1 Действия с таблицами

1.1 Объединение таблиц

Предположим, мы хотим напечатать список пригласительных билетов на вечер, куда приглашаются и преподаватели, и студенты. Сложность создания такого списка в том, что фамилии преподавателей и студентов хранятся в разных таблицах, а в результате они должны составить один столбец таблицы приглашенных. Для выполнения этой задачи лучше всего использовать инструкцию объединения `union`

```
(select ...)
  union
(select ... )
  union
...
```

Все операторы `select`, участвующие в инструкции `union`, должны генерировать таблицы с одинаковым списком столбцов (с одинаковыми именами и типами столбцов). При этом следует иметь ввиду, что инструкция `union` неявно применяет `distinct` к получаемой таблице, то есть она отбрасывает повторяющиеся строки. Например, если мы напишем

```
(select l_name from student)
  union
(select prof as l_name from professor)
```

То получим:

l_name
Волушкова
Дудаков
Иванов
Климок
Колдунов
Сидоров
Сорокин
Федоров
Хижняк

Получилось не совсем то, что нужно, так как вместо трех Сорокиных образовалась одна строка. Чтобы `union` не отбрасывал повторения нужно к нему добавлять слово `all`:

```
(select l_name from student)
  union all
(select prof as l_name from professor)
```

Тогда получается

l_name
Иванов
Иванов
Сидоров
Сорокин
Федоров
Сорокин
Колдунов
Колдунов
Климок
Хижняк
Дудаков
Волушкова
Дудаков
Сорокин

Теперь повторяются фамилии преподавателей. Чтобы этого избежать, как мы уже говорили следует использовать слово distinct в запросе, выбирающем фамилии преподавателей:

```
(select l_name from student)
  union all
(select distinct prof as l_name  from professor)
```

l_name
Иванов
Иванов
Сидоров
Сорокин
Федоров
Сорокин
Волушкова
Дудаков
Климок
Колдунов
Сорокин
Хижняк

Теперь все правильно.

Для удобства иногда желательно отсортировать результат. Если с помощью `union` или `union all` несколько запросов соединяются в один, то инструкция сортировки `order by` ставиться в самом конце, чтобы отсортировать все результаты объединения:

```
(select ... )
  union [all]
(select ... )
...
  union [all]
(select ... )
order by ...
```

Для нашего примера

```
(select l_name from student)
  union all
(select distinct prof as l_name  from professor)
order by l_name
```

Получим

l_name
Волушкова
Дудаков
Иванов
Иванов
Климок
Колдунов
Сидоров
Сорокин
Сорокин
Сорокин
Федоров
Хижняк

1.2 Пересечение и вычитание таблиц

Инструкция `union` выполняет объединение таблиц как множеств строк. Точно так же используются инструкции `intersect` и `except`, означающие пересечение и вычитание соответсвенно. Например, чтобы узнать, какие фамилии встречаются как среди студентов, так и среди преподавателей, можно выполнить запрос

```
(select l_name from student)
  intersect
(select prof as l_name from professor)
order by l_name
```

l_name
Сорокин

1.3 Представления

Представления — это запросы, которые хранятся в базе данных и используются как таблицы. Когда представление используется в качестве таблицы, то вместо него подставляется результат выполнения запроса. Для того чтобы отличать таблицы, реально хранящиеся в базе данных, от представлений реально хранящиеся таблицы называют основными.

Представление создается с помощью оператора `create view`:

```
create view <имя представления> as
  <SQL-запрос>
```

Например, представление, содержащее информацию о сдаче студентами экзамена по анализу:

```
create view analysis as
  select stud_nomer, dat, res from ball
  where dis = 'Анализ'
```

Напомним, что таблицы является неупорядоченной совокупностью строк, то же должно относиться и к представлениям: порядок следования строк в представлении не определен. Поэтому в операторе SQL, образующем представление, инструкции `order by` быть не должно. Другое важное условие на SQL-запрос: все его столбцы должны иметь имена. Если в представлении в качестве столбцов используются какие-то выражения, то они обязательно должны быть поименованы с помощью `as`.

