Министерство образования и науки Российской Федерации

Севастопольский государственный университет

ИЗУЧЕНИЕ ОСНОВ ЯЗЫКА МАНИПУЛИРОВАНИЯ ДАННЫМИ SQL НА БАЗЕ СЕРВЕРА FIREBIRD

Методические указания

к лабораторной работе №№ 2-5

по дисциплине

“Управление данными”

для студентов специальности 09.03.02 –

"Информационные системы и технологии"

всех форм обучения

Севастополь
2014

УДК 004.92

Изучение основ языка манипулирования данными SQL на базе сервера Firebird.
Методические указания к лабораторной работе №1 по дисциплине “Управление данными”, для студентов всех форм обучения специальности 09.03.02 -"Информационные системы и технологии /Сост. Ю.В. Доронина, О.Л. Тимофеева, М.Р. Валентюк. - Севастополь: Изд-во СевНТУ, 2014.-20с.

Цель методических указаний: выработка у учащихся практических навыков по работе с реляционными базами данных.

Методические указания утверждены на заседании кафедры
Информационных Систем.
Протокол № от 2014 г.

Рецензент: доц. кафедры кибернетики и вычислительной техники, канд.техн.наук. Литвинова Л.А.

Допущено учебно-методическим центром в качестве методических указаний.

СОДЕРЖАНИЕ

Лабораторная работа № 2. SQL. Агрегатные функции 4

Лабораторная работа № 3. Создание схемы БД. Ссылочная целостность 8

Лабораторная работа № 4. Язык SQL. Запросы на основе нескольких таблиц 11

Лабораторная работа № 5. Язык SQL. Коррелированные вложенные подзапросы 15

Приложение А. Ограничения целостности данных 18

ЛАБОРАТОРНАЯ РАБОТА № 2.
SQL. АГРЕГАТНЫЕ ФУНКЦИИ

1. Цель лабораторной работы:

Изучение возможности обработки данных с помощью агрегатных функций языка SQL.

2. Основные положения

2.1. Агрегатные функции

Агрегатные функции – это функции, которые работают не с одним значением, взятым из строки таблицы, а с группой значений.

Общий алгоритм, по которому работают агрегатные функции следующий:

1. производится упорядочивание таблицы по тем полям, по которым осуществляется группировка;
2. от каждой сформированной группы вычисляется требуемое значение, которое и заносится в результат.

Существуют следующие агрегатные функции:

- COUNT считает количество строк, которые вернул запрос;
- SUM суммирует значение всех полей, по указанному атрибуту;
- AVG находит среднее значение поля, по указанному атрибуту;
- MAX находит максимальное значение поля, по указанному атрибуту;
- MIN находит минимальное значение поля, по указанному атрибуту.

2.2. Тестовая база данных

Рассмотрим работу агрегатных функций на примере базы данных, состоящий из трех таблиц – «Торговый агент», «Покупатель», «Сделка».

Таблица «Торговый агент» *(*Salespeople), представленная на рисунке 1, содержит следующие атрибуты:

- SNUM – номер агента;
- SNAME – имя агента;
- CITY – город, где работает агента;
- COMM – комиссионные торгового агента.

SNUM	SNAME	CITY	COMM
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Рисунок 1 – Таблица «Торговый агент»

Таблица «Покупатель» (Customers), представленная на рисунке 2,состоит из следующих атрибутов:

- CNUM – номер покупателя;
- CNAME – имя покупателя;
- CITY – город проживания покупателя;
- RATING – некоторый рейтинг, присвоенный покупателю;
- SNUM – номер торгового агента, за которым закреплен покупатель.

CNUM	CNAME	CITY	RATING	SNUM
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003
2003	Liu	SanJose	200	1002
2004	Grass	Berlin	300	1002
2006	Clemens	London	100	1001
2008	Cisneros	SanJose	300	1007
2007	Pereira	Rome	100	1004

Рисунок 2 - Таблица «Покупатель»

Таблица «Сделка» (Orders), представленная на рисунке 3, содержит следующие атрибуты:

- ONUM – номер сделки;
- AMT – сумма сделки
- ODATE – дата свершения сделки
- CNUM – номер покупателя
- SNUM – номер торгового агента

ONUM	AMT	ODATE	CNUM	SNUM
3001	18.69	10/03/1990	2008	1007
3003	767.19	10/03/1990	2001	1001
3002	1900.10	10/03/1990	2007	1004
3005	5160.45	10/03/1990	2003	1002
3006	1098.16	10/03/1990	2008	1007
3009	1713.23	10/04/1990	2002	1003
3007	75.75	10/04/1990	2004	1002
3008	4723.00	10/05/1990	2006	1001
3010	1309.95	10/06/1990	2004	1002
3011	9891.88	10/06/1990	2006	1001

Рисунок 3 - Таблица «Сделка»

2.3. Действия с базовыми агрегатными функциями

Чтобы найти сумму всех покупок в таблице «Сделки», можно ввести следующий запрос:

SELECT SUM (amt) **FROM** Orders;

Результат его выполнения – число 26658.4

Запрос

SELECT AVG (amt) **FROM** Orders;

вернет среднее значение сумм сделок -2665.84.

Функция COUNT отличается от других агрегатных функций тем, что она не выполняет математических действий над значением столбца. Она считает число значений в данном столбце, или число строк в таблице. Если необходимо подсчитать количество различных значений некоторого поля в таблице, функцию COUNT надо использовать с DISTINCT. Например, чтобы подсчитать количество продавцов в настоящее время описанных в таблице сделок, можно использовать следующий запрос:

SELECT COUNT (**DISTINCT** snum) **FROM** Orders;

Результат – 5, так как конструкция (**DISTINCT** snum) исключает записи с повторяющимися значениями.

