
REM STORAGE(INITIAL 81920 FREELISTS 1 FREELIST GROUPS 1) TABLESPACE
REM "USERS" LOGGING NOCOMPRESS ;
REM ... 6 rows
REM CREATE TABLE "MILLER"."OFFICES" ("OFFICE" NUMBER(*,0), "CITY"
REM VARCHAR2(30) NOT NULL ENABLE, "REGION" VARCHAR2(30) NOT NULL ENABLE,
REM "MGR" NUMBER(*,0), "TARGET" NUMBER, "SALES" NUMBER NOT NULL ENABLE)
REM PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 STORAGE(INITIAL 81920
REM FREELISTS 1 FREELIST GROUPS 1) TABLESPACE "USERS" LOGGING NOCOMPRESS ;
REM ... 5 rows
.
.
```

Затем запустите сценарий создания таблиц и индексов в **SQL\*Plus** и проведите импорт данных с такими параметрами:

```
imp.exe USERID=MILLER/KOLOBOK FILE=MILLER.DAT FROMUSER=MILLER TOUSER=MILLER BUFFER
= 64000
COMMIT = Y INDEXES = N
```

Но если количество индексов для отделения не большое, то можно воспользоваться опцией **rebuild** команды **alter index**. Вот собственно кратко о том, как выполняется импорт и экспорт в БД **Oracle**! Все остальное я надеюсь вы сможете одолеть и сами, если есть желание и время! И конечно же, если что-то не совсем ясно, можете спрашивать! Чем могу помочь!

---

## Шаг 135 - БД Oracle - Загрузка данных SQL\*Loader

Импорт и экспорт данных это конечно здорово! А, как вообще после создания БД, в нее "залить" данные из уже имеющейся БД (например **Microsoft Access** или **Microsoft SQL Server** - дай бог здоровья Билу!)? Какие данные и в каком виде можно отправлять в БД **Oracle**? Все очень просто, в **Oracle** есть такой инструмент, как **SQL\*Loader**. На первый взгляд он может показаться примитивным и каким-то не серьезным, но это только на первый взгляд! Но тем не менее загрузчик **SQL\*Loader** может получать данные следующим образом:

1. Данные могут загружаться из нескольких файлов данных.
2. Входные данные могут быть фиксированной или переменной длины.
3. За один вызов можно загрузить много таблиц. Возможна так же загрузка только выбранных записей - каждой в соответствующую таблицу.
4. Перед загрузкой в таблицу данные можно обрабатывать средствами языка **SQL**.
5. Несколько физических записей могут быть объединены в одну логическую и наоборот - при помощи **SQL** можно выбрать одну физическую запись и загрузить ее в несколько логических.
6. Так же поддерживаются вложенные, столбцы вложенных таблиц. Массивы **VARRAY** и объектов **LOB** (включая **BLOB**, **CLOB**, **NCLOB** и **BFILE**).

Обычно для начала работы вызывается загрузчик из командной строки по одному из следующих имен: **sqlldr**, **sqlload**, **sqlldr80**.

В разных операционных системах, могут быть разные имена загрузчиков **SQL\*Loader**. Итак, давайте попробуем для начала получить список параметров **SQL\*Loader**. Для удобства я создал **bat** файл следующего содержания:

```
\\.. .bat ->

set nls_lang=russian_cis.ru8pc866

sqlldr.exe >helloloader.txt
```

А, вот содержимое файла **helloloader.txt**:

```
.
.
Синтаксис: SQLLDR ключ.слово=значение [,ключ.слово=значение,...]

Допустимые ключевые слова:

userid -- ORACLE username/password
control -- Control file name
log -- Log file name
bad -- Bad file name
data -- Data file name
discard -- Discard file name
discardmax -- Number of discards to allow (По умолчанию - все)
skip -- Number of logical records to skip (По умолчанию 0)
load -- Number of logical records to load (По умолчанию - все)
errors -- Number of errors to allow (По умолчанию 50)
```

rows -- Number of rows in conventional path bind array or between direct path data saves  
(По умолчанию: условный маршрут - 64, прямой маршрут - ALL)

bindsize -- Size of conventional path bind array in bytes (По умолчанию 256000)  
silent -- Suppress messages during run (header, feedback, errors, discards, partitions)  
direct -- use direct path (По умолчанию FALSE)  
parfile -- parameter file: name of file that contains parameter specifications  
parallel -- do parallel load (По умолчанию FALSE)  
file -- File to allocate extents from

skip\_unusable\_indexes -- disallow/allow unusable indexes or  
index partitions (По умолчанию FALSE)

skip\_index\_maintenance -- do not maintain indexes, mark affected indexes  
as unusable (По умолчанию FALSE)

readsize -- Size of Read buffer (По умолчанию 1048576)

external\_table -- use external table for load; NOT\_USED,  
GENERATE\_ONLY, EXECUTE (По умолчанию NOT\_USED)

columnarrayrows -- Number of rows for direct path column array (По умолчанию 5000)  
streamsize -- Size of direct path stream buffer in bytes (По умолчанию 256000)  
multithreading -- use multithreading in direct path  
resumable -- enable or disable resumable for current session (По умолчанию FALSE)  
resumable\_name -- text string to help identify resumable statement  
resumable\_timeout -- wait time (in seconds) for RESUMABLE (По умолчанию 7200)  
date\_cache -- size (in entries) of date conversion cache (По умолчанию 1000)

Примечание: параметры командной строки могут быть заданы, либо по позициям позиции или по ключевым словам. Примером предыдущего случая является 'sqlldr scott/tiger foo'; а во втором случае: 'sqlldr control=foo userid=scott/tiger'. Параметры, заданные по позициям, могут стоять перед, но не после параметров, заданных ключевыми словами. Например запись 'sqlldr scott/tiger control=foo logfile=log' разрешен, а 'sqlldr scott/tiger control=foo log' не разрешен, несмотря на то, что позиция параметра 'log' является правильной.

.

Для более полного понимания приведу перевод назначения параметров, который вы в дальнейшем можете использовать как справку! Хотя поучить английский тоже не помешает! Так как без него как без рук! Итак:

- **userid** - Пользователь проводящий загрузку данных. (ORACLE username/password)
- **control** - Имя контрольного файла. (Control file name)
- **log** - Имя файла журнала. (Log file name)
- **bad** - Имя файла, в который будут помещены некорректные данные. (Bad file name)
- **data** - Имя файла с данными. (Data file name)
- **discard** - Имя файла с отброшенными данными. (Discard file name)
- **discardmax** - Допустимое количество отброшенных записей. (Number of discards to allow (По умолчанию - все))
- **skip** - Допустимое количество пропущенных записей. (Number of logical records to skip (По умолчанию 0))

- **load** - Количество записей которые должны быть загружены. (Number of logical records to load (По умолчанию - все))
- **errors** - Допустимое количество ошибочных записей. (Number of errors to allow (По умолчанию 50))
- **rows** - Количество строк в массивах привязки для обычной загрузки или количество сохраняемых за один раз строк в случае прямой загрузки. (Number of rows in conventional path bind array or between direct path data saves (По умолчанию: условный маршрут - 64, прямой маршрут - ALL))
- **bindsize** - Размер в байтах массива привязки для обычного метода загрузки. (Size of conventional path bind array in bytes (По умолчанию 256000))
- **silent** - Подавление вывода сообщений при выполнении (заголовки, обратная связь, ошибки, отброшенные записи, разделы) (Suppress messages during run (header, feedback, errors, discards, partitions))
- **direct** - Использовать прямой метод. (use direct path (По умолчанию FALSE))
- **parfile** - Имя файла с параметрами. (parameter file: name of file that contains parameter specifications)
- **parallel** - Выполнять загрузку в параллельном режиме. (do parallel load (По умолчанию FALSE))
- **file** - Файл с загружаемыми фрагментами данных. (File to allocate extents from)
- **skip\_unusable\_indexes** - Запретить или разрешить неиспользуемые индексы или индексные разбиения. (disallow/allow unusable indexes or index partitions (По умолчанию FALSE))
- **skip\_index\_maintenance** - Не обрабатывать индексы помеченные как неиспользуемые. (do not maintain indexes, mark affected indexes as unusable (По умолчанию FALSE))
- **readsize** - Размер буфера чтения в байтах. (Size of Read buffer (По умолчанию 1048576))
- **external\_table** - Использовать внешние таблицы для загрузки данных. (use external table for load; NOT\_USED, GENERATE\_ONLY, EXECUTE (По умолчанию NOT\_USED))

Последние семь параметров мы рассмотрим чуть позже в процессе, а пока, то что, мы с вами рассмотрели хватит для того, чтобы начать работать с **SQL\*Loader**-ом. Для начала, я думаю достаточно, далее мы продолжим разбирать, как все это работает и будем использовать этот шаг еще многократно!

---

## Шаг 136 - БД Oracle - SQL\*Loader структура загрузчика

Для начала давайте посмотрим на саму структуру загрузчика **SQL\*Loader**. Она представлена ниже на рисунке:

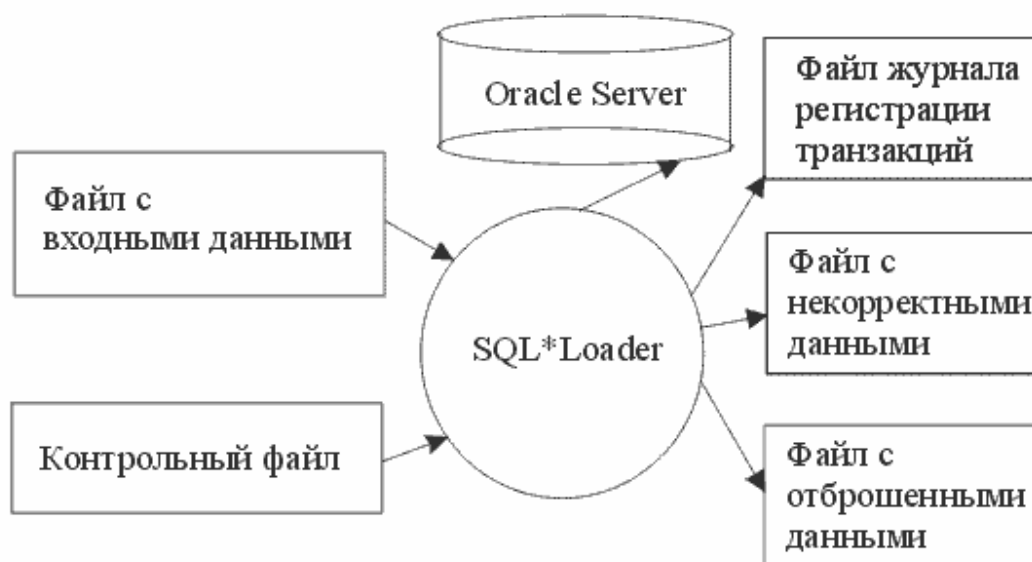


Рис. 1 Компоненты загрузчика SQL\*Loader

Здесь, так называемый, "контрольный файл" - является основным звеном в работе загрузчика. Он задает параметры преобразования информации, в данные размещаемые в таблицах БД. При этом типы данных во внешних файлах могут не совпадать с типами данных в соответствующих столбцах таблиц вашей БД. Как правило при этом производится неявное преобразование типов. В контрольном файле используется специальный язык определения данных **SQL\*Loader DDL**. Собственно сам загрузчик **SQL\*Loader** может воспринимать файлы данных в различных форматах - переменной длины, фиксированной длины и т.д. Это что касается файлов с входными данными. Файл некорректных данных, получает данные отвергнутые загрузчиком при чтении. При этом имеется две стадии определения корректности данных загрузки. Первая это проверка соответствия формата данных спецификации заданной в контрольном файле. Если имеются несоответствия, то данные помещаются в файл некорректных данных. Вторая стадия это когда сама БД, отвергает записи при загрузке, скажем, по причине нарушения ограничений ссылочной целостности. Такие данные так же помещаются в файл некорректных данных. При этом что, самое интересное, что некорректные данные помещаются в том же формате, что и исходные данные и после устранения недочетов все можно повторить снова и залить данные уже полностью. Файл отброшенных данных, формируется если при записи в БД, имеются данные не удовлетворяющие некоторому условию, заранее заданному в начале. При этом количество отброшенных записей не ограничивается, если есть такая необходимость это можно сделать, установив соответствующий параметр при загрузке. И последнее это журнальные файлы. Они формируются при загрузке данных и отражают весь ход процесса загрузки. Если по какой-либо причине создать файл загрузки не удастся, то **SQL\*Loader** не будет запущен! Файл журнала имеет тоже имя, что и контрольный файл, но при этом его расширение меняется на **log**.

Теперь давайте рассмотрим контрольный файл и процесс и правила его создания. Контрольный файл как правило начинается ключевыми словами:

## LOAD DATA

Далее следует имя файла с данными для последующей загрузки:

INFILE 'mydata.dat'

или

INFILE 'loaddata.txt'

Прописывать кавычки для имени файла не обязательно, но желательно для избежания неоднозначности чтения контрольного файла. Далее идет метод загрузки данных в таблицу. Их всего четыре:

| Метод    | Описание                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| INSERT   | Метод используемый по умолчанию. При этом предполагается, что таблица перед загрузкой пустая! Если в таблице есть строки данных, то выполнение загрузки <b>SQL*Loader</b> -ом будет прекращено!                                                                                                                                                                                                                                                               |
| APPEND   | Этот метод позволяет добавлять строки в таблицу таким образом, чтобы они не оказывали воздействия на уже существующие строки данных.                                                                                                                                                                                                                                                                                                                          |
| REPLACE  | При использовании этого метода вначале удаляются все имеющиеся в таблице строки, а затем загружаются новые. Заметим, что при удалении строк срабатывают триггеры установленные на удаление данных в этой таблице.                                                                                                                                                                                                                                             |
| TRUNCATE | Для удаления старых строк используется <b>SQL</b> команда <b>TRUNCATE</b> . Она намного удобнее чем <b>REPLACE</b> поскольку при срабатывании триггеров удаления не генерируются сегменты отката транзакции. Но команда <b>TRUNCATE</b> является необратимой, так как происходит усечение таблицы и установка указателя таблицы на нулевую отметку. При применении данного способа требуется так же убрать все ограничения ссылочной целостности с таблиц БД. |

Дальше идет определение таблицы для загрузки данных вот в таком формате:

INTO TABLE имя\_таблицы (метод)

В данном случае "метод" это то, что мы описали выше, но применяться он будет только к указанной таблице. Далее Идет спецификация полей для загрузки данных. Например:

(поле1, поле2, ..... поле-n)

Вот так собственно строится контрольный файл для загрузки данных в таблицы БД **Oracle**. Для примера давайте я покажу вам как выглядит контрольный файл для таблички нашей учебной БД **PRODUCTS**. Вот его внутренность:

..\products.ctl->

LOAD DATA

INFILE 'products.dat'