Предположим, мы хотим создать представление для среднего балла каждого студента:

```
create view avg_ball as
  select stud_nomer, avg(res) as avg_res
  from ball
  group by stud_nomer
```

Теперь представление `avg_ball` можно использовать как таблицу, содержащую два столбца: `stud_nomer` и `avg_res`. На-

пример, чтобы вывести ее содержимое:

```
select *
  from avg_ball
```

stud_nomer	avg_res
010001	85
010002	66
011003	63
011004	63
011005	76

При изменении основной таблицы `ball` изменения автоматически отразятся и в представлении `avg_ball`. Представления можно использовать как часть соединения как с основными таблицами, так и друг с другом.

Другой пример использования представлений — двойная группировка данных. С помощью одного оператора `select` можно выполнить только одну группировку, например, найти средний балл для каждого студента. Если же требуется еще и среди этих средних баллов найти самый высокий, то для этого придется использовать представление, так как вложенные агрегатные функции язык SQL не допускает:

```
select max(avg_res)
  from avg_ball
```

Нельзя написать непосредственно:

```
select max(avg(res))
  from ball
  group by stud_nomer
```

Однако, SQL допускает в качестве исходных таблиц запроса использовать другие запросы. Тогда наш запрос можно переписать в виде:

```
select max(avg_res)
from (select avg(res) as avg_res
       from ball
       group by stud_nomer)
```

То есть, вместо имени представления в инструкции `from` просто пишется SQL запрос, реализующий это представление.

Если бы мы захотели переименовать столбцы при создании представления, то это можно было бы сделать следующим образом:

```
create view avg_ball(nomer, result) as
  select stud_nomer, avg(res) as avg_res
  from ball
  group by stud_nomer
```

2 Изменение данных.

2.1 Изменение данных основных таблиц.

Самая простая операция изменения данных — вставка новой строки:

```
insert into <имя таблицы>(<список полей>)
  values (<список значений>)
```

Например, чтобы добавить в таблицу `ball` новую строку, нужно записать следующее:

```
insert into ball(stud_nomer, dis, dat, form, res)
  values ('010001', 'Практикум на ЭВМ',
            '29.12.2003', 'Зачет', 85)
```

Если какой-либо столбец таблицы в списке полей не указан, то будет попытка вставить на его место значение по умолчанию, если оно есть, или `NULL`, если значение по умолчанию отсутствует.

Например, оператор

```
insert into ball(stud_nomer, dis, dat, res)
  values ('010001', 'Практикум на ЭВМ',
                '29.12.2003', 85)
```

приведет к вставке строки

010001	Практикум на ЭВМ	04.01.2004	Экзамен	100

Оператор

```
insert into ball(stud_nomer, dis)
  values ('010001', 'Практикум на ЭВМ')
```

вставит

010001	Практикум на ЭВМ	NULL	Экзамен	NULL

А оператор

```
insert into ball(stud_nomer)
  values ('010001')
```

приведет к ошибке и не выполнится, так как для поля `dis` значение по умолчанию не задано, а значение `NULL` это поле не допускает.

Более сложный оператор — удаление данных из таблицы:

```
delete from <имя таблицы>
  [where <условие>]
```

Если условие отсутствует, то удаляются все строки из таблицы. Если оно записано, то удаляются только строки, ему удовлетворяющие.

Например, чтобы удалить всю информацию из таблицы `ball` о студенте с номером '010001', следует написать:

```
delete from ball
  where stud_nomer = '010001'
```

Правила написания условия такие же, как в инструкции `where` оператора `select`.

Оператор изменения существующих строк:

```
update <имя таблицы>
 set <имя поля> = <значение>
   [,<имя поля> = <значение>
   [, ... ]]
 [where <условие>]
```

При выполнении оператора в указанные после слова `set` поля записываются соответствующие значения. Если присутствует инструкция `where`, то изменения касаются лишь строк, удовлетворяющих условию, иначе — всех строк таблицы.

