RxO - прототип.
Объектно-ориентированные SQL подобные языковые расширения.

Евгений Григорьев (C) 2011
Grigoriev.e@gmail.com

Содержание

Введение

Со времен манифестов баз данных [1,2,3] было сделано большое число теоретических и практических попыток соединить возможности объектно-ориентированных языков программирования и реляционных СУБД. Общее решение до сих пор отсутствует; предлагаемые подходы не пользуются большой популярностью у специалистов[6,7,8,9,10].

Используемый далее подход [5] базируется на простой идее использовать одни и те же имена и их комбинации для того, что бы представить одни и те же значения как в виде сложных объектов, так и в виде таблиц. Этот подход реализован в прототипе называемом далее "RxO-системой".

RxO-система представляет собой программу (C#), которая эмулирует активность СУБД сервера. Она выполняет входящие команды непроцедурного языка высокого уровня и выводит результат. Собственная функция этой программы – трансляция команд входного языка в язык "программируемой табличной машины" (в качестве такой машины выступает MS SQL сервер). Являясь исключительно транслятором, прототип не создает в своей памяти какие-либо переменные для промежуточного представления объектов, не использует какие-либо итераторы и не создает курсоры для выполнения групповых операций над множествами сложных объектов. RxO-прототип не является ни системой OR-маппинга, ни ООСУБД.

RxO – система выполняет функции…
1) …инструмента объектно-ориентированного моделирования предметной области. Команды DDL RxO – системы реализуют следующие возможности:

- Возможность создания классов со сложной (0NF) структурой и ссылками между классами
- Инкапсуляция свойств класса для отдельных компонентов объектов в т.ч. состояния, свойства персистентности данных, поведения и ограничений целостности данных.
- Возможность переопределения указанных свойств при множественном наследовании классов

2) … среды персистентного существования объектов. RxO DML включает команды, позволяющие создавать объекты, манипулировать состоянием существующих объектов, выполнять методы объектов и т.д. Эти действия могут быть выполнены для любого

заданного множества объектов и не требуют определения экстентов класса или использования итераторов.
3)…инструмента, позволяющим получать данные о состоянии объектов, существующих на стороне сервера. Реализующий незапланированные (ad-hoc) запросы RxO DQL можно рассматривать как надмножество традиционного SQL, позволяющее в одном выражении данные из множеств объектов. Отметим, что запросы возвращают данные об объектах, а не сами объекты.

Созданный прототип реализует несколько новых языковых конструкций, синтаксически близких к традиционному SQL. Новыми являются команды DDL, позволяющие определять и описывать классы.`CREATE CLASS` и `ALTER CLASS … REALIZE`, и команды DML, служащие для создания объектов `NEW` и их уничтожения `DESTROY`. Для изменения состояния объектов служат традиционные SQL-DML команды `INSERT`, `UPDATE`, `DELETE`, получившие возможность работать с данными, описанными в терминах структур более сложных, чем таблицы. То же самое можно сказать о SQL-DQL команде `SELECT`.

Хотя это и не реализовано в прототипе, в ходе дальнейшего обсуждения станет ясно, что предлагаемый подход не препятствует использованию традиционных табличных структур, что делает возможным эволюционное развитие существующих SQL-СУБД в направлении, альтернативном текущим стандартам SQL и существующим объектно-реляционным СУБД [11, 12, 7].

Уточним, что главной целью статьи является демонстрация подхода, позволяющего соединить возможности ОО-языков и реляционных СУБД в рамках одной системы, одного языка, одного пространства имен. Используемые далее языковые конструкции показывают лишь примерное направление, и не являются предметом обсуждения.

В качестве иллюстрации используется простая модель торгового предприятия. Мы постарались использовать как можно более "говорящие" имена и комментарии. Далее будет приведен полный код примера в порядке выполнения команд. В тексте код примера заключен в рамку.

Создание классов.

Создание классов включает два раздельных шага
1) Спецификация класса (команда CREATE CLASS …) определяет существование класса и его интерфейс.
2) Для каждого компонента и метода, перечисленного в спецификации класса, задается своя реализация (команда ALTER CLASS… REALIZE …). Объекты класса могут быть созданы только после того, как реализованы все компоненты класса.