Иногда возникает необходимость решить обратную задачу – подсчитать количество значений поля вместе с повторениями. Для этого существует описатель **ALL** (подразумевается по умолчанию).

Например:

Необходимо определить количество торговых агентов snum с учетом повторений.

SELECT COUNT (**ALL** snum) **FROM** Customers;

Результат – 7.

Чтобы подсчитать общее число строк в таблице, используется функция COUNT со звездочкой вместо имени поля, как, например, в следующем примере.

SELECT COUNT (*) **FROM** Customers;

Результат – 7.

Внимание! Только **COUNT** (*) может подсчитывать значения NULL. Все остальные функции игнорируют неопределенные значения.

2.4. Простые вычисления

Столбцы, значение которых может быть получено с помощью простых арифметических действий через другие столбцы в БД, как правило, не хранятся. SQL предоставляет простой способ производить подобные вычисления.. Также можно помещать в некоторый столбец константу.

Например, чтобы представить комиссионные торгового агента в процентном отношении, а не в виде десятичного числа можно записать такой запрос:

SELECT snum, sname, city, comm*100 **FROM** Salespeople;

Можно дополнить указанный запрос знаком процента, выводимым после поля comm*100:

SELECT snum, sname, city, comm*100, ' % ' **FROM** Salespeople;

Знак процента в данном случае – константное выражение. При выполнении вычислений в запросе допустимы арифметические действия – сложение, вычитание, умножение, деление.

2.5. Вычисления в агрегатных функциях

Допустим, необходимо вывести 10% от стоимости самой дорогой сделки. Тогда можно записать такой запрос:

SELECT MAX (amt*0.1) **FROM** Orders;

Таким образом, можно выполнять вычисления с использованием других полей и арифметических действий.

Внимание! Вложение агрегатов не допускается.

2.6. Использование GROUP BY

В случае если таблица рассматривается целиком, в списке вывода запроса присутствуют только агрегатные функции.

В случае если производится группировка таблицы по какому-то полю, в списке вывода могут присутствовать те поля, относительно которых таблица группируется и агрегатные функции. При этом все поля, не охваченные агрегатными функциями, должны быть перечислены в предложении GROUP BY.

Предположим, что необходимо найти сделку с максимальной стоимостью для каждого торгового агента. Можно сделать персональный запрос для каждого из них, выбрав MAX(amt) из таблицы сделок. Можно записать следующий запрос:

SELECT snum, **MAX** (amt) **FROM** Orders **GROUP BY** snum;

Результат выполнения запроса:

snum	
1001	9891.88
1002	5160.45
1003	1713.23
1004	1900.10
1007	1098.16

В данном случае таблица группируется относительно номера торгового агента, поэтому поле snum присутствует в списке вывода запроса и в предложении GROUP BY.

2.7. Использование HAVING

HAVING подобен WHERE – он задает условия отбора групп строк так же, как это делает WHERE для каждой строки.

Внимание! В предложении HAVING нельзя проверять имена атрибутов на какое-либо условие – для этого существует WHERE. То, что проверяется в HAVING должно иметь только одно значение для группы.

Например, если необходимо узнать, какие торговые агенты имеют заработок от одной сделки более чем 3000, и в какой день, можно использовать запрос вида:

```
SELECT snum, odate, MAX (amt)
FROM Orders
GROUP BY snum, odate
HAVING MAX (amt) > 3000.00;
```

Результат работы запроса:

Snum	Odate	------
1001	10/05/1990	4723.00
1001	10/06/1990	9891.88
1002	10/03/1990	5160.45

3. Порядок выполнения лабораторной работы:

1. Ознакомиться с принципами работы агрегатных функций COUNT, SUM, AVG, MAX, MIN.
2. Продемонстрировать использование COUNT(*)
3. Продемонстрировать выполнение простых вычислений в запросе.
4. Использовать простое вычисление как параметр агрегатной функции.
5. Ознакомиться с использованием предложения GROUP BY, продемонстрировать его работу.
6. Ознакомиться с использованием предложения HAVING, продемонстрировать его работу.
7. Составить и защитить отчет.

4. Задание на работу

Вариант задания, соответствует варианту, полученному в Лабораторной работе №1.

5. Содержание отчета

1. Отчет состоит из титульного листа, цели работы, описания процесса выполнения работы и вывода.
2. Отчет должен содержать таблицу исходных данных, тексты запросов и результаты их выполнения.

6. Контрольные вопросы

1. Назначение агрегатных функций?
2. Возможности предложения GROUP BY?
3. Условия отбора групп. Предложения HAVING и WHERE назначение и отличия в использовании?
4. Простые вычисления над данными?
5. Требования к списку выводимых столбцов фразы SELECT при задании группировки таблицы по какому-либо полю?

ЛАБОРАТОРНАЯ РАБОТА № 3.
СОЗДАНИЕ СХЕМЫ БД. ССЫЛОЧНАЯ ЦЕЛОСТНОСТЬ

1. Цель лабораторной работы:

Научиться анализировать предметную область с целью создания схемы БД, учитывая ссылочную целостность набора.

2. Основные положения

2.1. Типы связей между отношениями. Требования к ссылочной целостности данных

Свое название реляционные базы данных получили именно по той причине, что таблицы в БД не существуют независимо друг от друга. Таблицы взаимосвязаны друг с другом, т.е. действие, произведенное в одной таблице, вызовет некоторые действия в другой таблице.

Существует три основных класса связей между таблицами:

- один к одному (1:1);
- один ко многим (1:М);
- многие ко многим (М:М).

На практике связи первого типа используются редко. Связи третьего типа не реализуются в РБД напрямую, одну связь многие ко многим приводят к двум связям один ко многим. Поэтому связь 1:М используется наиболее часто, ее следует рассмотреть наиболее подробно.