```
INTO TABLE PRODUCTS
```

```
FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS
```

```
(MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND)
```

Здесь хорошо видны все секции необходимые для выполнения загрузки данных. Далее, мы с вами рассмотрим примеры загрузки данных в таблицы БД, на основе нашей учебной БД и не только! А пока усваивайте пройденное.

---



## Шаг 137 - БД Oracle - SQL\*Loader Способы загрузки данных в БД

Что-же давайте после целой кучи теории, наконец, перейдем к практике! Для начала рассмотрим загрузку данных фиксированной длины. Это означает, что **SQL\*Loader** будет рассматривать каждую строку в соответствии с положением символа в ней! Для начала давайте очистим от данных, таблицу **PRODUCTS** тем более, что я наделал там кучу ошибок - за одно и их исправим! Итак, запускаем **SQL\*Plus** и вводим команды (схема **MILLER** пароль **KOLOBOK!**):

```
DELETE FROM PRODUCTS
/

COMMIT
/
```

Получаем:

```
SQL> DELETE FROM PRODUCTS
2 /
```

25 строк удалено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Затем в подкаталоге **C:\Oracle\ora81\bin** создаем подкаталог **LOAD** (как на ваше усмотрение, например, **md LOAD**). Заходим в этот каталог и создаем в нем файл **PRODUCTS.bat** вот с таким содержанием:

```
set nls_lang=russian_cis.ru8pc866
```

```
sqlldr.exe userid=miller/kolobok@proba control=PRODUCTS.ctl errors=100 bad=PRODUCTS.bad
```

Теперь самое интересное! Создаем файл загрузки для считывания данных фиксированной длины **PRODUCTS.ctl**. Он будет иметь вот такое содержание:

```
LOAD DATA

INFILE 'PRODUCTS.DAT'

INTO TABLE PRODUCTS

(
 MFR_ID POSITION(01:03) CHAR,
 PRODUCT_ID POSITION(04:10) CHAR,
 DESCRIPTION POSITION(11:31) CHAR,
 PRICE POSITION(32:36) INTEGER EXTERNAL,
 QTY_ON_HAND POSITION(37:39) INTEGER EXTERNAL
)
```

Помните [прошлый шаг](#)? Все правила соблюдены для загрузки данных! А ключевое слово **POSITION** читает данные согласно заданных значений! Далее там же создаем файл **PRODUCTS.dat** с вот таким содержимым:

```
REI2A45C Бочка металлическая 79 210
ACI4100Y Коробка картонная 2750 25
QSAXK47 Труба алюминиевая 355 38
BIC41672 Тарелка фарфоровая 180 0
IMM779C Профиль специальный 1875 9
ACI41003 Рейка деревянная 107 207
ACI41004 Рейка пластмассовая 117 139
BIC41003 Стекломасс рулоны 652 3
IMM887P Рубероид рулоны 250 24
QSAXK48 Гвоздь длинный 134 203
REI2A44L Доска профильная 4500 12
FEA112 Стол офисный 148 115
IMM887H Тумбочка прикроватная54 223
BIC41089 Сапоги юфтевые 225 78
ACI41001 Лампа настольная 55 277
IMM775C Осветитель ртутный 1425 5
ACI4100Z Монитор LG 2500 28
QSAXK48A Подушка ватная 177 37
ACI41002 Носки черные 76 167
REI2A44R Телевизор SAMSUNG 4500 12
IMM773C Наушники SONY 975 28
ACI4100X Карандаш простой 25 37
FEA114 Электродвигатель 243 15
IMM887X Нож специальный 475 32
REI2A44G Бочка пластмассовая 350 14
```

Здесь имеются позиции загрузки 01:03, 04:10, 11:31, 32:36, 37:39, если что-то не верно, то можете проверить сами. Это просто позиции знаков в строке отсюда и название загрузка данных, фиксированной длины. Итак, наши файлы готовы, таблица пустая можно приступать к загрузке! Запустите **PRODUCTS.bat** на исполнение и получите следующее сообщение в командной строке:

```
C:\Oracle\ora81\bin\LOAD>set nls_lang=russian_cis.ru8pc866
```

```
C:\Oracle\ora81\bin\LOAD>sqlldr.exe userid=miller/kolobok@proba control=PRODUCTS.ctl
errors=100
bad=PRODUCTS.bad
```

```
SQL*Loader: Release 8.1.5.0 - Production on Вск Май 16 14:13:34 1004
```

```
Copyright (c) 1881, 1001, Oracle Corporation. All rights reserved.
```

```
Достигнута точка фиксации - счетчик логич. записей 27
```

А, вот теперь заглянем в табличку **PROCDUCTS**, выполнив такой простой запрос к таблице **PRODUCTS**:

```
SELECT * FROM PRODUCTS
/
```

Получаем:

```
SQL> SELECT * FROM PRODUCTS
2 /
```

| MFR_ID | PRODUCT_ID | DESCRIPTION           | PRICE | QTY_ON_HAND |
|--------|------------|-----------------------|-------|-------------|
| REI    | 2A45C      | Бочка металлическая   | 79    | 210         |
| ACI    | 4100Y      | Коробка картонная     | 2750  | 25          |
| QSA    | XK47       | Труба алюминиевая     | 355   | 38          |
| BIC    | 41672      | Тарелка фарфоровая    | 180   | 0           |
| IMM    | 779C       | Профиль специальный   | 1875  | 9           |
| ACI    | 41003      | Рейка деревянная      | 107   | 207         |
| ACI    | 41004      | Рейка пластмассовая   | 117   | 139         |
| BIC    | 41003      | Стекломасс рулоны     | 652   | 3           |
| IMM    | 887P       | Рубероид рулоны       | 250   | 24          |
| QSA    | XK48       | Гвоздь длинный        | 134   | 203         |
| REI    | 2A44L      | Доска профильная      | 4500  | 12          |
| FEA    | 112        | Стол офисный          | 148   | 115         |
| IMM    | 887H       | Тумбочка прикроватная | 54    | 223         |
| BIC    | 41089      | Сапоги юфтевые        | 225   | 78          |
| ACI    | 41001      | Лампа настольная      | 55    | 277         |
| IMM    | 775C       | Осветитель ртутный    | 1425  | 5           |
| ACI    | 4100Z      | Монитор LG            | 2500  | 28          |
| QSA    | XK48A      | Подушка ватная        | 177   | 37          |
| ACI    | 41002      | Носки черные          | 76    | 167         |
| REI    | 2A44R      | Телевизор SAMSUNG     | 4500  | 12          |
| IMM    | 773C       | Наушники SONY         | 975   | 28          |
| ACI    | 4100X      | Карандаш простой      | 25    | 37          |
| FEA    | 114        | Электродвигатель      | 243   | 15          |
| IMM    | 887X       | Нож специальный       | 475   | 32          |
| REI    | 2A44G      | Бочка пластмассовая   | 350   | 14          |

25 строк выбрано.

Сразу замечу, если вместо русских букв у вас будут вражеские кабалистические символы, нужно все проделать снова, но файл **PRODUCTS.dat** открыть при помощи **Microsoft Word** и сохранить его в формате **Text DOS(!)** И тогда все получится! Не знаю как у вас, а у меня этот фокус прокатывал обычно! И последнее взглянем на содержимое файла журнала **PRODUCTS.log**:

SQL\*Loader: Release 8.1.5.0.0 - Production on Вск Май 16 14:23:34 2004

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Управляющий файл: PRODUCTS.ctl  
 Файл данных: PRODUCTS.DAT  
 Файл плохих записей: PRODUCTS.bad  
 Файл удаленных записей: ничего не задано

(Разрешить удалять все записи)

Количество записей для загрузки: ALL

Количество записей для пропуска: 0  
Допускается ошибок: 100  
Массив привязки: 64 строк, макс. из 256000 байт  
Продолжение: ничего не задано  
Использован маршрут: Условный

Таблица PRODUCTS, загружен из каждой логической записи.  
Режим вставки действует для этой таблицы: INSERT  
Действует опция TRAILING NULLCOLS

| Имя столбца | Позиция | Дл. | Огр. | Вкл | Тип данных     |
|-------------|---------|-----|------|-----|----------------|
| MFR_ID      | 1:3     | 3   |      |     | O(") CHARACTER |
| PRODUCT_ID  | 4:10    | 7   |      |     | O(") CHARACTER |
| DESCRIPTION | 11:31   | 21  |      |     | O(") CHARACTER |
| PRICE       | 32:36   | 5   |      |     | O(") CHARACTER |
| QTY_ON_HAND | 37:39   | 3   |      |     | O(") CHARACTER |

Запись 26: Удалена - все столбцы пустые.  
Запись 27: Удалена - все столбцы пустые.

Таблица PRODUCTS:  
25 Строки успешно загружено.  
0 Строки не загружены из-за ошибки в данных.  
0 Строки не загружены из-за сбоев во всех фразах WHEN.  
2 Строки не загружены из-за того, что все поля были пусты.

Для массива привязки отведено: 3456 байт(64 строк)  
Байтов буфера чтения: 1048576

Всего пропущено логических записей: 0  
Всего прочитано логических записей: 27  
Всего забраковано логических записей: 0  
Всего удалено логических записей: 2

Прогон начался в Вск Май 16 14:23:34 2004  
Прогон кончился в Вск Май 16 14:23:36 2004

Общее время: 00:00:02.31  
Процессорное время: 00:00:00.01

Думаю, особых комментариев не нужно, здесь хорошо отражена вся сессия, загрузки данных и все что происходило при загрузке. В том числе и позиции загрузки данных в таблицу. Вот таким образом производится загрузка данных в БД **Oracle** фиксированной длины! Если, что кому не понятно спрашивайте! Или разбирайтесь сами! :)

## Шаг 138 - БД Oracle - SQL\*Loader Способы загрузки - ЧАСТЬ II

С фиксированной длиной все ясно, но что я не припомню, чтобы мне где-то попадались, такие данные, да и вам такое может понадобиться редко! Поверь те мне, если только в особых случаях. Как правило, я всегда работал и работаю с данными переменной длины! И вот как. Очень неплохим и полезным средством для загрузки разнообразных данных в **Oracle**, как это ни странно, является **MS Access**! Да, вы будете смеяться, но это так! В нем есть очень забавная форматовка экспорта данных, в текстовом формате с разделителями! Вот этот механизм, мне очень помогает! Я сгружаю в **MS Access** разнообразные **xls**, **dbf**, **mdb** и т.д. А, на выходе получаю стройные ряды текстовых файлов для загрузки в **Oracle**! Все просто замечательно! Итак, что-то я разболтался! Давайте снова зальем данные в таблицу **PRODUCTS** для чего снова ее вычистим:

```
DELETE FROM PRODUCTS
/

COMMIT
/
```

Получаем:

```
SQL> DELETE FROM PRODUCTS
2 /

25 строк удалено.

SQL> COMMIT
2 /
```

Фиксация обновлений завершена.

Сечас можете взять файлы из [прошлого шага](#) или сделать новые, как вам больше нравится, но я все пройду до конца, чтобы вы не путались! Файл **PRODUCTS.bat** можно оставить без изменений, а вот два других будем переделывать! Для начала контрольный файл переписываем вот так:

```
LOAD DATA

INFILE 'PRODUCTS.dat'

INTO TABLE PRODUCTS

FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS

(MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND)
```

Здесь **FIELDS TERMINATED BY ';'** означает, что границы данных определены по символу ";", а **OPTIONALLY ENCLOSED BY '"'** определяет, что данные могут содержать символ обрамления. **TRAILING NULLCOLS** говорит о том, что если поле для загрузки не имеет данных в файле данных, то в поле записать **NULL**! Далее идет просто описание полей и все! А, вот **PRODUCTS.dat** будет иметь следующее содержимое:

```
REI;2A45C;Бочка металлическая;79;210
```

ACI;4100Y;Коробка картонная;2,750;25  
QSA;XK47;Труба алюминиевая;355;38  
BIC;41672;Тарелка фарфоровая;180;0  
IMM;779C;Профиль специальный;1875;9  
ACI;41003;Рейка деревянная;107;207  
ACI;41004;Рейка пластмассовая;117;139  
BIC;41003;Стекломаст рулоны;652;3  
IMM;887P;Рубероид рулоны;250;24  
QSA;XK48;Гвозд длинный;134;203  
REI;2A44L;Доска профильная;4500;12  
FEA;112;Стол офисный;148;115  
IMM;887H;Тумбочка прикроватная;54;223  
BIC;41089;Сапоги юфтиевые;225;78  
ACI;41001;Лампа настольная;55;277  
IMM;775C;Осветитель ртутный;1425;5  
ACI;4100Z;Монитор LG;2500;28  
QSA;XK48A;Подушка ватная;177;37  
ACI;41002;Носки черные;76;167  
REI;2A44R;Телевизор SAMSUNG;4500;12  
IMM;773C;Наушники SONY;975;28  
ACI;4100X;Карандаш простой;25;37  
FEA;114;Электродвигатель;243;15  
IMM;887X;Нож специальный;475;32  
REI;2A44G;Бочка пластмассовая;350;14

Видите данные отделены символом ";" друг от друга и все! Естественно, откуда я знаю все эти позиции, так гораздо проще и быстрее! А, если записей скажем миллион или десять! А? :) Запускаем файл загрузки на исполнение и получаем следующее:

```
C:\Oracle\ora81\bin\LOAD>set nls_lang=russian_cis.ru8pc866
```

```
C:\Oracle\ora81\bin\LOAD>sqlldr.exe userid=miller/kolobok@proba control=PRODUCTS.ctl
errors=100
bad=PRODUCTS.bad
```

```
SQL*Loader: Release 8.1.5.0 - Production on Вск Май 16 14:13:34 1004
```

```
Copyright (c) 1881, 1001, Oracle Corporation. All rights reserved.
```

```
Достигнута точка фиксации - счетчик логич. записей 27
```

Ничего нового вроде нет! И содержимое таблицы тоже самое, что и в [первом случае](#). Смотрим:

```
SELECT * FROM PRODUCTS
/
```

Получаем:

```
SQL> SELECT * FROM PRODUCTS
2 /
```

| MFR_ID | PRODUCT_ID | DESCRIPTION | PRICE | QTY_ON_HAND |
|--------|------------|-------------|-------|-------------|
|--------|------------|-------------|-------|-------------|

```

REI 2A45C Бочка металлическая 79 210
ACI 4100Y Коробка картонная 2750 25
QSA XK47 Труба алюминиевая 355 38
BIC 41672 Тарелка фарфоровая 180 0
IMM 779C Профиль специальный 1875 9
ACI 41003 Рейка деревянная 107 207
ACI 41004 Рейка пластмассовая 117 139
BIC 41003 Стекломасс рулоны 652 3
IMM 887P Рубероид рулоны 250 24
QSA XK48 Гвоздь длинный 134 203
REI 2A44L Доска профильная 4500 12
FEA 112 Стол офисный 148 115
IMM 887H Тумбочка прикроватная 54 223
BIC 41089 Сапоги юфтевые 225 78
ACI 41001 Лампа настольная 55 277
IMM 775C Осветитель ртутный 1425 5
ACI 4100Z Монитор LG 2500 28
QSA XK48A Подушка ватная 177 37
ACI 41002 Носки черные 76 167
REI 2A44R Телевизор SAMSUNG 4500 12
IMM 773C Наушники SONY 975 28
ACI 4100X Карандаш простой 25 37
FEA 114 Электродвигатель 243 15
IMM 887X Нож специальный 475 32
REI 2A44G Бочка пластмассовая 350 14

```

25 строк выбрано.

А, вот содержимое файла журнала уже другое, как и следовало ожидать:

SQL\*Loader: Release 8.1.5.0.0 - Production on Вск Май 16 15:36:03 2004

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Управляющий файл: PROD.ctl  
 Файл данных: PROD.dat  
 Файл плохих записей: PROD.bad  
 Файл удаленных записей: PROD.dis  
 (Разрешить удалять все записи)

Количество записей для загрузки: ALL  
 Количество записей для пропуска: 0  
 Допускается ошибок: 100  
 Массив привязки: 64 строк, макс. из 256000 байт  
 Продолжение: ничего не задано  
 Использован маршрут: Условный

Таблица PRODUCTS, загружен из каждой логической записи.  
 Режим вставки действует для этой таблицы: INSERT  
 Действует опция TRAILING NULLCOLS

| Имя столбца | Позиция | Дл. | Огр. | Вкл | Тип данных     |
|-------------|---------|-----|------|-----|----------------|
| MFR_ID      | FIRST   | *   |      |     | O(") CHARACTER |
| PRODUCT_ID  | NEXT    | *   |      |     | O(") CHARACTER |
| DESCRIPTION | NEXT    | *   |      |     | O(") CHARACTER |
| PRICE       | NEXT    | *   |      |     | O(") CHARACTER |
| QTY_ON_HAND | NEXT    | *   |      |     | O(") CHARACTER |

Таблица PRODUCTS:

25 Строки успешно загружено.

0 Строки не загружены из-за ошибки в данных.

0 Строки не загружены из-за сбоев во всех фразах WHEN.

0 Строки не загружены из-за того, что все поля были пусты.

Для массива привязки отведено: 82560 байт(64 строк)  
Байтов буфера чтения: 1048576

Всего пропущено логических записей: 0  
Всего прочитано логических записей: 25  
Всего забраковано логических записей: 0  
Всего удалено логических записей: 0

Прогон начался в Вск Май 16 15:36:03 2004  
Прогон кончился в Вск Май 16 15:36:05 2004

Общее время: 00:00:02.31  
Процессорное время: 00:00:00.01

Обратите внимание, что поля **PRICE**, **QTY\_ON\_HAND** обозначены как **CHARACTER**, а в реальной таблице они имеют другой тип! Именно здесь и сработало неявное преобразование. Вот так грузятся данные переменной длины. Еще один способ, загрузки носит название - загрузка вложенных данных. Давайте рассмотрим его. Итак, для начала очистим нашу учебную табличку:

```
DELETE FROM PRODUCTS
/
```

```
COMMIT
/
```

Получаем:

```
SQL> DELETE FROM PRODUCTS
2 /
```

25 строк удалено.

```
SQL> COMMIT
2 /
```

Фиксация обновлений завершена.



Теперь слейте вместе файлы **PRODUCTS.ctl** и **PRODUCTS.dat**, например, вот так:

```
C:\copy PRODUCTS.dat+ PRODUCTS.ctl PRODUCTS.txt
```

Затем смените расширение файла **PRODUCTS.txt** на **PRODUCTS.ctl** и отредактируйте его содержимое вот так:

```
LOAD DATA
```

```
INFILE *
```

```
INTO TABLE PRODUCTS
```

```
FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS
```

```
(MFR_ID, PRODUCT_ID, DESCRIPTION, PRICE, QTY_ON_HAND)
```

```
BEGINDATA
```

```
REI;2A45C;Бочка металлическая;79;210
ACI;4100Y;Коробка картонная;2,750;25
QSA;XK47;Труба алюминиевая;355;38
BIC;41672;Тарелка фарфоровая;180;0
IMM;779C;Профиль специальный;1875;9
ACI;41003;Рейка деревянная;107;207
ACI;41004;Рейка пластмассовая;117;139
BIC;41003;Стекломаст рулоны;652;3
IMM;887P;Рубероид рулоны;250;24
QSA;XK48;Гвозд длинный;134;203
REI;2A44L;Доска профильная;4500;12
FEA;112;Стол офисный;148;115
IMM;887H;Тумбочка прикроватная;54;223
BIC;41089;Сапоги юфтиевые;225;78
ACI;41001;Лампа настольная;55;277
IMM;775C;Осветитель ртутный;1425;5
ACI;4100Z;Монитор LG;2500;28
QSA;XK48A;Подушка ватная;177;37
ACI;41002;Носки черные;76;167
REI;2A44R;Телевизор SAMSUNG;4500;12
IMM;773C;Наушники SONY;975;28
ACI;4100X;Карандаш простой;25;37
FEA;114;Электродвигатель;243;15
IMM;887X;Нож специальный;475;32
REI;2A44G;Бочка пластмассовая;350;14
```

Здесь слово **BEGINDATA** означает начала блока загружаемых данных. В данном случае контрольный файл выполняет двойную функцию и описывает правила загрузки и несет данные для нее! Если вам такой способ кажется более удачным, можете его использовать, если нет, то это как вам удобнее. Обратите так же внимание, что после ключевого слова **INFILE** стоит просто знак "\*". Содержимое журнала после такой загрузки будет такое:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Вск Май 16 15:58:22 2004
```

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Управляющий файл: PROD.ctl  
Файл данных: PROD.ctl  
Файл плохих записей: PROD.bad  
Файл удаленных записей: PROD.dis  
(Разрешить удалять все записи)

Количество записей для загрузки: ALL  
Количество записей для пропуска: 0

.  
.  
.  
PRICE NEXT \* ; O(") CHARACTER  
QTY\_ON\_HAND NEXT \* ; O(") CHARACTER

.  
.  
.

Общее время: 00:00:02.31  
Процессорное время: 00:00:00.01

Отличие только в записи файла данных, все остальное, так же как и в [прошлый раз](#). Для примера можете попробовать загрузку с опциями загрузки для таблиц, например **TRUNCATE!** Удачи! :)

## Шаг 139 - БД Oracle - SQL\*Loader Способы загрузки - ЧАСТЬ III

Продолжаем загрузку данных в БД **Oracle**. На первый взгляд, вроде уже достаточно того, что было изложено! Но есть еще несколько моментов, которые вы будете применять, может быть не часто, но это значительно облегчит вам жизнь в дальнейшем! Итак, рассмотрим загрузку по условию. Допустим, что есть данные, которые подготовлены к загрузке, но не все они нам необходимы в данное время. Можно было конечно просто загрузить их в таблицу и применив знание языка **DML** убрать лишнее, но можно поступить еще проще - применив условия загрузки данных. Для начала создадим табличку дубликат **ORDERS** и назовем ее **ORDERSTWO**:

```
CREATE TABLE MILLER.ORDERSTWO
(
 ORDER_NUM INTEGER PRIMARY KEY,
 ORDER_DATE DATE,
 CUST INTEGER,
 REP INTEGER,
 MFR VARCHAR2(3),
 PRODUCT VARCHAR2(5),
 QTY INTEGER,
 AMOUNT NUMBER
)
```

Получаем:

```
SQL> CREATE TABLE MILLER.ORDERSTWO
2 (
3 ORDER_NUM INTEGER PRIMARY KEY,
4 ORDER_DATE DATE,
5 CUST INTEGER,
6 REP INTEGER,
7 MFR VARCHAR2(3),
8 PRODUCT VARCHAR2(5),
9 QTY INTEGER,
10 AMOUNT NUMBER
11)
12 /
```

Таблица создана.

