**1.Объектно-реляционные СУБД**

Проблемы объектных СУБД, которые не получили широкого использования, имели следующие характеристики:

* В объектно-ориентированных базах данных, в отличие от реляционных, хранятся не записи, а объекты. Это соответствует естественному представлению данных;
* имеется возможность определения новых типов данных и операций с ними.

Однако имеются недостатки:

* отсутствуют мощные непроцедурные средства извлечения объектов из базы. Все запросы приходится писать на процедурных языках, проблема их оптимизации возлагается на программиста.
* вместо чисто декларативных ограничений целостности (типа явного объявления первичных и внешних ключей реляционных таблиц с помощью ключевых слов PRIMARY KEY и REFERENCES) или полудекларативных триггеров для обеспечения внутренней целостности приходится писать процедурный код.

Таким образом, ответственность за организацию целостности и эффективного поиска полностью переносится на программиста.

Другой способ объединения возможностей реляционного и объектно-ориентированного подхода к управлению данными предложил известный американский ученый [**Майкл Стоунбрейкер**](http://s2k-ftp.cs.berkeley.edu:8000/nasa_e2e/mike.html%22). Согласно его воззрениям реляционную СУБД нужно просто дополнить средствами доступа к сложным данным. При этом ядро СУБД не требует переработки и сохраняет все присущие реляционным системам достоинства. Объектные расширения реализуются в виде надстроек, которые динамически подключаются к ядру. На основе этой идеи под руководством М.Стоунбрейкера в университете Беркли (Калифорния, США) была разработана объектно-реляционная **СУБД Postgres**, которая имеет следующие ключевые возможности:

* Типы, операторы и методы доступа, определяемые пользователем;
* Поддержка сложных объектов, представляющих собой наборы других объектов.
* Перегрузка операторов манипулирования данными.
* Создание функций, определяемых пользователем.
* Динамическое (т.е. без прерывания работы СУБД) добавление новых типов данных, операторов, функций и методов доступа. Описание всех этих возможностей создается на языке C и компилируется в объектный файл, который может динамически загружаться сервером СУБД.
* Наследование данных и функций.
* Использование массивов как значений полей кортежей.

**Общие приемы объектно-ориентированных возможностей работы с PostgreSQL**

Основные объектно-ориентированные возможности связаны с использованием принципов инкапсуляции, наследования и полиморфизма. Каждый из этих принципов нашел свое отражение в рамках PostgreSQL.

1. Инкапсуляция. Основная цель заключается в возможности хранения объекта как единого целого, даже если он состоит из сложных объектов-частей. Эти возможности характеризуются созданием собственных типов данных и возможностью хранения в записях атрибутов-массивов.

Например, создадим тип для хранения адреса (организации или человека):

Create type Address as (

"number" integer,

"street" text,

"city" text

);

Далее этот тип можно использовать для создания, например, столбцов таблиц:

CREATE TABLE "Person"

(

id oid,

"FirstName" text,

"LastName" text,

"Birth" date,

"Address" address

)

При вставке значений данные сложного типа объединяются с помощью круглых скобок:

insert into "Person" ("FirstName","LastName","Birth","Address") values ('Иван', 'Иванов','23.01.1998',(5,'Восстания','Казань'))

К столбцу с нестандартным типом данных, естественно, можно обращаться, но SQL не дает возможности стандартно обратиться к его части. В таком запросе будет ошибка:

select "Address"."number" from "Person"

Проблема решается за счет использования пользовательских функций:

create function getStreet(Address) returns text

as $$

begin

return $1."street";

end;

$$ language plpgsql;

select getStreet("Address") from "Person"

select "FirstName","LastName" from "Person" where getStreet("Address")='Журналистов'

Изменение данных:

update "Person" set "Address"=(13,'Вишневского','Казань') where id=3

Для изменения части данных определенного пользователем типа можно использовать функцию:

create function createAddress (address,text) returns address

as $$

declare adr address;

begin

adr.number:=$1.number;

adr.street:=$2;

adr.city=$1.city;

return adr;

end;

$$ language plpgsql;

Соответствующая команда обновления:

update "Person" set "Address"=createAddress("Address",'пр.Амирхана') where "person\_id"=1

Таблица может содержать массив, например, массив телефонов:

alter table "Person" add column phones text[]

update "Person" set "phones"='{"89275432190","2517621"}' where id=1

update "Person" set "phones"='{"89176509821"}' where id=2

update "Person" set "phones"='{"89035621789","5235188","2776534"}' where id=3

select "FirstName","LastName", "phones"[2] from "Person" where "phones"[2] is not null

Интервал задает срез элементов массива, который необходим:

select "FirstName","LastName", "phones"[1:2] from "Person" where "phones"[2] is not null

Узнать размеры массива можно с помощью специальных встроенных функций:

select "FirstName","LastName", array\_dims("phones") as count\_phones from "Person"

Массивы являются динамическими и позднее можно с помощью команды update изменять и добавлять элементы в него. Если указывается несуществующий номер элемента, размер массива увеличивается до указанных пределов. Например, в массиве было 2 элемента. При добавлении 5-ого элемента будут созданы также 3-й и 4-й элементы, значениями которых будет NULL.