Пусть имеется две таблицы следующего вида:

1. **Клиент** (Номер_клиента, ФИО_клиента, Счет_клиента)
2. **Продажи**(Номер_клиента, Наименование_товара, Количество_товара)

Таблица **Клиент** представлена на рисунке 1. Таблица **Продажи** представлена на рисунке 2.

Таблицы связанны отношением 1:М, т.е. один клиент может совершить много покупок, каждая покупка совершается только одним клиентом. В таблицах хранятся следующие данные:

Номер_клиента	ФИО_клиента	Счет_клиента
1	Сидоров И.И.	1923563
2	Петров А.Б.	1736523
3	Федоров Н.Г.	9265623

Рисунок 1 – Таблица **Клиент**.

Номер_клиента	Наименование_товара	Количество_товара
1	Шоколад «Корона»	2
1	Шейки Куринные	1,5
2	Крупа манная	3

Рисунок 2 – Таблица **Продажи**.

Очевидно, что поле Номер_клиента в обеих таблицах имеет одинаковый смысл - оно обозначает номер, присвоенный клиенту. В случае если таблицы не имеют связи друг с другом, ничего не мешает добавить в таблицу «Продажи» запись, например, (8, «Хлеб белый»,1), несмотря на то, что в таблице «Клиент» нет покупателя с номеров 8. Это нарушение целостности данных, так как такая строка может быть занесена, но она не соответствует действительности. Чтобы подобная ситуация стала невозможной, необходимо установить связь между таблицами по полю «Номер_клиента». В этом случае сервер БД автоматически проследит, чтобы поле «Номер_клиента» из вставляемой строки существовало в таблице «Клиенты».

Подобной проверки достаточно, чтобы условие целостности данных выполнялось при добавлении данных в таблицы. Но его недостаточно при манипулировании данными.

Рассмотрим ситуацию, когда необходимо удалить из таблицы «Клиент» некоторую запись, например клиента с номером 1. В случае если таблицы не связанны, удаление клиента повлечет за собой изменение только одной таблицы. В таблице «Продажи» останутся сведения о покупках покупателя с номером 1. Такая ситуация - также нарушение целостности данных, так как о данном покупателе, после его удаления, базе данных ничего не известно. В случае если таблицы связанны, удаление покупателя может повлечь за собой удаление всех его покупок (говорят, что удаление каскадируется). Использование конструкций оператора CREATE TABLE для обеспечения целостности данных приведено в Приложении А.

Такая же ситуация с модификацией данных в родительской таблице. Если покупатель с номером 1 изменил номер на 5, то в связанных таблицах изменение родительской таблицы повлечет за собой автоматическое изменение дочерних таблиц.

Кроме арности связи и ограничений целостности различают также идентифицирующие и не идентифицирующие связи.

С помощью идентифицирующей связи устанавливается взаимосвязь между зависимыми сущностями (клиент и его покупки). Не идентифицирующая связь устанавливает взаимосвязь между независимыми сущностями (автомо-

биль и его цвет). Разница между идентифицирующей и не идентифицирующей связью состоит в том, что при идентифицирующей связи внешний ключ является частью первичного ключа дочерней сущности, а при не идентифицирующей связи внешний ключ не входит в первичный ключ дочерней сущности. На **ER-диаграмме** идентифицирующая связь изображается непрерывной линией, не идентифицирующая - пунктирной.

2.2. Стандартная нотация ER-диаграмм

Приведем пояснения к стандартной нотации ER-диаграмм, в которой представлены варианты к лабораторной работе.

Сущности

Сущность изображается прямоугольником, над которым пишется имя сущности. Одна сущность соответствует одной таблице. Прямоугольник разделяется линией на две части. В верхней части указываются ключевые атрибуты. После имени атрибута могут стоять символы, уточняющие назначение атрибута. Допустимые символы:

- PK - первичный ключ
- AK - альтернативный ключ
- FK - внешний ключ
- IE - inversion entry, говорит о том, что по данному полю должен быть создан индекс.

Связи

Связь между двумя отношениями изображается с помощью линии. Идентифицирующая связь изображается сплошной линией, не идентифицирующая - пунктирной. Арность связи указывается следующим образом: со стороны "многие" ставится жирная точка, со стороны "один" точка не ставится. Допустимость Null -значений изображается ромбиком с той стороны связи, где позволяются Null -значения.

Категории

В случае если у нас есть некоторая общая сущность (например – люди), и необходимо хранить информацию о некоторых разновидностях данной сущности (например – родители и дети, мужчины и женщины, работающие и безработные и т.д.) имеет место понятие категории.

Рассмотрим на примере использование понятия категории. Допустим, нам необходимо хранить следующую информацию о родителях и детях:

Родитель (Номер, ФИО, Адрес, Образование)
Ребенок (Номер, ФИО, Адрес, Номер_детского_садика)
ЯвляетсяРебенком(Номер_родителя, Номер_ребенка)

Схема данных:
Родитель ------- 1:M -------> **Ребенок**

Очевидно, что для ребенка поле «Образование» не имеет смысла, как и поле «Номер_детского_садика» для родителя. Кроме того, в случае, если подобную информацию хранить в виде двух таблиц, то возникает дублирование информации и проблемы с целостностью данных. Например, адрес ребенка совпадает с адресом родителей, и если данные вводятся вручную, то можно ввести адрес ребенка отличный от адреса родителя.

Подобная проблема решается следующим образом: вводится обобщающая категория «Личность», в нее выносятся общие поля из двух дочерних таблиц. Дочерние таблицы привязываются к родительской связью один к одному.