Спецификация определяет класс и полностью описывает интерфейс, по которому можно обращаться к данным и методам объектов класса. В спецификации перечисляются родительские классы, значимые (т.е. имеющие значение) компоненты, методы (возможно параметризованные) и ключи.

Значимые компоненты имеют один из следующих типов.
1) Скалярные базовые тип. В прототипе реализованы следующие типы: - BOOLEAN, INTEGER, FLOAT, STRING, DATETIME.
2) Скалярные ссылочные типы. В качестве имени ссылочного типа используется имя ранее определенного класса.

3) Набор (конструкция SET OF). Компонент типа набор представляет собой множество кортежей, определенных на скалярных (базовых и ссылочных) типах. Для наборов могут быть заданы ключи (необязательно).

Необязательные ключи также могут быть заданы для классов в целом. Связь между объектами различных классов может быть задана как ссылками, так и внешними ключами.

В нашей модели мы будем оперировать следующими классами:

```
CREATE CLASS BANKS   //БАНКИ - Очень простая структура класса
{
  Name STRING;
  BIC STRING;
}KEY uniqBIC (BIC)   //ключ класса - БИК уникальный код для каждого банка
;
```

```
CREATE CLASS CONTRACTORS // КОНТРАГЕНТЫ – тоже простая структура
{
  Name STRING;
  Bank BANKS;            // это компонент-ссылка на БАНКИ
  BankAcc STRING;
  INN STRING;
}KEY uniqINN (INN)       // ключ класса - ИНН уникальный для клиента
;
```

```
CREATE CLASS GOODS  // «карточки» ТОВАРОВ -  описание, операции, остатки
{
  Art STRING;
  PricePL FLOAT;    // цена прайс-листа
  Turnover SET OF   // Компонент-набор - все операции по товару
  {
    Comment STRING;              // описание операции
    Date DATETIME;           // дата
    DocNo STRING;            // номер документа
    Contractor CONTRACTORS;  //ссылка на КОНТРАГЕНТОВ
    Pieces INTEGER;          // кол-во
  }KEY Key4GoodsList (DocNo, Contractor); //ключ компонента-набора
  Pieces2IN INTEGER;  // общий планируемый приход
  Pieces INTEGER;     // текущий остаток на складе
  PiecesRes INTEGER;  // зарезервировано для отгрузок
  PiecesFree INTEGER; // свободный остаток
}KEY UniqArt (Art)   //ключ класса - товар уникален своим артикулом
;
```

```
CREATE CLASS GOODSMOVEMENTS    // товарные операции ("движения товаров")
{
  DocNo STRING;
  Contractor CONTRACTORS;       // ссылка на КОНТРАГЕТОВ
  DocDate DATETIME;             // дата заказа
  Comment STRING;               // комментарии
  PostIt(inDate DATETIME);      // метод "учесть"
  PostingDate DATETIME;         // дата учета

  Items SET OF                  // компонент-набор - какие товары "двигаем"
  {
    Art STRING;                 // артикул - (внешний ключ класса см.ниже)
    Pieces INTEGER;             // штук
  }KEY uniqArt (Art);           // ключ набора - артикулы  не повторяются
}KEY uniqDocNo (DocNo)          // ключ класса - уникальный номер документа
REFERENCE ToUniqArt             // внешний ключ - артикулы товарных позиций
Items.(Art) ON GOODS.UniqArt      //  должны существовать в ТОВАРАХ
;
```

Использование спецификация класса является единственным и достаточным условием для обращения к этому классу, например, что бы выполнить запрос к этому классу. В нашем примере, сразу же после того, как были определены описанные выше классы, можно выполнить следующий запрос (его детальное описание будет дано позже)

Естественно, на данном этапе он ничего не вернет – классы пусты. Объекты еще не могут быть созданы, поскольку в нашем примере классы пока не реализованы. Однако, поскольку спецификация классов уже задана, система выполнит запрос к ним.

Реализация класса определяется отдельно для каждого компонента классов. Для этого используется команда

```
ALTER CLASS имя_класса
REALIZE сигнатура_компонента_или_ метода
AS способ_реализации;
```

Основным свойством, реализуемым для значимых компонентов, является свойство хранимости (persistence). Значимые компоненты могут быть реализованы двумя способами.

