Таблиці

CREATE TABLE — для створення таблиці
ALTER TABLE — для модифікації таблиці
DROP TABLE — для її знищення

Найбільш простий варіант команди створення таблиці:

```
CREATE TABLE    таблиця
      (поле     тип_поля        [DEFAULT значення][,
       поле     тип_поля        [DEFAULT значення][,…]]);
```

Приклад:

```
CREATE TABLE    Student
    (SecondName     CHAR(30),
     FirstName      CHAR(20),
     Patronymic     CHAR(30),
     SNum   SMALLINT,
     Spec   CHAR(2)     DEFAULT 'AI',
     GNam   INTEGER,
     Address CHAR(30),
     Email  CHAR(30));
```

Обмеження

– на певні стовпці або просто *на стовпці* і на підмножину стовпців:

вказуються у визначенні відповідного поля і мають позиційне значення тільки в тім розумінні, що розміщуються після оголошення типу:

поле тип_поля обмеження_С

– *на таблицю*:

обмеження на таблицю записуються після визначення останнього поля і відокремлюються від нього комою

```
CREATE TABLE    таблиця
      (поле     тип_поля      [обмеження_C][,
       поле     тип_поля      [обмеження_C][,…]],
       обмеження_T);
```

– ***обмеження на заборону пропуску значення*** відповідного атрибута при додаванні кортежу (рядка): оператор NOT NULL. Наприклад:

```
…   SecondName     char(30)     NOT NULL,
    FirstName      char(20)     NOT NULL, …
```

Цей варіант обмеження не використовується як обмеження на таблицю, хоча при відсутності значення відповідного поля *буде відхилена вся операція* відновлення.

– ***обмеження за значенням*****:** оператор CHECK. Наприклад, може виконуватися перевірка значення на відповідність діапазону:

```
CREATE TABLE    Rating
      (Kod     INTEGER     CHECK(Kod > 0)   NOT NULL,
       DKod    INTEGER     CHECK(Kod > 0)   NOT NULL,
       Mark    INTEGER     DEFAULT 0        CHECK(Mark >= 0 AND Mark <= 100),
       MDate   DATE        NOT NULL);
```

– перевірка на належність значення заданій множині:

```
… Spec     char(2)     CHECK(Spec IN('АП', 'ОС', 'АМ', 'АС', 'ОМ', 'ОІ')), …
```

Обмеження за значенням поширюється на обмеження на таблицю.
Наприклад:

```
CREATE TABLE    Rating
      (Kod        INTEGER        NOT NULL,
       DKod       INTEGER        NOT NULL,
       Mark       INTEGER        DEFAULT 0,
       CHECK(Kod > 0 AND Dkod > 0 AND (Mark BETWEEN 0 AND 100)) ,
       MDate      DATE    NOT NULL);
```

Обмеження за значенням відхиляють операцію відновлення, якщо значення атрибутів кортежу не відповідають умовам обмеження.

Логічні оператори

Результатом дії буде значення „істина“ або „неправда“. До них відносяться оператори порівняння, булеві оператори і спеціальні оператори.

Оператори порівняння: =, >, <, >=, <=, <>.

Булеві оператори в *SQL* реалізують 3 операції булевої алгебри: AND, OR, NOT.

Спеціальних операторів у *SQL* сім: IN, BETWEEN, LIKE, IS NULL, EXISTS, ANY (SOME) та ALL. Проте три останні оператори використовуються ***тільки*** з *вкладеними підзапитами*.

Оператор **IN** цілком визначає *деяку множину значень*. Наприклад:

```
… Spec    CHAR(2)     CHECK(Spec IN(‘АП’, ‘AM’, ‘AC’, ‘OM’, ‘OI’, ‘OC’)), ...
```

Аналогом цього була б досить складна конструкція типу:

```
… Spec    CHAR(2)     CHECK(Spec = ‘АП’ OR Spec = ‘AM’ OR … OR Spec = ‘OC’), …
```