Пример:

Личность (Номер, ФИО);
Родитель (Номер, Адрес, Образование);
Ребенок (Номер, Номер_детского_садика);
ЯвляетсяРебенком(Номер_родителя, Номер_ребенка)

Схема:

Родитель --------1:1--------> Личность<--------1:1-------- Ребенок

Для того чтобы добавить данные о ребенке, нужно занести данные в две таблицы – «личность» и «ребенок». Например:

ISERT INTO Личность VALUES (2,‘Иванов И.И.’)
ISERT INTO Ребенок VALUES (2,‘д/с №23’)

Нумерация в таблицах сквозная!

На рисунке 3 изображена схема категориальной связи.

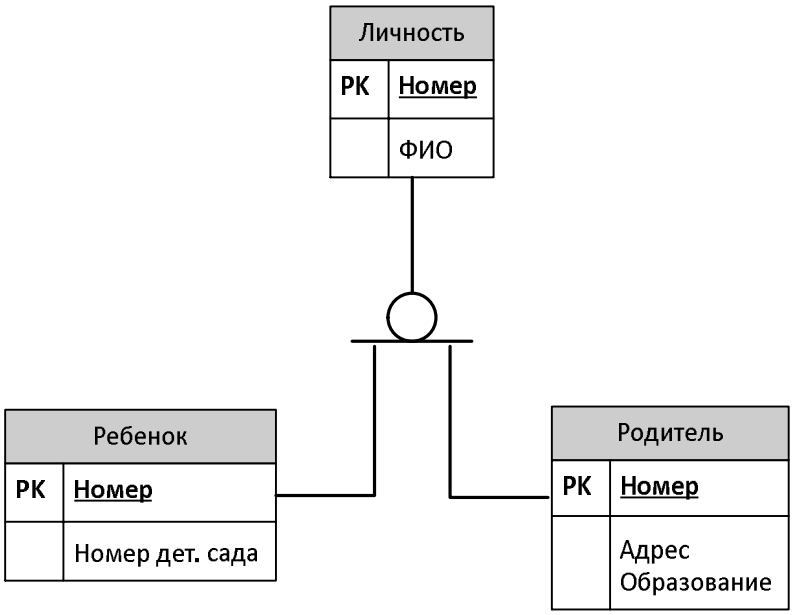

Рисунок 3 - Схеме категориальной связи

Допустимость NULL-значений связи на диаграмме указывается ромбиком.

Связь таблиц в реляционной БД устанавливается при создании таблиц с помощью описателя FOREIGEN KEY (см. приложение А).

3. Порядок выполнения лабораторной работы

1. Проанализировать схему БД своего варианта задания (вариант то же, что и в лабораторной работе №1), выделить и классифицировать все существующие связи, определить необходимые ограничения целостности.
2. Создать все еще не созданные таблицы, изменить существующие таким образом, чтобы они могли участвовать в связях (описание ALTER TABLE).
3. Установить связи между таблицами.
4. Проверить работу ограничений целостности (каскадирование удаления, модификации и др.)
5. Подготовить и защитить отчет.

4. Содержание отчета

1. Отчет состоит из титульного листа, цели работы, описания процесса выполнения работы и вывода.
2. Отчет должен содержать исходные и модифицированные таблицы.
3. В отчете необходимо привести список связей по арности, описать ограничения целостности, допустимость Null-значений, идентифицируемость.
4. Отражение работы по проверке целостности данных.

5. Контрольные вопросы

1. Требования к ссылочной целостности данных?
2. Типы связей между отношениями?
3. Стандартная нотация ER-диаграмм?
4. Нормальные формы для баз данных?
5. Необходимость процесса нормализации БД?
6. Приведение БД к нормальной форме Бойса-Кодда?
7. Обосновать в какой нормальной форме находится полученная схема БД?
8. Способы повышения надежности данных?

ЛАБОРАТОРНАЯ РАБОТА № 4.
ЯЗЫК SQL. ЗАПРОСЫ НА ОСНОВЕ НЕСКОЛЬКИХ ТАБЛИЦ

1. Цель работы:

- изучить способы получения информации из нескольких таблиц
- изучить способы выполнения и принцип действия рекурсивных запросов
- научится использовать вложенные подзапросы

2. Основные положения

Запросы к нескольким таблицам являются наиболее часто используемым типом запросов. Реляционные БД нормализованы, и хранимая информация разбита по большому количеству таблиц**.**

Структура, используемых отношений (Salespeople, Customers, Orders) и их содержимое приведено в лабораторной работе №2 п.2.2 Тестовая база данных.

2.1. Простое соединение двух таблиц

Существует два основных способа соединения таблиц – с помощью оператора JOIN языка SQL, и с помощью условия в разделе WHERE. Например, соединить таблицу Customers и таблицу Salespeople по полю city можно следующим образом:

```
SELECT
        Customers.cname,
        Salespeople.sname,
        Salespeople.city
FROM
        Salespeople, Customers
WHERE
        Salespeople.city = Customers.city;
```

Для выполнения данного запроса сервер произведет декартово произведение двух таблиц, после чего выполнит операцию селекции нужных строк, затем – проекцию нужных полей. Декартово произведение – это операция, которая требует для своего выполнения максимальное количество памяти и процессорного времени, по сравнению с другими известными операциями. Данный способ неэффективен.

В случае если используется, оператор JOIN, запрос будет выглядеть так:

```
SELECT Customers.cname,
          Salespeople.sname,
          Salespeople.city
FROM
Salespeople JOIN Customers ON Salespeople.city = Customers.city;
```

В данном случае сервер БД не будет производить декартово произведение, сервер воспользуется индексами для определения совпадающих значений поля city, выберет строки с совпадающими значениями, а затем произведет проекцию. Данная операция происходит на порядок быстрее. В случае, если по полям учавствующим в соединении, не создан индекс, сервер вынужден производить декартово произведение.