1) Как хранимый

```
… AS STORED;
```

Значение такого компонента хранятся в системе (подобно тому, как хранятся данные в таблицах). Компоненты, содержащие ключевые поля, реализуются только как хранимые.

2) Как вычисляемый с помощью процедуры, возвращающей значение

```
…AS
{
    тело_процедуры
};
```

Такой компонент можно рассматривать как функцию, возвращающую скалярное или табличное значение.

Методы класса реализуются только как процедуры и не возвращают значения.

Реализация компонентов инкапсулирована в классе. В классах-наследниках реализации наследуемых компонентов и методов могут переопределяться (см далее " Множественное наследование и полиморфизм").

Реализация компонентов может быть изменена в существующих непустых классах.

В нашей модели все компоненты классов BANKS и CONTRACTORS будет реализованы как хранимые

```
ALTER CLASS BANKS
REALIZE
  Name STRING
AS STORED;
```

```
ALTER CLASS BANKS
REALIZE
  BIC STRING
AS STORED;
```

```
ALTER CLASS CONTRACTORS// несколько скалярных компонентов могут
REALIZE (          // реализоваться как хранимые одной командой
  Name STRING,
  Bank BANKS,
  BankAcc STRING,
  INN STRING
)AS STORED;
```

Для класса GOODS некоторые компоненты будут реализованы как вычисляемые. Так компонент Turnover, содержащий информацию о движениях по товару, будет вычисляться на основании данных класса GOODSMOVEMENTS, описывающего «движения» товаров. (используемое при этом SELECT-выражение будет описано далее). Отметим, что, реализуя компонент Turnover, мы обращаемся к классу GOODSMOVEMENTS, используя лишь его спецификацию - этот класс еще не реализован.

```
ALTER CLASS GOODS    // реализуем
REALIZE
  Turnover SET OF    // оборот
  {
    Comment STRING;
    Date DATETIME;
    DocNo STRING;
    Contractor CONTRACTORS;
    Pieces INTEGER;
  }KEY Key4GoodsList (DocNo, Contractor)
AS
{
RETURN               // запросом
SELECT
  #g.Comment,
  #g.DocDate,
  #g.DocNo,
  #g.Contractor,
  SUM(#g.Items.Pieces) AS Pieces
FROM GOODSMOVEMENTS #g // из GOODSMOVEMENTS
WHERE #g.Items.Art = Art  // для данного артикула (this-поле Art)
GROUP BY
  #g.Comment,
  #g.DocDate,
  #g.DocNo,
  #g.Contractor ;
};
```

Скалярные компоненты, содержащие остатки и планируемые количества, вычисляются агрегирующими запросами к тому же классу `GOODSMOVEMENTS`.

```
ALTER CLASS GOODS           // реализуем..
REALIZE
Pieces2IN INTEGER AS        // ..кол-во планируемых поставок
{
RETURN
SELECT SUM(#g.Items.Pieces) // как сумму количеств из строк
FROM GOODSMOVEMENTS #g   // «движений»
WHERE #g.PostingDate IS NULL  // которые еще непроведены
  AND #g.Items.Art = Art      // для данного артикула(this-поле Art)
  AND #g.Items.Pieces > 0;    // только положительные кол-ва
};
```

```
ALTER CLASS GOODS // реализуем кол-во на складе
REALIZE
Pieces INTEGER AS
{
RETURN
SELECT SUM(#g.Items.Pieces) // как сумму количеств из строк
FROM GOODSMOVEMENTS #g   // «движений»
WHERE #g.Items.Art = Art    // для данного артикула(this-поле Art)
  AND #g.PostingDate IS NOT NULL;  //которые уже проведен
};
```

```
ALTER CLASS GOODS // реализуем кол-во планируемых отгрузок
REALIZE
PiecesRes INTEGER AS
{
RETURN
SELECT SUM(-#g.Items.Pieces) // как сумму и убираем знак `-`
FROM GOODSMOVEMENTS #g
WHERE #g.Items.Art = Art
  AND #g.PostingDate IS NULL
  AND #g.Items.Pieces < 0;   // отрицательные кол-ва
};
```