Оператор **BETWEEN** разом з оператором AND задає *діапазон значень*. Причому границі діапазону *входять* у число припустимих значень і можуть бути як числами, так і рядками ASCII-символів. В останньому випадку при порівнянні рядків різної довжини більш короткий рядок *доповнюється* пробілами, що мають найменший ASCII-код серед символів алфавіту. Тому інструкція вигляду

```
… SecondName    char(30)   CHECK(SecondName BETWEEN ‘A’ AND ‘МАЛИЙ’), …
```

дозволить увести будь-яке прізвище від А до М, точніше, до МАЛИЙ, але не дозволить, наприклад, МАЛАХОВ. Якщо необхідно *виключити границі* діапазону з числа припустимих значень, можна скористатися інструкцією такого вигляду:

```
… Mark          INTEGER      CHECK((Mark BETWEEN 0 AND 100)
                                        AND NOT Mark IN(0,100))…
```

Оператор **LIKE** застосовуваний *тільки до символьних полів* типу CHAR і VARCHAR і використовується для накладення *шаблонів* на рядки.

Для цього в операторі використовуються спеціальні **символи-шаблони**: символ „підкреслення“ (‘_’), що заміняє *один будь-який* символ, і символ „відсоток“ (‘%’), що заміняє символьний *рядок довільної довжини*.

Наприклад, шаблону „*д_м*“ відповідають рядки „*дам*“, „*дим*“, „*дум*“ тощо, а шаблону „*%М%В*“ в полі прізвища

```
… SecondName      char(30)     CHECK(SecondName LIKE ‘%M%B’), …
```

відповідають і **М**АЛАХО**В**, і ЛО**М**ОНОСО**В**, але не відповідає, наприклад, **М**АЛИНО**В***СЬКИЙ*.

Для того, щоб у шаблоні оператора LIKE використовувати і самі символи ‘_’ та ‘%’ необхідно будь-який символ, наприклад слеш ‘/’, визначити як *Escape-символ* і випереджати їм кожний з керуючих, у тому числі і самого себе:

```
… SpName        char(60)   CHECK(SpName LIKE ‘%/_%/%%//%’ ESCAPE ‘/’), …
```

У цьому прикладі команда CHECK пропустить будь-які символьні рядки, в яких у будь-якому місці, але послідовно зустрічаються символи ‘_’, ‘%’ і ‘/’. Наприклад:

„У спец_фонд виділити 15**%** доходу, але не більш 1000 грн/місяць“

Оператор **IS NULL** використовується для визначення кортежів, у яких *відсутні значення* тих чи інших атрибутів. Наприклад:

```
… Patronymic IS NULL …
```

відповідає кортежам, у яких відсутнє по-батькові. Зворотний йому оператор IS NOT NULL дозволяє *відсіяти відсутні значення*.

Регулярні вирази.

Регулярні вирази застосовують для:
— пошуку у рядку підрядка, який задовольняє шаблону регулярного виразу;
— пошуку та заміни у рядку підрядка, який задовольняє шаблону регулярного виразу;
— перевірки на відповідність заданого рядка шаблону;
— добування з рядка підрядка, який задовольняє шаблону регулярного виразу.

В стандарт *SQL*'99 було додано можливість використання регулярних виразів через оператор SIMILAR TO як розвиток оператора LIKE:
— рядок SIMILAR TO *шаблон*;
— рядок NOT SIMILAR TO *шаблон*;

Шаблон оператора SIMILAR TO крім символів '%', '_' використовує додаткові символи:
— символ '|' позначає альтернативи елементів;
— символ '?' позначає повторення попереднього елемента 0 або 1 раз;
— символ '*' позначає повторення попереднього елемента 0 або більше раз;
— символ '+' позначає повторення попереднього елемента 1 або більше раз;
— символи '{*m*}' позначають повторення попереднього елемента рівно *m* раз;
— символи '{*m*,}' позначають повторення попереднього елемента *m* або більше раз;
— символи '{*m*, *n*}' позначають повторення попереднього елемента від *m* до *n* раз;
— символи '( )' групують елементи в один логічний блок;
— символи '[ ]' визначають клас символів через перерахування припустимих символів або з використанням символу діапазону '-';