Можно сделать следующие выводы:

- если соединение таблиц происходит по ключам, выгоднее использовать JOIN;
- если запрос выполняется медленно, нужно создать индекс по полю – параметру соединения (CREATE INDEX)

2.2. Соединение более чем двух таблиц

Соединение более чем двух таблиц по форме не отличается от простого соединения. Например, чтобы соединить три таблицы по некоторому условию, можно записать:

```
SELECT onum, cname, Orders.cnum, Orders.snum
FROM Salespeople, Customers, Orders
        WHERE
    Customers.city <> Salespeople.city AND
    Orders.cnum = Customers.cnum AND
    Orders.snum = Salespeople.snum;
```

Тот же запрос, переписанный с использованием JOIN, выглядит более сложно:

```
SELECT onum, cname, Orders.cnum, Orders.snum
```

```
FROM (Orders JOIN Customers ON Orders.cnum = Customers.cnum) JOIN Salespeople ON Orders.snum =
Salespeople.snum
        WHERE
    Customers.city <> Salespeople.city AND
```

2.3. Псевдонимы и рекурсивные объединения

На практике часто встречается ситуация, когда необходимо соединять таблицу саму с собой. Запрос, выполняющий такое соединение, называется рекурсивным.

Таблица «Студент» *(*Students*)*, представленная на рисунке 1, содержит следующие атрибуты:

- CNAME – имя студента;
- RATING – рейтинг студента;
- SPES – специальность.

CNAME	RATING	SPES
Иванов	42	ИС
Калинин	46	А
Петров	42	А
Сидоров	34	ИС
Шахов	46	М

Рисунок 1 – Таблица «Студент»

Например, если необходимо вывести все пары студентов с одинаковым рейтингом, можно записать следующий запрос:

```
SELECT
        first.cname,
        second.cname,
        first.rating
FROM
        Students first,
        Students second
WHERE
        first.rating = second.rating;
```

Здесь в списке вывода SELECT указываются поля cname и rating таблицы first, и поле cname таблицы second. В разделе FROM указывается, что first и second – просто псевдонимы для таблицы Students. Для выполнения запроса сервер создаст две копии таблицы Students, одну с именем first, другую с именем second, выполнит запрос так, как будто это две разные таблицы, и уничтожит копии. Естествнно, сервер физически не создает копий таблиц, но с псевдонимами в запросе он работает так, будто это две разные таблицы.

Можно заметить, что вывод запроса будет повторять каждую пару дважды - сначала «Иванов - Петров», затем «Петров - Иванов». Кроме того, вывод содержит строки «Иванов - Иванов», «Петров - Петров». Это происходит потому, что сервер берет первую строку из первого псевдонима и сравнивает ее со всеми строками из второго псевдонима. Будут выбраны строки «Иванов - Иванов» и «Иванов - Петров». Затем он переходит к следующей строке и снова сравнивает ее со всеми строками из второго псевдонима, и так далее. Будут выбраны строки «Петров - Иванов» и «Петров - Петров».

Чтобы избежать дубликатов, надо определить еще одно условие, налагающее отношение порядка на сравниваемые строки. Например, сравнивать имена.

```
SELECT
        first.cname,
        second.cname,
        first.rating
FROM
        Students first,
        Students second
WHERE
        first.rating = second.rating AND
        first.cname < second.cname;
```

2.4. Вложенные подзапросы

Вложенные подзапросы так же служат для получения информации из нескольких таблиц. С их помощью можно выполнять рекурсивные запросы. Очень часто на практике встречаются ситуации, когда запрос выражается очень

сложно через соединения, и легко – через вложенный подзапрос, и наоборот. На конкретном сервере БД запрос с использованием JOIN может выполняться очень долго, а с использованием подзапроса – быстро.

Пример запроса с вложенным подзапросом:

```
SELECT *
FROM Orders
WHERE snum =
        ( SELECT snum
        FROM Salespeople
        WHERE sname = 'Motika');
```

Так как подзапрос стоит после знака равенства, он должен возвращать только одно значение. В случае если подзапрос вернет более чем одно значение, произойдет ошибка.

Внимание! При записи подзапроса допустима следующая форма:

<имя/константа> <оператор> <подзапрос>,

а не <подзапрос> <оператор> <имя/константа> или

< подзапрос > < оператор > < подзапрос >.

Во вложенных подзапросах можно использовать агрегатные функции:

```
SELECT *
FROM Orders
WHERE amt >
        (SELECT AVG (amt)
         FROM Orders
         WHERE odate = 10/04/1990 );
```

Ограничение на вложенный подзапрос то же самое – он должен возвращать единственное значение.

В случае, если подзапрос возвращает несколько записей, вместо операций сравнения нужно использовать IN:

```
SELECT *
FROM Orders
WHERE snum IN
        ( SELECT snum
          FROM Salespeople
          WHERE city = "LONDON" );
```

Данный запрос более просто записывается с использованием соединения:

```
SELECT onum, amt, odate, cnum, Orders.snum
FROM Orders, Salespeople
WHERE Orders.snum = Salespeople.snum
        AND Salespeople.city = "London";
```

Допустимо использовать выражение, основанное на столбце, а не просто сам столбец в предложении SELECT подзапроса:

```
SELECT *
FROM Customers
WHERE cnum =
 ( SELECT snum + 1000
   FROM Salespeople
   WHERE sname = Serres );
```

Также допустимы подзапросы в выражении HAVING:

```
SELECT rating, COUNT ( DISTINCT cnum )
FROM Customers
GROUP BY rating
HAVING rating >( SELECT AVG (rating)
                 FROM Customers
                 WHERE city = " San Jose');
```

3. Порядок выполнения лабораторной работы

1. Записать запросы, соединяющие две таблицы с помощью JOIN и без него.
2. Записать запросы, соединяющие более чем две таблицы с помощью JOIN и без него.
3. Продемонстрировать следующие возможности SQL:

- использование псевдонимов на примере рекурсивного запроса.