Теперь создайте файл **ORDTWO.DAT** с вот таким содержимым:

```
112961;17-12-1989;2117;106;REI;2A44L;7;31,500
113012;11-06-1990;2111;105;ACI;41003;35;3,745
112989;03-06-1990;2101;106;FEA;114;6;1,458
113051;10-02-1990;2118;108;QSA;XK47;4;1,420
112968;12-10-1989;2102;101;ACI;41004;34;3,978
113036;30-06-1990;2107;110;ACI;4100Z;9;22,500
113045;02-02-1990;2112;108;REI;2A44R;10;45,000
112963;17-12-1989;2103;105;ACI;41004;28;3,276
113013;14-06-1990;2118;108;BIC;41003;1;652,000
```

```
113058;23-02-1990;2108;109;FEA;112;10;1,478
112997;08-06-1990;2124;107;BIC;41003;1;652,000
112983;27-12-1989;2103;105;ACI;41004;6;702,000
113024;20-06-1990;2114;108;QSA;XK47;20;7,100
113062;24-02-1990;2124;107;FEA;114;10;2,430
112979;12-10-1989;2114;102;ACI;4100Z;6;15,000
113027;22-06-1990;2103;105;ACI;41002;54;4,104
113007;08-06-1990;2112;108;IMM;773C;3;2,925
113069;02-03-1990;2109;107;IMM;775C;22;31,350
113034;29-06-1990;2107;110;REI;2A45C;8;632,00
112922;04-11-1989;2118;108;ACI;41002;10;760,000
112975;12-10-1989;2111;103;REI;2A44G;6;2,100
113055;15-02-1990;2108;101;ACI;4100X;6;150,000
113048;10-02-1990;2120;102;IMM;779C;2;3,750
112993;04-06-1989;2106;102;REI;2A45C;24;1,896
113065;27-02-1989;2106;102;QSA;XK47;6;2,130
113003;25-06-1990;2108;109;IMM;779C;3;5,625
113049;10-02-1990;2118;108;QSA;XK47;2;710,000
112987;31-12-1989;2103;105;ACI;4100Y;11;27,500
113057;18-02-1990;2111;103;ACI;4100X;24;600,34
113042;02-02-1990;2113;101;REI;2A44R;5;22,500
```

Самое главное проверьте, чтобы в конце файла данных для загрузки не стояло лишних символов - пробелов, табуляций и возврата каретки. Их наличие в конце файла может вызывать излишние ошибки, которые могут ввести вас в заблуждение! Далее создайте контрольный файл **ORDTWO.ctf** собственно загрузчика примерно вот такого содержания:

```
LOAD DATA
```

```
INFILE 'ORDTWO.DAT'
```

```
INTO TABLE ORDERSTWO
WHEN MFR = 'ACI'
```

```
FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS
```

```
(ORDER_NUM, ORDER_DATE, CUST, REP, MFR, PRODUCT, QTY, AMOUNT ":AMOUNT * 2")
```

Здесь задано условие **WHEN MFR = 'ACI'**, которое определит загрузку только данных с идентификатором **ACI**, все остальное будет отброшено! Так же я хочу продемонстрировать возможность использования хост переменных, которые во время загрузки данных позволяют изменять содержимое полей. Например, выражение **AMOUNT ":AMOUNT \* 2"** увеличивает значение в столбце **AMOUNT** в двое! Ну и последнее создадим, исполняемый файл для загрузки данных с именем **ORDTWO.bat**:

```
..\->ORDTWO.bat
```

```
@echo off
```

```
set nls_lang=russian_cis.ru8pc866
```

---

```
sqlldr.exe userid=miller/kolobok control=ORDTWO.ctl errors=100 bad=ORDTWO.bad
discard=ORDTWO.dis
```

После запуска получаем сообщение об успешной загрузке данных:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Сбт Май 29 18:14:29 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Достигнута точка фиксации - счетчик логич. записей 32
```

Обратите внимание, что количество записей 32! Хотя в таблицу были загружены не все записи, а только 11 строк, что соответствует идентификатору **ACI**. Вот часть содержимого **log** файла после загрузки данных:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Вск Май 30 12:28:56 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Управляющий файл: ORDTWO.ctl
Файл данных: ORDTWO.DAT
Файл плохих записей: ORDTWO.bad
Файл удаленных записей: ORDTWO.dis
```

```
.
.
```

```
Действует опция TRAILING NULLCOLS
```

| Имя столбца                            | Позиция | Дл. | Огр. | Вкл  | Тип данных |
|----------------------------------------|---------|-----|------|------|------------|
| ORDER_NUM                              | FIRST   | *   | ;    | O(") | CHARACTER  |
| ORDER_DATE                             | NEXT    | *   | ;    | O(") | CHARACTER  |
| CUST                                   | NEXT    | *   | ;    | O(") | CHARACTER  |
| REP                                    | NEXT    | *   | ;    | O(") | CHARACTER  |
| MFR                                    | NEXT    | *   | ;    | O(") | CHARACTER  |
| PRODUCT                                | NEXT    | *   | ;    | O(") | CHARACTER  |
| QTY                                    | NEXT    | *   | ;    | O(") | CHARACTER  |
| AMOUNT                                 | NEXT    | *   | ;    | O(") | CHARACTER  |
| Строка SQL для столбца : ":AMOUNT * 2" |         |     |      |      |            |

```
Запись 1: Удалена - сбой во всех фразах WHEN.
```

```
Запись 3: Удалена - сбой во всех фразах WHEN.
```

```
Запись 4: Удалена - сбой во всех фразах WHEN.
```

```
Запись 7: Удалена - сбой во всех фразах WHEN.
```

```
.
.
```

```
Запись 30: Удалена - сбой во всех фразах WHEN.
```

```
Запись 31: Удалена - сбой во всех фразах WHEN.
```

```
Запись 32: Удалена - сбой во всех фразах WHEN.
```

Таблица ORDERSTWO:

```

11 Строки успешно загружено.
0 Строки не загружены из-за ошибки в данных.
21 Строки не загружены из-за сбоев во всех фразах WHEN.
0 Строки не загружены из-за того, что все поля были пусты.

```

```

.
.
.
Всего забраковано логических записей: 0
Всего удалено логических записей: 21
.
.
Процессорное время: 00:00:00.04

```

Как видим, строки **"Запись 1: Удалена - сбой во всех фразах WHEN."** как раз и показывают, что записи не соответствующие заданному условию были отброшены! Вот таким образом можно использовать условия при загрузке данных! Например, можете изменить содержимое контрольного файла на ниже следующее и догрузить еще часть данных вот так:

```

LOAD DATA

INFILE 'ORDTWO.DAT'

APPEND

INTO TABLE ORDERSTWO
WHEN MFR = 'REI'

FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"' TRAILING NULLCOLS

(ORDER_NUM, ORDER_DATE, CUST, REP, MFR, PRODUCT, QTY, AMOUNT ":AMOUNT * 2")

```

Запустите еще раз **bat** файл загрузчика и записей в таблице станет 17. Посмотрим, что получилось дав запрос:

```

SELECT * FROM ORDERSTWO
/

```

Получим:

```

SQL> SELECT * FROM ORDERSTWO
2 /

```

| ORDER_NUM | ORDER_DATE | CUST | REP | MFR | PRODUCT | QTY | AMOUNT |
|-----------|------------|------|-----|-----|---------|-----|--------|
| 113012    | 11.06.1990 | 2111 | 105 | ACI | 41003   | 35  | 7,49   |
| 112968    | 12.10.1989 | 2102 | 101 | ACI | 41004   | 34  | 7,956  |
| 113036    | 30.06.1990 | 2107 | 110 | ACI | 4100Z   | 9   | 45     |
| .         |            |      |     |     |         |     |        |
| .         |            |      |     |     |         |     |        |
| .         |            |      |     |     |         |     |        |
| 112975    | 12.10.1989 | 2111 | 103 | REI | 2A44G   | 6   | 4,2    |

|                   |      |               |    |       |
|-------------------|------|---------------|----|-------|
| 112993 04.06.1989 | 2106 | 102 REI 2A45C | 24 | 3,792 |
| 113042 02.02.1990 | 2113 | 101 REI 2A44R | 5  | 45    |

17 строк выбрано.