Якщо вказані символи необхідно використовувати в шаблоні як звичайні, перед ними використовують символ '\'. В шаблоні всі символи '\' потрібно також дублювати, тобто писати як '\\'.

```
SELECT * FROM Student WHERE (Address).Street NOT SIMILAR TO ‘%[0-9]+%’;
```

```
SELECT    *   FROM   Student    WHER    Email    SIMILAR TO ‘%[a-z]{3,4}[0-9]{5}%’
```

```
SELECT    *   FROM   Letters   WHERE     Content      SIMILAR TO ‘%[0-9]?.[0-9]+%’
```

```
SELECT * FROM Letters
        WHERE Content SIMILAR TO ‘%(1[012]|[1-9]):[0-5][0-9]? +(am|pm)%’
```

```
SELECT * FROM Letters
        WHERE Content SIMILAR TO ‘%([01]?[0-9]|2[0-3]):[0-5][0-9]%’
```

```
SELECT * FROM Letters
    WHERE Content SIMILAR TO ‘%\\([0-9]{3,5}\\)[ ]+[0-9]{3}\\-[0-9]{2}\\-[0-9]{2}%’
```

```
 ‘%\\+38\\((050|063|067|093)\\) +[0-9]{3}\\-[0-9]{2}\\-[0-9]{2}%’
```

```
SELECT * FROM Letters
        WHERE Content SIMILAR TO ‘%[a-z0-9._%-]+@[a-z0-9._%-]+\.[a-z]{2,4}%’
```

СУБД PostgreSQL також підтримує ***регулярні вирази через наступні оператори***:

— ‘~’ пошук за шаблоном, який чутливий до регістру символів;
— ‘~ *’ пошук за шаблоном, який нечутливий до регістру символів;
— ‘! ~’ пошук по виключаючому шаблону, нечутливому до регістру символів;
— ‘! ~ *’ пошук по виключаючому шаблону, нечутливому до регістру символів.

На відміну від оператора SIMILAR TO, вони повертають істину, якщо шаблон регулярного виразу відповідає *будь-якій частині рядка*, тому початковий та кінцевий символ ‘%’ в шаблоні використовувати нема потреби.

Для добування підрядку, що задовольняє шаблону регулярного виразу, з рядка використовується функція SUBSTRING(*string from pattern*).

Якщо в шаблоні присутні *круглі дужки*, то повертається підрядок по шаблону усередині дужок.

Наприклад, у результаті виконання функції

```
SUBSTRING(‘ E-mail my@ukr.net ‘ FROM ‘[a-z0-9._%-]+@[a-z0-9._%-]+\.[a-z]{2,4}’);
```

буде виділений підрядок „*my@ukr.net*“.

Якщо ж у шаблон включити круглі дужки, то функція

```
SUBSTRING(‘ E-mail my@ukr.net ‘ FROM ‘([a-z0-9._%-]+)@[a-z0-9._%-]+\.[a-z]{2,4}’)
```

виділить підрядок „*my*“.

Для пошуку та заміни в рядку *source* підрядку, що задовольняє шаблону регулярного виразу *pattern* у рядку, на підрядок *replacement* використовується функція

```
REGEXP_REPLACE(source, pattern, replacement [, flags]).
```

Додаткові функції REGEXP_MATCHES, REGEXP_SPLIT_TO_TABLE і REGEXP_SPLIT_TO_ARRAY використовують регулярні вирази для формування з рядка множини підрядків.

Функція REGEXP_MATCHES(*string, pattern* [, *flags*]) повертає з рядка *string* масив підрядків відповідно до шаблону *pattern*. Якщо шаблон містить дужкові підвирази, функція повертає масив, у якому *n*-й елемент визначений як результат відповідності рядка з *n*-м дужковим виразом. Значення опції *flags* еквівалентні функції REGEXP_REPLACE.

Функція REGEXP_SPLIT_TO_TABLE(*string, pattern* [, *flags*]) повертає з рядка *string* множину підрядків відповідно до шаблона *pattern* у вигляді роздільника між підрядками. Функція REGEXP_SPLIT_TO_ARRAY має той же синтаксис, але повертає масив підрядків.