- привести пример запроса с подзапросом
- использование агрегатных функций в подзапросе
- подзапросы, возвращающие единственное и множественные значения
- подзапросы, использующие вычисление
- использование подзапросов в HAVING

4. Написать отчет.

4. Содержание отчета

1. Отчет состоит из титульного листа, цели работы, описания процесса выполнения работы и вывода.
2. Отчет должен содержать исходные данные, тексты запросов и результаты их выполнения.

5. Контрольные вопросы

1. Операции реляционной алгебры (операторной формы записи)?
2. Бинарные операторы над отношениями, операторная форма записи?
3. В чем различие соединения таблиц по условию и с использованием JOIN?
4. Свойства операции соединения?
5. В чем различие вложенных запросов и запросов с соединением?
6. Какие формы записи подзапроса недопустимы?
7. В чем особенность подзапроса, перед которым стоит знак арифметического сравнения?

ЛАБОРАТОРНАЯ РАБОТА №5.
ЯЗЫК SQL. КОРРЕЛИРОВАННЫЕ ВЛОЖЕННЫЕ ПОДЗАПРОСЫ

1. Цель работы:

Ознакомится с принципом работы коррелированных подзапросов

2. Основные положения

2.1. Коррелированные вложенные подзапросы

Вложенные подзапросы и операция JOIN предоставляют большие возможности по манипулированию данными из нескольких таблиц. Существует еще один способ - использовать коррелированные подзапросы, он более сложен для понимания, но в некоторых случаях позволяет формулировать запросы самым коротким образом. Кроме того, есть некоторые вещи, доступные для соединения и недоступные для вложенных подзапросов, и наоборот.

Например, при использовании подзапроса становятся возможным использовать агрегатные функции в предикате запроса. При использовании соединений в выводе запроса могут участвовать поля из всех таблиц соединения.

Коррелированным называется подзапрос, который использует псевдоним таблицы определенный не во вложенном запросе, а во внешнем. При этом вложенный запрос выполняется много раз - для каждой строки внешней таблицы. Пусть требуется найти всех покупателей, совершивших покупки 10.03.1990. Можно записать следующий коррелированный запрос:

```
SELECT *
FROM Customers outer
WHERE 10.03.1990 IN
 ( SELECT odate
    FROM Orders inner
    WHERE outer.cnum = inner.cnum );
```

Логика работы запроса такова: внутренний запрос производит соединение таблицы покупателей и таблицы покупок (outer.cnum = inner.cnum). Для тех строк, у которых условие WHERE выполняется, запрос возвращает дату покупки. Таким образом, для каждого покупателя формируется множество дней, когда он делал покупки. Полученное множество передается во внешний запрос. В случае если 10.03.1990 входит в сформированное множество, внешний запрос возвращает всю строку (*) строку из таблицы Customers.

Приведем процесс выполнения коррелированного подзапроса в алгоритмической форме:

1. Выбрать строку из таблицы во внешнем запросе (строка-кандидат).
2. Сохранить значения полей строки-кандидата в псевдониме.
3. Выполнить подзапрос. Везде, где найдена ссылка на псевдоним из внешнего запроса, подставлять значения из текущей строки-кандидата. Использование значения из строки-кандидата называется внешней ссылкой.
4. Оценить предикат внешнего запроса на основе результатов подзапроса. Если предикат верен - строка выбирается для вывода
5. Перейти на следующую строку во внешнем запросе.

Подобный запрос можно гораздо легче записать другим способом. Например:

```
SELECT DISTINCT *
 FROM Customers, Orders
 WHERE Customers.cnum = Orders.cnum
   AND Orders.odate = 10.03.1990;
```

Более показательным является следующий пример. Пусть необходимо вывести имена и номера всех продавцов, которые имеют более одного заказчика

```
SELECT snum, sname
  FROM Salespeople main
  WHERE 1 <
    ( SELECT COUNT (*)
       FROM Customers
       WHERE snum = main.snum );
```

В случае, если перед именем поля не указывается имя псевдонима, то данное поле относится к текущему подзапросу. Таким образом, поле snum во вложенном подзапросе относится к таблице Customers, а не к Salespeople. В случае, если такого поля в соответствующей таблице нет, оно ищется в подзапросе верхнего уровня. Так, если бы поле snum отсутствовало в таблице Customers, оно относилось бы к Salespeople.

2.2. Соотнесение таблицы со своей копией

Коррелированные запросы можно писать, основываясь только на одной таблице. Данное свойство делает их незаменимыми при написании аналитических запросов (т.е. запросов, производящих сложный анализ данных). Например, пусть нам необходимо найти все покупки со значениями суммы покупки выше среднего значения для каж-

дого покупателя (т.е. надо найти все покупки данного покупателя, найти среднюю сумму покупок, и выдать все покупки со стоимостью выше средней):

```
SELECT *
FROM Orders outer
WHERE amt > =
  ( SELECT AVG (amt)
    FROM Orders inner
    WHERE inner.cnum = outer.cnum );
```

Существуют так называемые специальные операторы SQL, они имеют смысл только для подзапросов. Отличительная особенность специальных операторов - они принимают подзапрос как аргумент, точно так же, как это делает IN.

2.3. Оператор EXISTS

EXISTS(X) - булевский оператор. Он получит значение "истинна", если запрос X вернет хоть одну строку. Допустим, нам необходимо узнать список продавцов, у которых есть более одного покупателя

```
SELECT DISTINCT snum
  FROM Customers outer
  WHERE EXISTS
    ( SELECT *
      FROM Customers inner
      WHERE inner.snum = outer.snum
        AND inner.cnum < > outer.cnum );
```

Для каждой строки-кандидата из внешнего запроса (продавец, проверяемый в настоящее время), внутренний запрос находит строки, у которых совпадают значения поля snum (номер продавца), но не совпадают значение поля cnum (номер покупателя). Если не указать DISTINCT, каждый продавец будет выбран один раз для каждого своего заказчика.