1. Наследование. Как известно, наследование является одним из важных принципов объектно-ориентированного программирования, который позволят создавать новые классы, используя существующие свойства родительского класса. Модель баз данных «Сущность-Связь» также подразумевала существования специального типа связи – отношение «подтипов». В реляционных моделях эту связь реализовывали с помощью разбиения записи на две части, которые хранились в двух таблицах с поддержкой связи «1:1».

Для примера создадим таблицу «Студенты», которая является производной от таблицы «Person».

CREATE TABLE "Student" ("num\_group" text, "reyting" integer) inherits ("Person")

insert into "Student" values (4,'Андрей', 'Андреев','13.07.1998',(6,'Декабристов','Казань'),'{"89034511134"}','09-210',100)

Добавление данных осуществляется только в производную таблицу, однако при запросе к базовой таблице мы увидим также и данные производных таблиц. Если эти данные не нужны, добавляется ключевое слово ONLY:

select \* from only "Person"

Ключи и другие поля с ограничениями уникальности могут пересекаться в двух таблицах – это может привести к логическим противоречиям.

Следует помнить, что при модификации данных в базовой таблице, этот же запрос на изменение будет проводиться для производных таблиц:

update "Person" set "phones"='{"111111111"}' where id=3

Так, если в обоих таблицах существовали записи с id=3, изменение телефонов будет произведено в обеих записях.

Интересно, что при добавлении нового столбца в родительскую таблицу после создания и работы с дочерней таблицей, новый столбец появится и во всех дочерних таблицах, т.е. связь между схемами таблиц поддерживается все время.

Стоит отметить возможность множественного наследования таблиц. В этом случае одноименные (при условии однотипности) столбцы, которые могут оказаться сразу в нескольких родительских таблицах будут соединены в один в дочерней таблице. Поскольку наследование касается только структуры, а не хранимых данных, это к противоречиям не приводит. Естественно, если одноименные столбцы родительских таблиц будут иметь разные типы данных, наследование не будет возможным.

1. Полиморфизм. Полиморфизм часто связывают с разным содержанием поведения, закрепленное за конкретным именем. Здесь нет такого четкого полиморфизма, как в объектно-ориентированных языках. Однако элементы закрепления отдельных форм за различными алгоритмами, в принципе, имеются.

Один из популярных вариантов полиморфизма – создание пользовательских операторов. Здесь можно создавать операторы, комбинируя символы +,-,\*,/,<,>,=,~,!,@,#,%,^,&,|,?,$.

Например, закрепим за обозначением ## поиск максимума из двух чисел:

create function MaxInt(integer, integer) returns integer

as $$

begin

if $1<$2 then

return $2;

else

return $1;

end if;

end;

$$ language plpgsql;

create function MaxDouble(numeric, numeric) returns numeric

as $$

begin

if $1<$2 then

return $2;

else

return $1;

end if;

end;

$$ language plpgsql;

create operator ## (procedure=MaxInt, leftarg=integer, rightarg=integer)

create operator ## (procedure=MaxDouble, leftarg=numeric, rightarg=numeric)

select 5##6

select 5.6##2.1

Другой вариант полиморфизма имеет аналогию с использованием виртуальных функций и обобщений в объектно-ориентированных языках программирования. Так, PostgreSQL определяет понятие полиморфных типов (псевдотипов), которые можно рассматривать как обобщенные типы при создании обобщенных функций, или как базовые абстрактные типы, от которых наследуется все конкретные типы, реализуемые в СУБД. Это типы anyelement, anyarray, anynoarray, anyenum и anyrange. Здесь наиболее обобщенным вариантом является типа anyelement. Остальные типы накладывают небольшие фильтры на использование типов. Очевидно, что anyarray позволяет использовать массив любого типа данных, а anyenum – любой перечислимый тип.

Полиморфные типы можно использовать в качестве параметров и при необходимости как возвращаемое значение в так называемых полиморфных функциях. Однако следует помнить, что такая функция будет работать только с одним конкретным типом данных. Также следует отметить, что использование полиморфного типа только для возвращаемого значения недопустимо.

Например, создадим полиморфную функцию создания массива из двух элементов, которые передаются функции как параметры:

create function make\_array(anyelement, anyelement) returns anyarray as

$$

select array[$1, $2];

$$ language sql;

select make\_array(1,2);

При вызове функции с параметрами-строками произойдет ошибка:

select make\_array('abc','bca') as textarray;

не удалось определить полиморфный тип, так как входные аргументы имеют тип "unknown"

Это связано с неоднозначностью строковых констант. С помощью операции приведения типа (причем, отметим, что нет разницы, в каком аргументе функции) функция будет вызвана корректно:

select make\_array('abc'::text,'bca') as textarray;