Если, например, нужно изменить номер студента с '010001' на '010010', то следует изменить таблицы `student` и `ball`:

```
update student
  set nomer = '010010'
  where nomer='010001'

update ball
  set stud_nomer = '010010'
  where stud_nomer='010001'
```

В качестве новых значений для полей могут использоваться выражения с участием полей. Если нужно увеличить все баллы за экзамен по информатике на 5, то это можно сделать так:

```
update ball
  set res = res + 5
  where dis = 'Информатика' and form = 'Экзамен'
```

Как и в операторе `select`, в инструкциях `where` операторов `delete` и `update` можно использовать подзапросы. Например, чтобы студентам, набравшим наибольшее число баллов на экзамене по информатике, установить результат в 100 баллов, можно выполнить такой оператор:

```
update ball
  set res = 100
  where dis = 'Информатика' and
    form = 'Экзамен' and res = any(
      select max(res)
        from ball
        where dis = 'Информатика' and
          form = 'Экзамен')
```

2.2 Изменение данных представлений.

Гарантируется, что представления, построенные с помощью SQL-запросов к одной основной таблице и без использования группировки, можно изменять, подобно основной таблице. Это легко объяснимо — такие представления просто являются частью основной таблицы и изменения, вносимые в эти представления, легко преобразуются в изменения основной таблицы.

Например, представление `analisys` обязательно является обновляемым. Можно, например, выполнить оператор удаления данных:

```
delete from analysis
  where stud_nomer = '010001'
```

или оператор изменения:

```
update analysis
  set dat = '07.01.2004'
```

Первый из этих операторов просто удалит из основной таблицы `ball` строки, удовлетворяющие указанному условию, а второй — изменит поле `dat` на '07.01.2004' во всех строках таблицы `ball`, которые попали в представление `analysis`.

Более сложная ситуация с оператором `insert`. Для его использования могут возникнуть два препятствия. Первое из них: представление содержит не все столбцы исходной таблицы. В частности, наше представление `analysis` содержит 3 из 5 столбцов таблицы `ball`. Поэтому, если попытаться выполнить оператор вставки

```
insert into analysis(stud_nomer, dat, res)
  values ('011101', '04.01.2004', 60)
```

то возникнут проблемы с заполнением полей `dis` и `form` таблицы `ball`. Ситуация разрешается с помощью записи в такие поля значения по умолчанию или `NULL`, если значения по умолчанию не определены. В нашем случае для `form` будет использоваться значение по умолчанию ('Экзамен'), а для `dis` — `NULL`. Но так как поле `dis` не допускает значений `NULL`, то этот оператор вставки приведет к ошибке и не будет выполнен.

Другой подводный камень добавления строк в представлении — появление строк, не удовлетворяющих условиям представления. Предположим, что представление `analysis` определено так:

```
create view analysis as
  select stud_nomer, dis, dat, res
    from ball
    where dis = 'Анализ'
```

Если теперь в это представления вставить строку, для которой поле `dis` будет иметь значение 'Алгебра':

```
insert into analysis(stud_nomer, dis, dat, res)
  values('011101', 'Алгебра', '15.01.2004', 60)
```

то формально никакой ошибки не будет, эта строка вставиться в таблицу `ball`. Но при повторном выборе данных из представления `analysis` эта строка не появится, так как не удовлетворяет условию представления. Получается ситуация, когда мы вроде бы вставляем данные, но не видим их. То же самое может случиться при использовании оператора `update`: можно изменить данные таким образом, что они перестанут удовлетворять условию представления и окажутся как бы удаленными из него.

Если нужно избежать таких проблем, представление создается с параметром `with check option`:

```
create view as ...
  with check option
```

Например,

```
create view analysis as
  select stud_nomer, dis, dat, res from ball
  where dis = 'Анализ'
  with check option
```

Представления, созданные таким образом, не допускают вставку данных, не удовлетворяющих условию представления. То есть, приведенный выше оператор `insert` для последнего представления вызовет ошибку и не вставит новой строки. Точно так же через представление `analysis` нельзя произвести изменение существующих строк, в результате которого строки перестанут удовлетворять условиям представления. Например, оператор

```
update analysis
  set dis = 'Алгебра'
```

вызовет ошибку.

3 Вторичные ключи

3.1 Параметр Unique

Этот параметр можно использовать точно так же, как и параметр PRIMARY KEY. Он позволяет требовать, чтобы в различных строчках значения полей с параметром UNIQUE были разные.