На практике очень часто приходится использовать EXISTS вместе с NOT. Допустим, нам необходимо вывести список продавцов, за которыми числится только один покупатель

```
SELECT DISTINCT snum
FROM Customers outer
WHERE NOT EXISTS
    ( SELECT *
      FROM Customers inner
      WHERE inner.snum = outer.snum
          AND inner.cnum < > outer.cnum );
```

Кроме EXISTS в подзапросах возможно использование еще двух операторов - ANY и ALL.

2.4. Оператор ANY

Оператор ANY становится верным, если значение из верхнего подзапроса совпадает, по крайней мере, с одним значением из вложенного подзапроса. Например, если значение - кандидат равно 1, а вложенный подзапрос вернул {1, 2, 3}, оператор ANY станет верным, (внешний запрос проверяет на равенство). Например, если нам нужно найти всех продавцов, которые живут в тех же городах, что и покупатели, можно записать следующий запрос:

```
SELECT *
FROM Salespeople
WHERE city = ANY
  (SELECT city
    FROM Customers );
```

Существует синоним оператора ANY - SOME. Так как SQL похож на английский, то одни предложения, верно, звучат с ANY, другие - с SOME. Действие операторов эквивалентно.

2.5. Оператор ALL

Вторым допустимым оператором является ALL. Действие его противоположно оператору ANY. Оператор ALL становится верным, если все значения из вложенного подзапроса равны значению-кандидату из внешнего запроса. Например, если значение - кандидат равно 1, а вложенный подзапрос вернул {1, 1, 1}, оператор ALL станет верным, (внешний запрос проверяет на равенство).

Например, если нам необходимо найти всех продавцов, у которых рейтинг выше, чем у любого продавца из Рима, можно записать следующий запрос:

```
SELECT *
  FROM Customers
  WHERE rating > ALL
    (SELECT rating
       FROM Customers
       WHERE city = Rome );
```

Операторы ANY и ALL можно выразить через EXISTS в коррелированном подзапросе, в явном виде они нужны лишь для упрощения записи запроса. Обратное утверждение не верно - т.е. не все то, что можно выполнить с помощью EXISTS, можно сделать с помощью ANY и ALL

Замечание 1: Когда говорят, что значение больше (или меньше) чем любое (ANY) из набора значений, это то же самое, что сказать, что оно больше (или меньше) чем любое отдельно взятое из этих значений. И наоборот, сказать что некоторое значение не равно всему (ALL) набору значений, это то же самое, что сказать, что в наборе нет такого же значения.

Замечание 2: В случае если подчиненный запрос вернул пустое множество, оператор ALL становится верным, а ANY - ложным.

3. Порядок выполнения лабораторной работы

1. Записать запрос, соединяющий таблицу со своей копией.
2. Привести пример коррелированного запроса, использующего две разные таблицы.
3. Продемонстрировать следующие возможности SQL
   - работу оператора EXISTS;
   - работу оператора ALL;
   - работу оператора ANY.
4. Написать отчет.

4. Содержание отчета

1 Отчет состоит из титульного листа, цели работы, описания процесса выполнения работы и вывода.

2 Отчет должен содержать исходные данные, тексты запросов и результаты их выполнения.

3 Контрольные вопросы

1. Коррелированные вложенные подзапросы?
2. Для чего таблицам назначается псевдоним? Приведите примеры назначения и использования псевдонимов в запросах.
3. Операторы: EXISTS, ALL, ANY – назначение, правила использования?
4. Какой из операторов EXISTS, ALL, ANY являются альтернативой друг другу? Приведите примеры.

ПРИЛОЖЕНИЕ А
(обязательное)
Ограничения целостности данных

Ограничения целостности - это правила, которые отслеживают достоверность данных. Ограничения целостности могут проверять значение, вводимое в один столбец (например, возраст не может быть отрицательным, и не может быть больше 150 лет), может проверять значение, вводимое в несколько столбцов например, если в поле "возраст" введено значение 10 лет и менее, то в поле "количество детей" не может быть значение большее 0, может проверять несколько таблиц (например, при удалении заказчика надо удалить все его заказы). Ограничения, проверяющие значение, вводимое в столбец называются ограничениями на уровне атрибута, проверяющие несколько столбцов – ограничения на уровне таблицы, проверяющие несколько таблиц – ограничения на уровне связи.

CREATE TABLE может создавать следующие типы ограничений целостности:

- PRIMARY KEY (первичный ключ) - уникально идентифицирует каждую строку таблицы. Значение в указанном столбце либо в упорядоченном наборе столбцов не могут повторяться в более чем одной строке. Столбец PRIMARY KEY должен быть определен только с атрибутом NOT NULL. Таблица может иметь только один PRIMARY KEY, который может быть определен на одном или более столбцов.
- UNIQUE (уникальные) ключи гарантируют, что не существует двух строк имеющих одно и тоже значение в указанном столбце или упорядоченном наборе столбцов. Уникальный столбец должен быть определен с атрибутом NOT NULL. Таблица может иметь один или более UNIQUE ключей.

В случае, если первичный или альтернативный ключ состоит из одного поля, его можно описать в той же строке, что и само поле (такое ограничение называется ограничением на уровне столбца). Если первичный или альтернативный ключ состоит из нескольких атрибутов, он описывается после определения всех полей таблицы (такое ограничение называется ограничением на уровне таблицы).

Ограничение на уровне столбца имеет следующий синтаксис:
<col_constraint> = [CONSTRAINT <имя ограничения>] <определение ограничения>.