Что собственно и следовало ожидать! Вот таким образом вы можете использовать условия при загрузке данных, что может быть особенно полезно когда загружается большое количество данных или таблицы секционированы. Пробуйте! :)

## Шаг 140 - БД Oracle - SQL\*Loader Загрузка данных - ЧАСТЬ IV

Рассмотрим еще один интересный аспект при загрузке данных. Это загрузка в разделы таблицы. Для начала скажу, что таблица в которой применяется секционирование, имеет несколько разделов данных, которые могут быть расположены в различных табличных пространствах. Мы с вами такие таблицы еще не рассматривали. До них мы доберемся чуть позже, а пока может быть кому-нибудь пригодиться, то как это выполняется! Рассмотрим пример, который я нашел в одной очень интересной книге и практически без изменений переложил для нашего с вами рассмотрения. Итак, создадим таблицу с разделами, но в одном табличном пространстве **USERS** с именем **partition\_xact**:

```
create table MILLER.partition_xact(
 acct_nbr number not null,
 xact_amt number not null,
 xact_flag char not null,
 xact_date date not null)
PARTITION BY RANGE (xact_date)
(PARTITION P1 VALUES LESS THAN (to_date('01-04-1999','DD-MM-YYYY')),
 PARTITION P2 VALUES LESS THAN (to_date('01-07-1999','DD-MM-YYYY')),
 PARTITION P3 VALUES LESS THAN (to_date('01-11-1999','DD-MM-YYYY')),
 PARTITION P4 VALUES LESS THAN (MAXVALUE))
/
```

Получаем:

```
SQL> create table MILLER.partition_xact(
 2 acct_nbr number not null,
 3 xact_amt number not null,
 4 xact_flag char not null,
 5 xact_date date not null)
 6 PARTITION BY RANGE (xact_date)
 7 (PARTITION P1 VALUES LESS THAN (to_date('01-04-1999','DD-MM-YYYY')),
 8 PARTITION P2 VALUES LESS THAN (to_date('01-07-1999','DD-MM-YYYY')),
 9 PARTITION P3 VALUES LESS THAN (to_date('01-11-1999','DD-MM-YYYY')),
10 PARTITION P4 VALUES LESS THAN (MAXVALUE))
11 /
```

Таблица создана.

Мы с вами получили таблицу с четырьмя разделами по полю **xact\_date**. В данном случае это и есть секционированная таблица. Далее создадим контрольный файл с именем **PART.CTL** следующего содержания:

LOAD DATA

```
INFILE 'xact.dat'
INTO TABLE partition_xact PARTITION (P1)
WHEN xact_flag = 'D'
```

```
(acct_nbr POSITION(01:10) INTEGER EXTERNAL,
 xact_amt POSITION(11:20) INTEGER EXTERNAL ":xact_amt * -1",
```



```
xact_flag POSITION(21:21) CHAR,
xact_date POSITION(22:31) DATE "DD-MM-YY" NULLIF xact_date=BLANKS)

INTO TABLE partition_xact PARTITION (P1)
WHEN xact_flag = 'C'
(acct_nbr POSITION(01:10) INTEGER EXTERNAL,
xact_amt POSITION(11:20) INTEGER EXTERNAL,
xact_flag POSITION(21:21) CHAR,
xact_date POSITION(22:31) DATE "DD-MM-YY" NULLIF xact_date=BLANKS)
```

Здесь применяется та же загрузка по условию с данными переменной длины, еще и в два захода! Для начала, мы пробуем произвести загрузку в раздел **P1**. Далее создадим файл с исходными данными для загрузки с именем **xact.dat** и следующим содержимым:

```
0000459023 123D01-02-98
0000459023 1233C01-03-99
0000459023 987P01-13-98
0000459024 1000C03-06-99
0000211108 875D23-07-98
0000211123 20987C30-12-99
0000211123 12500D10-01-98
0000023388 1C19-05-99
0000043992 350C12-03-97
0050699390 2899D01-09-97
0000023330 100D26-06-97
0000433020 60C20-11-97
0000004566 230C20-08-97
0000004599 14D05-06-97
0000004599 14D07-06-97
0008887544 9999D11-7-
```

Далее и последнее создадим исполняемый файл для загрузки данных с именем **PART.bat** и следующим содержимым:

```
@echo off

set nls_lang=russian_cis.ru8pc866

sqlldr.exe userid=miller/kolobok control=PART.ctl errors=100 bad=PART.bad discard=PART.dis
```

Итак, все готово, запустим файл **PART.bat** на исполнение и посмотрим, что получится после его работы! Уу-пс! Возник **PART.bad**, в котором все записи отправленные нами на загрузку! А вот и **log** файл:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Вск Май 30 13:04:52 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Управляющий файл: PART.ctl
Файл данных: xact.dat
Файл плохих записей: PART.bad
Файл удаленных записей: PART.dis
```

(Разрешить удалять все записи)

Количество записей для загрузки: ALL  
 Количество записей для пропуска: 0  
 Допускается ошибок: 100  
 Массив привязки: 64 строк, макс. из 256000 байт  
 Продолжение: ничего не задано  
 Использован маршрут: Условный

Таблица PARTITION\_XACT, раздел P1, загружен когда XACT\_FLAG = 0X44(символ 'D')  
 Для раздела INSERT включен параметр Insert

| Имя столбца                               | Позиция | Дл. | Огр. | Вкл | Тип данных    |
|-------------------------------------------|---------|-----|------|-----|---------------|
| ACCT_NBR                                  | 1:10    | 10  |      |     | CHARACTER     |
| XACT_AMT                                  | 11:20   | 10  |      |     | CHARACTER     |
| Строка SQL для столбца : ":xact_amt * -1" |         |     |      |     |               |
| XACT_FLAG                                 | 21:21   | 1   |      |     | CHARACTER     |
| XACT_DATE                                 | 22:31   | 10  |      |     | DATE DD-MM-YY |
| NULL, если XACT_DATE = BLANKS             |         |     |      |     |               |

Таблица PARTITION\_XACT, раздел P1, загружен когда XACT\_FLAG = 0X43(символ 'C')  
 Для раздела INSERT включен параметр Insert

| Имя столбца                   | Позиция | Дл. | Огр. | Вкл | Тип данных    |
|-------------------------------|---------|-----|------|-----|---------------|
| ACCT_NBR                      | 1:10    | 10  |      |     | CHARACTER     |
| XACT_AMT                      | 11:20   | 10  |      |     | CHARACTER     |
| XACT_FLAG                     | 21:21   | 1   |      |     | CHARACTER     |
| XACT_DATE                     | 22:31   | 10  |      |     | DATE DD-MM-YY |
| NULL, если XACT_DATE = BLANKS |         |     |      |     |               |

Запись 3: Удалена - сбой во всех фразах WHEN.

Запись 1: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 5: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 7: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 10: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 11: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 14: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 15: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
 ORA-14401: ключ вставляемой секции вне указанной секции

Запись 16: Забракована - Ошибка в таблице PARTITION\_XACT, столбце XACT\_DATE.  
ORA-01840: вводимое значение недостаточно длинное для формата даты

Запись 2: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 4: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 6: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 8: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 9: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 12: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Запись 13: отвергнута - Ошибка в таблице PARTITION\_XACT, раздела P1.  
ORA-14401: ключ вставляемой секции вне указанной секции

Таблица PARTITION\_XACT, раздел P1:

0 Строки успешно загружено.

8 Строки не загружены из-за ошибки в данных.

8 Строки не загружены из-за сбоев во всех фразах WHEN.

0 Строки не загружены из-за того, что все поля были пусты.

Таблица PARTITION\_XACT, раздел P1:

0 Строки успешно загружено.

7 Строки не загружены из-за ошибки в данных.

9 Строки не загружены из-за сбоев во всех фразах WHEN.

0 Строки не загружены из-за того, что все поля были пусты.

Для массива привязки отведено: 5120 байт(64 строк)

Байтов буфера чтения: 1048576

Всего пропущено логических записей: 0

Всего прочитано логических записей: 16

Всего забраковано логических записей: 15

Всего удалено логических записей: 1

Прогон начался в Вск Май 30 13:04:52 2004

Прогон кончился в Вск Май 30 13:04:54 2004

Общее время: 00:00:02.47

Процессорное время: 00:00:00.05

Как видим, все записи отвергнуты, так как "ключ вставляемой секции вне указанной секции"! Что-ж в данном случае это соответствует действительности, изменим запись в контролфайле вот так:

## LOAD DATA

INFILE 'xact.dat'

INTO TABLE partition\_xact PARTITION (P4)

WHEN xact\_flag = 'D'

```
(acct_nbr POSITION(01:10) INTEGER EXTERNAL,
 xact_amt POSITION(11:20) INTEGER EXTERNAL ":xact_amt * -1",
 xact_flag POSITION(21:21) CHAR,
 xact_date POSITION(22:31) DATE "DD-MM-YY" NULLIF xact_date=BLANKS)
```

INTO TABLE partition\_xact PARTITION (P4)

WHEN xact\_flag = 'C'