Потенційні ключі

На етапі побудови БД використовується таке визначення потенційного ключа:

UNIQUE [(*список_атрибутів*)]

Причому це визначення може використовуватися і як обмеження на стовпець, і як обмеження на таблицю. Наприклад, у таблиці Student

… SNum SMALLINT UNIQUE, …

Тут є *дві некоректності*. *Перша* полягає в тому, що, як було відзначено раніше, потенційний ключ не може містити атрибутів, що допускають відсутність значень. Отже, необхідно додати NOT NULL. Але це не скасовує *другу* некоректність: номер студента за списком унікальний для даної групи, але може повторюватися в іншій групі і, тим більше, на іншій спеціальності. Тому номер залікової книжки складається із шифру спеціальності, номера групи і номера студента:

```
… SNum  SMALLINT   NOT NULL,
  Spec  CHAR(2)    NOT NULL   CHECK(Spec IN('OI', 'OM', 'OП', 'OC')),
  GNum  INT        NOT NULL,
  UNIQUE(Spec, GNum, SNum));
```

Первинний ключ

Для цього в *SQL* існує оператор PRIMARY KEY:

PRIMARY KEY [(*список_атрибутів*)]

Змінимо попередній приклад:

… PRIMARY KEY (Spec, GNum, SNum));

Це приклад використання первинного ключа як *обмеження на таблицю*. Іноді більш доцільно використовувати *одноатрибутний* привілейований ідентифікатор, тобто використовувати первинний ключ як *обмеження на стовпець*. Наприклад, у таблицю Student можна додати поле

… Kod INT NOT NULL PRIMARY KEY, …

яке забезпечує наскрізну нумерацію по університету або містить, наприклад, податковий індекс (унікальність у межах держави).

Зовнішні ключі

Як обмеження на таблицю *SQL* становить такий синтаксис для зовнішнього ключа:

FOREIGN KEY *список_атрибутів* REFERENCES *базова_таблиця* [*список_атрибутів*]

Наприклад.

```
CREATE TABLE    Student
      (Kod            INT     NOT NULL    PRIMARY KEY     CHECK(Kod>0),
       SecondName     CHAR(30)    NOT NULL,
       FirstName      CHAR(20)    NOT NULL,
       Patronymic     CHAR(30),
       SNum           SMALLINT    NOT NULL,
       Spec           CHAR(2)     NOT NULL    CHECK (Spec IN('ОІ','ОМ','ОС','ОП')),
       GNum           INT     NOT NULL,
       Address        CHAR(30),
       Email          CHAR(30),
       FOREIGN KEY (Spec, GNum)        REFERENCES Decanat (Spec, GNum));
```

Якщо в зовнішньому ключі присутній *тільки один* атрибут, то синтаксис буде трохи інший. Створимо ще одну таблицю для прикладу:

```
CREATE TABLE    Speciality
      (SpKod     CHAR(11)     NOT NULL    CHECK(SpKod LIKE '_ . _ _ _ _ _ _ _ _'),
       ScDirect  CHAR(20)     NOT NULL,
       SpName    CHAR(40)     NOT NULL    UNIQUE,
       Spec      CHAR(2)      NOT NULL    UNIQUE
       CHECK(Spec IN('ОІ', …, 'ОС')));
```

Тоді в таблиці Student можна змінити інструкцію

```
…    Spec    CHAR(2)     REFERENCES Speciality (Spec), …
```

Якщо батьківським є первинний ключ, наприклад, у таблиці *Speciality* замість *SpKod — Spec*, то у визначенні зовнішнього ключа можна обійтися *без імен атрибутів*:

```
…    Spec    CHAR(2)     REFERENCES Speciality, …
```

Використання NULL-значень у зовнішніх ключах для дотримання принципу посилальної цілісності є не тільки бажаним, але й необхідним у тому випадку, якщо *зовнішній ключ посилається на первинний ключ* ***власної*** *таблиці*:

```
CREATE TABLE    Employees
      (EmpNo    INT             NOT NULL      PRIMARY KEY,
       Name     CHAR(30)        NOT NULL      UNIQUE,
       Manager  INT             REFERENCES Employees);
```

У *SQL* для зміни або видалення кортежів з батьківським ключем існує три можливості:

1. Можна обмежити або заборонити модифікацію батьківського ключа:
   RESTRICTED (*обмеження*) або NO ACTION (у *SQL*\92);
2. Можна при зміні батьківського ключа автоматично змінювати і зовнішні ключі, що посилаються на нього:
   CASCADE[S] (*каскадно*), (без S у *SQL*\92);
3. Можна змінювати або видаляти кортеж з батьківським ключем, а значення відповідних йому зовнішніх ключів автоматично встановлюються (*і тут цей випадок поділяється ще на два*):
   —у NULL:
   NULLS або SET NULL (*SQL*\92);
   —у значення за замовчуванням:
   SET DEFAULT.

Наприклад, внесемо зміни в таблицю Student.

```
CREATE TABLE    Student
      (Kod              INT     NOT NULL      PRIMARY KEY      CHECK(Kod>0),
       SecondName       CHAR(30)        NOT NULL,
       FirstName        CHAR(20)        NOT NULL,
       Patronymic       CHAR(30),
       SNum             SMALLINT        NOT NULL,
       Spec             CHAR(2)         REFERENCES Speciality (Spec)   NOT NULL,
       GNum             INT     NOT NULL,
       Address          CHAR(30),
       Email            CHAR(30),
       UNIQUE(Spec, GNum, SNum),
       FOREIGN KEY (Spec, GNum)     REFERENCES Decanat (Spec, GNum)
       ON UPDATE CASCADE
       ON DELETE CASCADE,
       UPDATE OF Speciality CASCADES,
       DELETE OF Speciality RESTRICTED);
```

В діалекті ***PostgreSQL*** обмеження на стовпці типу UPDATE OF, DELETE OF тощо відсутні.

Їхню роль відіграють інструкції ON UPDATE або ON DELETE, які вказуються в якості обмежень на відповідні стовпці разом з інструкцією REFERENCES.

Для модифікації в *SQL* існує команда

```
ALTER TABLE    таблиця     операція   об'єкт[,
               таблиця     операція   об'єкт[, …]];
```

де *таблиця* — ім'я таблиці, *операція* — ADD (*додати*) або DROP (*видалити*), *об'єкт* — стовпець (при додаванні указуються всі необхідні параметри) або обмеження типу UNIQUE, PRIMARY KEY, CHECK і т.д.

Наприклад:

```
ALTER TABLE  Rating
      DROP   Mark,
      ADD    MarkECTS    CHAR(2)
                         CHECK(MarkECTS      IN('A', 'B', 'C', 'D', 'E', 'FX', 'F'))
                         DEFAULT 'F';
```

Щодо *додавання* та *видалення обмежень* необхідно звернути увагу на те, що для виконання цих дій обмеження повинні мати імена (назви).

Знищується таблиця командою

```
DROP TABLE    таблиця      опція;
```

Опціями може бути RESTRICT або CASCADE. Якщо при видаленні таблиці встановлюється опція RESTRICT, а сама таблиця містить батьківські ключі, на які *існують посилання*, то команда буде *відкинута*. Якщо ж у такій ситуації встановлена опція CASCADE, то команда буде виконана, а всі *посилання* на неї будуть *знищені* (замінені NULL або DEFAULT).

Створення об’єктно-реляційних зв'язків в PostgreSQL

При складанні запиту до таблиці-предка можна вимагати, щоб запит справив вибірку або тільки з самої таблиці, або ж переглянув таблицю з її таблицями-нащадками. При запиті до таблиці-нащадка у результат не включаються рядки з таблиці-предка.

При створенні таблиці-нащадка на основі таблиці-предка використовується оператор INHERITS:

CREATE TABLE *таблиця-нащадок* (…)
INHERITS (*таблиця-предок* [*атрибут*_1, *атрибут*_2, …])

Розглянемо **приклад**. Нехай існує таблиця *Student.*

Таблиця *Master* з даними про студентів-дипломників може успадковувати всі атрибути таблиці *Student*, доповнюючи назвою дипломної роботи:

```
CREATE TABLE    Master
    (Diploma    VARCHAR)
     INHERITS (Student);
```