т.е. определение ограничения состоит из необязательной части, состоящей из служебного слова CONSTRAINT и имени ограничения, и обязательной части, состоящей из собственно определения ограничения. В случае, если первая часть опущена, сервер генерирует имя ограничения автоматически. Посмотреть имя ограничения можно просмотрев системную таблицу RDB$RELATION_CONSTRAINTS.

Определение ограничения на уровне столбца описывается так:

<определение ограничения> = {
UNIQUE | **PRIMARY KEY**|**CHECK** (<условие проверки>)
| **REFERENCES** <ИмяДругойТаблицы> [(<имя столбца>[, <имя столбца> …])]
[**ON DELETE** {**NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL**}]
[**ON UPDATE** {**NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL**}]
}

Поле можно определить как первичный ключ, его значение можно сделать уникальным по таблице, либо можно указать, что поле ссылается (REFERENCES) на некоторое поле из другой таблицы. Таблица, на которую ссылается поле, называется родительской. Смысл ограничения REFERENCES следующий: значение в поле в дочерней таблицы можно занести только в том случае, если это значение есть в соответствующем поле родительской таблицы. Например, если у нас есть таблица клиентов, и таблица покупок, поле “Номер клиента” из таблицы покупок должно ссылаться на поле “Номер клиента” из таблицы клиентов. Таким образом в таблицу покупое невозможно будет занести номер несуществующего клиента. Вторая часть описателя REFERENCES описывает, какие действия необходимо предпринять при удалении из родительской таблицы (ON DELETE) и при обновлении родительской таблицы (ON UPDATE).

Всего возможно четыре варианта:

- NO ACTION - ничего не предпринимать
- CASCADE – каскадировать. В случае удаления из родительской таблицы будут удалены все связанные записи из дочерней таблицы. Например, удаление клиента повлечет удаление всех его покупок. При модификации записи из родительской таблицы будут модифицированы также записи в дочерней таблице. Например, если клиент поменял номер с пятого на десятый, во всех его покупках поле “Номер клиента” также изменит свое значение с 5 на 10.
- SET DEFAULT – поле получает свое значение по умолчанию
- SET NULL – поле устанавливается в NULL

Какое именно действие необходимо предпринимать зависит от смысла таблиц. При удалении клиента удалять все его покупки имеет смысл. Но при удалении желтого цвета из таблицы цветов не стоит удалять все желтые автомобили.

Ограничения целостности на уровне таблицы описывается так же, как и ограничение на уровне столбца, за исключением определения самого ограничения .

<ограничение> = {{**PRIMARY KEY** | **UNIQUE**} (<имя столбца> [,<имя столбца> …])
| **FOREIGN KEY** (<имя столбца> [,<имя столбца> …]) **REFERENCES** <ИмяДругойТаблицы>
[**ON DELETE** {**NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL**}]
[**ON UPDATE** {**NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL**}]
| **CHECK** (<условие>)}

Отличие в том, что в PRIMARY KEY, UNIQUE и FOREIGN KEY можно описать более одного столбца.

Ссылочные ограничения гарантируют, что значения в наборе столбцов, которые определены в FOREIGN KEY принимают те же самые значения, которые присутствуют в столбце UNIQUE или PRIMARY KEY в таблице, на которую они ссылаются. Это ограничение на уровне связи.

1. Следующая инструкция создает одиночную таблицу с PRIMARY KEY:

```
CREATE TABLE COUNTRY
  (COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
  CURRENCY VARCHAR(10) NOT NULL);
```

2. Следующая инструкция создает таблицу с UNIQUE ограничением и на уровне столбца MODEL и на уровне таблицы (столбцы MODELNAME, ITEMID):

```
CREATE TABLE STOCK
  (MODEL SMALLINT NOT NULL UNIQUE,
  MODELNAME CHAR(10) NOT NULL,
  ITEMID INTEGER NOT NULL,
  CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID));
```

3. Следующая инструкция создает таблицу с вычисляемым столбцом NEW_SALARY:

```
CREATE TABLE SALARY_HISTORY
  (EMP_NO EMPNO NOT NULL,
  CHANGE_DATE DATE DEFAULT "NOW" NOT NULL,
  UPDATER_ID VARCHAR(20) NOT NULL,
  OLD_SALARY SALARY NOT NULL,
  PERCENT_CHANGE DOUBLE PRECISION  DEFAULT 0  NOT NULL
                          CHECK (PERCENT_CHANGE BETWEEN -50 AND 50),
  NEW_SALARY COMPUTED BY (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
  PRIMARY KEY (EMP_NO, CHANGE_DATE, UPDATER_ID),
  FOREIGN KEY (EMP_NO) REFERENCES EMPLOYEE (EMP_NO));
```

Пояснения к оператору. Объявлены поля:
поле **EMP_NO** целочисленное, не может принимать значение NULL.
поле **CHANGE_DATE** имеет тип DATE**,** значение по умолчанию – текущая дата, не может принимать значение NULL.
поле **UPDATER_ID** – строка переменной длины с максимальной длиной 20 символов, не может принимать значение NULL.
поле **OLD_SALARY** – содержит денежное значение, которое хранится с точностью до трех знаков после запятой, не может принимать значение NULL.
поле **PERCENT_CHANGE** вещественное число, по умолчанию имеет значение 0, не может принимать значение NULL.
Поле **NEW_SALARY** вычисляется как

OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100

Объявлены ограничения:

- ограничение CHECK – поле **PERCENT_CHANGE** может принимать значение из диапазона [-50,50]
- объявляется составной первичный ключ, состоящий из трех полей: **EMP_NO, CHANGE_DATE, UPDATER_ID.**
- описывается внешний ключ (**FOREIGN KEY**) - поле **EMP_NO** ссылается (**REFERENCES**) на поле **EMP_NO** таблицы **EMPLOYEE**, обновление и удаление из таблицы EMPLOYEE каскадируется.