Например,

```
create table student(
  l_name name_type,
  f_name name_type,
  m_name name_type,
  nomer nomer_type primary key,
  gr_nomer group_type)

create table ball(
  stud_nomer nomer_type,
  dis dis_type,
  dat date,
  form form_type,
  res result_type,
  primary key(stud_nomer, dis, dat))

create table professor(
  dis_name dis_type,
  gr group_type,
  prof name_type,
  primary key(dis_name, gr),
  unique (prof, dis_name))
```

Главные отличия состоят в следующем:

может быть только один первичный ключ, но сколько угодно групп полей с параметром UNIQUE;

в то время, как поля первичного ключа все должны иметь

настоящие значения, поля с параметром UNIQUE могут принимать значение NULL.

При этом в нескольких различных строчках поле с параметром UNIQUE может принимать значение NULL.

3.2 Посторонние ключи

Важное значение имеет для базы данных её целостность. В частности, например, значение поля stud_nomer в таблице ball должно быть реально существующим номером студента.

При этом список полей, на который ссылаются (в нашем примере, nomer в таблице student) должен быть объявлен либо как primary key, либо как unique. Только для таких списков можно создавать посторонние ключи.

Любое значение списка полей со ссылкой в первой таблице должно встречаться как значение списка полей, на который ссылаются, во второй таблице. Кроме случая, когда в рассматриваемом значении списка полей первой таблице встречается значение NULL.

Имеется два способа задания посторонних ключей.

Первый способ можно применять только в случае, когда список состоит из одного атрибута.

REFERENCES <таблица>(<атрибут>)

Пример:

```
create table student(
  l_name name_type,
  f_name name_type,
  m_name name_type,
  nomer nomer_type primary key,
  gr_nomer group_type)

create table ball(
  stud_nomer nomer_type REFERENCES student(nomer),
  dis dis_type,
  dat date,
  form form_type,
  res result_type,
  primary key(stud_nomer, dis, dat))

create table professor(
  dis_name dis_type,
  gr group_type,
  prof name_type,
  primary key(dis_name, gr),
  unique (prof, dis_name))
```

Второй способ состоит в добавлении отдельного объявления о постороннем ключе.

FOREIGN KEY (<СПИСОК АТРИБУТОВ>) REFERENCES <таблица>(<список атрибутов>)

Пример:

```
create table student(
  l_name name_type,
  f_name name_type,
  m_name name_type,
  nomer nomer_type primary key,
  gr_nomer group_type)

create table ball(
  stud_nomer nomer_type,
  dis dis_type,
  dat date,
  form form_type,
  res result_type,
  primary key(stud_nomer, dis, dat),
  FOREIGN KEY (stud_nomer) REFERENCES student(nomer))

create table professor(
  dis_name dis_type,
  gr group_type,
  prof name_type,
  primary key(dis_name, gr),
  unique (prof, dis_name))
```

Более сложная ситуация здесь складывается с полем dis или dis_name. Ни одно из этих полей нельзя объявить посторонним ключом, так как ни одно из этих полей не является первичным или вторичным ключом.

Обычные действия системы с посторонними ключами при изменениях таблиц мы поясним на нашем примере.

Если некоторый студент отчисляется и сведения о нем удаляются из таблицы student, то система это запрещает до тех пор, пока записи о результатах проверки знаний этого студента не будут удалены из таблицы ball.

Если вставляются сведения в таблицу ball о результатах контроля знаний студента, которого нет в таблице student, то

система это запрещает. То же самое, если пытаются изменить в некоторой записи таблицы ball номер студента на другой номер, а этот другой номер не существует в таблице student.

Если пытаются изменить номер студента в таблице student, а этот изменяемый номер уже встречается в таблице ball, то система это запрещает.

Другой подход может состоять в том, что удаляя запись о студенте с некоторым номером, одновременно удаляются и все результаты экзаменов и зачётов этого студента. Аналогично, если меняется номер студента, то меняются и его номера во всех записях о результатах экзаменов и зачётов этого студента. Это — так называемая каскадная стратегия.