```
(acct_nbr POSITION(01:10) INTEGER EXTERNAL,
 xact_amt POSITION(11:20) INTEGER EXTERNAL,
 xact_flag POSITION(21:21) CHAR,
 xact_date POSITION(22:31) DATE "DD-MM-YY" NULLIF xact_date=BLANKS)
```

И снова запустим загрузку! После этого снова заглянем в **log** файл:

SQL\*Loader: Release 9.2.0.1.0 - Production on Вск Май 30 13:08:14 2004

Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.

Управляющий файл: PART.ctl

Файл данных: xact.dat

Файл плохих записей: PART.bad

Файл удаленных записей: PART.dis

(Разрешить удалять все записи)

Количество записей для загрузки: ALL

Количество записей для пропуска: 0

Допускается ошибок: 100

Массив привязки: 64 строк, макс. из 256000 байт

Продолжение: ничего не задано

Использован маршрут: Условный

Таблица PARTITION\_XACT, раздел P4, загружен когда XACT\_FLAG = 0X44(символ 'D')

Для раздела INSERT включен параметр Insert

| Имя столбца                               | Позиция | Дл. | Огр. | Вкл | Тип данных    |
|-------------------------------------------|---------|-----|------|-----|---------------|
| ACCT_NBR                                  | 1:10    | 10  |      |     | CHARACTER     |
| XACT_AMT                                  | 11:20   | 10  |      |     | CHARACTER     |
| Строка SQL для столбца : ":xact_amt * -1" |         |     |      |     |               |
| XACT_FLAG                                 | 21:21   | 1   |      |     | CHARACTER     |
| XACT_DATE                                 | 22:31   | 10  |      |     | DATE DD-MM-YY |
| NULL, если XACT_DATE = BLANKS             |         |     |      |     |               |

Таблица PARTITION\_XACT, раздел P4, загружен когда XACT\_FLAG = 0X43(символ 'C')

Для раздела INSERT включен параметр Insert

| Имя столбца                   | Позиция | Дл. | Огр. | Вкл | Тип данных    |
|-------------------------------|---------|-----|------|-----|---------------|
| ACCT_NBR                      | 1:10    | 10  |      |     | CHARACTER     |
| XACT_AMT                      | 11:20   | 10  |      |     | CHARACTER     |
| XACT_FLAG                     | 21:21   | 1   |      |     | CHARACTER     |
| XACT_DATE                     | 22:31   | 10  |      |     | DATE DD-MM-YY |
| NULL, если XACT_DATE = BLANKS |         |     |      |     |               |

Запись 3: Удалена - сбой во всех фразах WHEN.

Запись 16: Забракована - Ошибка в таблице PARTITION\_XACT, столбце XACT\_DATE.

ORA-01840: вводимое значение недостаточно длинное для формата даты

Таблица PARTITION\_XACT, раздел P4:

7 Строки успешно загружено.

1 Строка не загружены из-за ошибки в данных.

8 Строки не загружены из-за сбоев во всех фразах WHEN.

0 Строки не загружены из-за того, что все поля были пусты.

Таблица PARTITION\_XACT, раздел P4:

7 Строки успешно загружено.

0 Строки не загружены из-за ошибки в данных.

9 Строки не загружены из-за сбоев во всех фразах WHEN.

0 Строки не загружены из-за того, что все поля были пусты.

Для массива привязки отведено: 5120 байт(64 строк)  
Байтов буфера чтения: 1048576

Всего пропущено логических записей: 0  
Всего прочитано логических записей: 16  
Всего забраковано логических записей: 1  
Всего удалено логических записей: 1

Прогон начался в Вск Май 30 13:08:14 2004  
Прогон кончился в Вск Май 30 13:08:16 2004

Общее время: 00:00:02.34  
Процессорное время: 00:00:00.03

Теперь только две записи были отброшены, а все остальное ушло в таблицу **PARTITION\_XACT** и данные в ее разделе **P4** были успешно получены. Все это несколько мутновато, но может кому-нибудь будет полезно! Я в своей практике сталкивался раза два с такими таблицами, но сам я их не использовал, по этому можете сами придумать что-то подобное и произвести загрузку данных в такую табличку! Думаю у вас это должно получиться! :)

## Шаг 141 - БД Oracle - SQL\*Loader Теория и полезные примеры - ЧАСТЬ V

Для более полной картины работы **SQL\*Loader**, я думаю есть смысл остановиться на том, как данные вообще загружаются в БД **Oracle**. Существует два способа загрузки данных БД. Обычный (**conventional**) и прямой (**direct**). Для того, чтобы активировать прямой метод загрузки необходимо в командной строке указать параметр **DIRECT=TRUE**. При использовании обычного (**conventional**) типа загрузки данных, используется **SQL** оператор **INSERT** и буферы памяти. В этот момент может возникнуть конкуренция с другими процессами за выделение ресурсов памяти в **SGA**. Это может привести к замедлению работы загрузчика. Так же неудобство обычного метода может заключаться в том, что **SQL\*Loader** сканирует БД в поисках частично заполненных блоков загружаемой таблицы с целью их заполнения. Это может так же повлиять на загрузку сервера БД в целом. Но есть ряд случаев, при котором **conventional** метод более предпочтителен, либо вообще единственно возможный! А, именно в следующих случаях:

1. Если таблица, в которую направляются данные индексируются или к ней имеются обращения из других (помимо загрузки) процессов. А, так же когда происходит удаление или вставка записей в таблицу.
2. При использовании **SQL** операторов в контрольном файле.
3. Если загружаемая таблица является кластерной.
4. Если загружается небольшое число строк в большую индексируемую таблицу и/или, если для таблицы заданы ограничения ссылочной целостности или на значение данных.
5. Если работа происходит через **Net8** или **SQL\*Net**. при этом происходит использование различных платформ и оба узла принадлежат одному семейству компьютеров и использована одна и та же таблица кодировки символов.

Схематично по этапам обычная загрузка происходит следующим образом:

1. Разнесение элементов входных данных по столбцам базы данных.
2. Формирование **SQL** операторов **INSERT**.
3. Поиск конца секции табличных данных в памяти.
4. Работа с кэш-буфером.
5. Чтение и запись блоков базы данных.

При прямом методе загрузки данных, они формируются непосредственно в БД, без формирования операторов **INSERT**. При этом данные размещаются в памяти после существующих табличных данных. Этапы, которые имеются при прямом методе следующие:

1. Разнесение элементов входных данных, по столбцам базы данных.
2. Поиск концов секции табличных данных в памяти.
3. Форматирование блоков данных и запись их непосредственно в базу данных.

Вот таким образом происходит прямая загрузка. Как правило я почти не пользовался **direct** способом, так как вполне хватало обычного метода загрузки, но представлять как это выглядит не помешает. Для простоты картины, я пока пропускаю загрузку во вложенные таблицы и **VARRAY** массивы. Это отдельная тематика и мы ее еще коснемся! А, пока давайте рассмотрим ряд забавных примеров использования **SQL\*Loader** и его преимуществ! Я буду излагать все теоретически, а вы пробуйте практически, если будут ошибки пишите! Например, очень полезен вот такой примерчик.

Есть вот такой файл данных:

```
.dat
```

```
DEPOSIT 10015 10073
DEPOSIT 10020 7525
WITHDRAWAL 10015 2000
```

Создадим табличку:

```
CREATE TABLE register (
tx_type CHAR(15),
acct NUMBER,
amt NUMBER
)
/
```

Затем вот такой контрольный файл:

```
.ctl

LOAD DATA
INFILE 'month.dat'
INTO TABLE register
(tx_type POSITION(1:10),
acct POSITION(13:17),
amt POSITION(20:24) ":amt/100"
)
```

После загрузки получим такое содержимое таблички:

```
SELECT * FROM register;
```

| TX_TYPE    | ACCT  | AMT    |
|------------|-------|--------|
| DEPOSIT    | 10015 | 100.73 |
| DEPOSIT    | 10020 | 75.25  |
| WITHDRAWAL | 10015 | 20     |

Просто и полезно! Вот еще примерчик использования последовательности **Oracle** и предиката **USER**.

Создадим вот такую последовательность:

```
CREATE SEQUENCE db_seq
START WITH 1
INCREMENT BY 1
/
```

Создадим такую табличку:

```
CREATE TABLE load_db_seq
```

```
(
seq_number NUMBER,
username CHAR(30),
data1 NUMBER,
data2 CHAR(15)
)
/
```

И контрольный файл:

```
LOAD DATA
INFILE *
INTO TABLE load_db_seq_delimited
FIELDS TERMINATED BY "," TRAILING NULLCOLS
(seq_number "db_seq.nextval", username "USER", data1, data2)

BEGINDATA
11111,AAAAAAAAAA
22222,BBBBBBBBBB
33333,CCCCCCCCC
```

А вот содержимое после загрузки:

```
SELECT * FROM load_db_seq_delimited
/

SEQ_NUMBER USERNAME DATA1 DATA2

1 MILLER 11111 AAAAAAAAAA
2 MILLER 22222 BBBBBBBBBB
3 MILLER 33333 CCCCCCCCCC
```

Простенько и со вкусом, думаю будет полезно в дальнейшем для использования - если вспомнить сколько еще переменных в **Oracle** фантазии могут быть самые разные! Пробуйте! :)



## Шаг 142 - БД Oracle - SQL\*Loader Загрузка LOB объектов! ЧАСТЬ VI

Думаю самое время рассмотреть наиболее интересный метод загрузки данных, а именно загрузку больших объектов! Да, как ни странно, многие и не пробовали это делать! Хотя на самом деле это довольно просто! Например, у вас есть таблица содержащая поле **CLOB** и вам необходимо загрузить в нее скажем вот такой текст:

Here's my favorite story on recursion: it happened that a famous Darwinist was telling a story about primitive creation myths. "Some peoples," he said, "believe the world rests on the back of a great turtle. Of course, that raises the question: on what does the turtle rest?"  
An elderly woman from the back of the room stood up and said, "Very clever, Sonny, but it's turtles, all the way down."

Нет ничего проще, давайте сделаем это, причем несколькими способами. Для начала создадим таблицу, с именем **CLOBGET**:

```
CREATE TABLE CLOBGET
(
 id NUMBER,
 f_name VARCHAR2(30),
 text CLOB
)
/
```

Получаем: SQL> CREATE TABLE CLOBGET 2 ( 3 id NUMBER, 4 f\_name VARCHAR2(30), 5 text CLOB 6 )  
7 / Таблица создана.

Затем создадим контрольный файл для загрузки с именем **CLOB.ctl** и вот таким содержимым:

```
LOAD DATA

INFILE 'CLOB.DAT'

INTO TABLE CLOBGET

FIELDS TERMINATED BY ';'

(id, f_name, text LOBFILE (f_name) TERMINATED BY EOF)
```

Строка **LOBFILE ( f\_name ) TERMINATED BY EOF** и определяет загрузку **CLOB** объекта, опираясь на поле с именем файла, который должен находиться рядом. Далее создадим файл данных для загрузки в таблицу с именем **CLOB.DAT** и запишем в него следующее:

```
001;partone.txt
002;partone.txt
003;partone.txt
```

И последнее создадим сам файл для выполнения загрузки в таблицу с **CLOB** полем **getclob.bat** и запишем в него:

```
@echo off
```

```
set nls_lang=russian_cis.ru8pc866
```

```
sqlldr.exe userid=miller/kolobok control=CLOB.ctl errors=100 bad=CLOB.bad discard=CLOB.dis
```

Запустим файл на исполнение и убедимся, что все прошло верно заглянув в **log** файл:

```
.
.
.
```

Таблица CLOBGET, загружен из каждой логической записи.  
Режим вставки действует для этой таблицы: INSERT

| Имя столбца | Позиция | Дл. | Огр. | Вкл | Тип данных |
|-------------|---------|-----|------|-----|------------|
| ID          | FIRST   | *   | ;    |     | CHARACTER  |
| F_NAME      | NEXT    | *   | ;    |     | CHARACTER  |
| TEXT        | DERIVED | *   | EOF  |     | CHARACTER  |

Динамический LOBFILE. Имя файла в поле F\_NAME

Таблица CLOBGET:

3 Строки успешно загружено.  
0 Строки не загружены из-за ошибки в данных.  
0 Строки не загружены из-за сбоев во всех фразах WHEN.  
0 Строки не загружены из-за того, что все поля были пусты.

Для массива привязки отведено: 33024 байт(64 строк)  
Байтов буфера чтения: 1048576

Всего пропущено логических записей: 0  
Всего прочитано логических записей: 3  
Всего забраковано логических записей: 0  
Всего удалено логических записей: 0

```
.
.
```

Процессорное время: 00:00:00.03

Не знаю как у вас, у меня все прошло успешно и в поле **TEXT** таблицы **CLOBGET** у меня появилось три рассказа про черепах и рекурсию! Я в этом убедился используя свой старый добрый **PL/SQL\*Developer**! Вы можете использовать и другие средства. Давайте рассмотрим еще один способ, который вы можете так же использовать в дальнейшем! Сначала очистим таблицу **CLOBGET** от данных:

```
SQL> DELETE FROM CLOBGET
2 /
```

3 строк удалено.

```
SQL> commit
```

2 /

Фиксация обновлений завершена.

Далее вот так измените контролфайл используя загрузку вложенных данных, так будет удобнее:

```
LOAD DATA
INFILE *
INTO TABLE CLOBGET
FIELDS TERMINATED BY ';'
(id, f_name, text LOBFILE (CONSTANT turtles.dat) TERMINATED BY ';')

BEGIN DATA
001;partone
002;parttwo
003;parttri
```

Видите я в строке **LOBFILE (CONSTANT turtles.dat ) TERMINATED BY ';'**  заменил **EOF** на **';'** и вот почему, сейчас мы рассказ про черепах и рекурсию разделим на три части между тремя записями получив три части в полях **TEXT**, для этого создайте файл с именем **turtles.dat** и вот таким содержимым:

```
Here's my favorite story on recursion: it happened that a famous Darwinist was
telling a story about primitive ; creation myths. "Some peoples," he said, "believe the
world rests on the back of a great turtle. Of course, that raises the question: on what
does the turtle rest?"
An elderly woman from the back ; of the room stood up and said, "Very clever,
Sonny, but it's turtles, all the way down.";
```

Видите знаки **;"** после слов **primitive**, **back** и **down** - это определит части деления файла между строками таблицы, а параметр **CONSTANT** указывает на то, что мы получаем данные для **CLOB** из одного файла данных, все очень просто! Запустите загрузку и посмотрим содержимое **log** файла:

```
.
.
.
Таблица CLOBGET, загружен из каждой логической записи.
Режим вставки действует для этой таблицы: INSERT
```

| Имя столбца | Позиция | Дл. | Огр. | Вкл | Тип данных |
|-------------|---------|-----|------|-----|------------|
| ID          | FIRST   | *   | ;    |     | CHARACTER  |
| F_NAME      | NEXT    | *   | ;    |     | CHARACTER  |
| TEXT        | DERIVED | *   | ;    |     | CHARACTER  |

Статичный LOBFILE. Имя файла - turtles.dat

Таблица CLOBGET:

```
3 Строки успешно загружено.
0 Строки не загружены из-за ошибки в данных.
0 Строки не загружены из-за сбоев во всех фразах WHEN.
0 Строки не загружены из-за того, что все поля были пусты.
```

```
.
.
.
```

Процессорное время: 00:00:00.01

Все почти так же, только способ загрузки немного другой вот и все! Так можно загрузить вообще что угодно хоть видео клип, только для этого лучше использовать объект **BFILE** - но, о них позже. И напоследок загрузим пару тройку красивых картинок в **BLOB** поле таблицы! Создадим табличку с именем **IMGTBL**:

```
CREATE TABLE IMGTBL (
 id NUMBER(5),
 f_name VARCHAR2(30),
 img BLOB
)
/
```

Получаем:

```
SQL> CREATE TABLE IMGTBL (
2 id NUMBER(5),
3 f_name VARCHAR2(30),
4 img BLOB
5)
6 /
```

Таблица создана.

Создадим контрольный файл для загрузки изображений в **BLOB** поле таблички **IMGTBL** с именем **BLOBS.ctf** и вот таким содержимым:

```
LOAD DATA

INFILE *

INTO TABLE imgtbl

FIELDS TERMINATED BY ','
(id, f_name, img LOBFILE (f_name) TERMINATED BY EOF)

BEGINDATA
001,su3712.bmp
002,cu27.bmp
003,cu27_2.bmp
```

Файлы **su3712.bmp**, **cu27.bmp**, **cu27\_2.bmp** это фотографии СУ-37 и СУ-27, вы можете взять свои файлы и свои имена, только впишите их без пробелов между **001,su3712.bmp**! И последнее, файл **bat** для загрузки с именем **BLOBS.bat** и содержащим такие строки:

```
@echo off
```

```
set nls_lang=russian_cis.ru8pc866
```

```
sqlldr.exe userid=miller/kolobok control=BLOBS.ctl errors=100 bad=BLOBS.bad discard=BLOBS.dis
```

Что ж! Запускайте скорее файл и грузите ваши любимые картинки в базу! Смотрим **log** файл:

```
SQL*Loader: Release 9.2.0.1.0 - Production on Вск Май 30 15:43:54 2004
```

```
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
```

```
Управляющий файл: BLOBS.ctl
Файл данных: BLOBS.ctl
Файл плохих записей: BLOBS.bad
Файл удаленных записей: BLOBS.dis
(Разрешить удалять все записи)
```

```
Количество записей для загрузки: ALL
Количество записей для пропуска: 0
Допускается ошибок: 100
Массив привязки: 64 строк, макс. из 256000 байт
Продолжение: ничего не задано
Использован маршрут: Условный
```

```
Таблица IMG_TBL, загружен из каждой логической записи.
Режим вставки действует для этой таблицы: INSERT
```

| Имя столбца | Позиция | Дл. | Огр. | Вкл | Тип данных |
|-------------|---------|-----|------|-----|------------|
| ID          | FIRST   | *   | ,    |     | CHARACTER  |
| F_NAME      | NEXT    | *   | ,    |     | CHARACTER  |
| IMG         | DERIVED | *   | EOF  |     | CHARACTER  |

Динамический LOBFILE. Имя файла в поле F\_NAME

Таблица IMG\_TBL:

```
3 Строки успешно загружено.
0 Строки не загружены из-за ошибки в данных.
0 Строки не загружены из-за сбоев во всех фразах WHEN.
0 Строки не загружены из-за того, что все поля были пусты.
```

```
Для массива привязки отведено: 33024 байт(64 строк)
Байтов буфера чтения: 1048576
```

```
Всего пропущено логических записей: 0
Всего прочитано логических записей: 3
Всего забраковано логических записей: 0
```

Всего удалено логических записей: 0

Прогон начался в Вск Май 30 15:43:54 2004

Прогон кончился в Вск Май 30 15:43:59 2004

Общее время: 00:00:04.72

Процессорное время: 00:00:00.25

На этот раз было потрачено 04.72 сек. для загрузки моих файлов! Общий объем всех картинок у меня составил 6 838 440 bytes! Не плохо! И **PL/SQL\*Developer** мне все экспортировал и получились снова картинки! Что собственно и требовалось! Надеюсь вам стало понятно, что **SQL\*Loader** на первый взгляд не приметный и простой, является очень мощным и полезным инструментом для заливки ЛЮБЫХ(!) типов данных в БД **Oracle**! Так, что советую с ним подружиться и использовать его на всю катушку!!! :)))

---