Наконец компонент, содержащий значение о свободном остатке, реализуется методом, использующим значения скалярных this-компонентов.

```
ALTER CLASS GOODS // реализуем кол-во доступное для отгрузки
REALIZE
PiecesFree INTEGER AS
{
  DECLARE         //объявляем локальные переменные
  {
    tmpPieces INTEGER;
    tmpPiecesRes INTEGER;
  }
  tmpPieces := Pieces;                        //берем общий остаток
  IF( tmpPieces IS NULL) THEN tmpPieces := 0;//избавляемся от NULL
  tmpPiecesRes := PiecesRes;                  //берем зарезервированное кол-во
  IF( tmpPiecesRes IS NULL) THEN tmpPiecesRes := 0;   //избавляемся от NULL
  RETURN tmpPieces - tmpPiecesRes;  //вычисляем доступное кол-во
};
```

Класс `GOODS` содержит одновременно и вычисляемые и хранимые компоненты.

```
ALTER CLASS GOODS
REALIZE (
  Art STRING,
  PricePL FLOAT
)AS
STORED;
```

Все значимые компоненты класса `GOODSMOVEMENT` реализованы как хранимые – в том числе и компонент `Items` (далее это свойство компонента будет переопределено - см. "Множественное наследование и полиморфизм.")

```
ALTER CLASS GOODSMOVEMENTS
REALIZE (
  DocNo STRING,
  Contractor CONTRACTORS,
  DocDate DATETIME,
  PostingDate DATETIME,
  Comment STRING
)AS
STORED;
```

```
ALTER CLASS GOODSMOVEMENTS
REALIZE
  Items SET OF
  {
    Art STRING;
    Pieces INTEGER;
  }KEY uniqArt (Art)
AS
STORED;
```

Забегая вперед, скажем, что свойство хранимости компонента `Items` будет переопределено в ходе наследования класса `GOODSMOVEMENTS` (см. Множественное наследование и полиморфизм.)

Реализуем метод `DoPost` заданный спецификацией класса `GOODSMOVEMENT`

```
ALTER CLASS GOODSMOVEMENTS
REALIZE PostIt(inDate DATETIME)
AS
{
IF(PostingDate IS NULL) THEN
  {
  PostingDate := inDate;              // заполняем поле "дата учета".
  Comment := "DONE " + DocNo;         // делает пометку в комментариях
  IF( DocDate < '2010.01.01') THEN    // для старых заказов
    Comment := "OLD! " + Comment;     // делает особую пометку в комментариях
}
};
```

От путей к табличным видам.

Наиболее важной является возможность использования в командах DML и DQL путей, то есть обусловленных структурой классов и ссылками между ними последовательностей имен, определенных в спецификации этих классов. Для записи путей используется точечная нотация. Смысл путей – отразить иерархию, указать часть чего-то общего, что, в свою очередь, может являться частью чего-то более общего - кажется достаточно очевидным. Аналогичные по смыслу и по записи ссылки и ссылочные конструкции широко используются , в частности, в ОО-языках программирования.

Пути RxO-системы можно рассматривать как более общий случай таких конструкций, поскольку они начинаются любым определенным в текущем контексте выражением, которое определяет множество объектов. Это может быть имя класса (что подразумевает все существующие объекты класса), имя ссылки (единственный объект) или путь, оканчивающийся на имя ссылки (подмножество объектов класса).

В простейшем случаем путь может состоять из одного имени.

Тип пути определяется типом его последнего элемента.

В глобальном контексте любой путь начинается именем класса. При это, если путь состоит из единственного имени класса, считается, что он имеет ссылочный тип. В нашем примере в глобальном контексте среди прочих существуют следующие пути

```
BANKS.Name                          2B
CONTRACTORS                         1R
GOODSMOVEMENTS.Contractor.Name      3B
GOODS.Turnover                      2S
GOODS.Turnover.Pieces               3B
```

```
GOODS.Turnover.Contractor.Bank   4R
```

(Индексы напротив путей содержат их краткую характеристики: верхний содержит длину, нижний определяет тип, где B соответствует базовому типу, S – типу-набору, R – ссылочному типу, M – методу.)

В this-контекстах (т.е. в контекстах классов и в локальных контекстах методов) путь может начинаться также именем значимых компонентов класса (если это ссылки или наборы) или локальных переменных (также ссылок и наборов).

Например в контексте класса `GOODSMOVEMENT` определен путь (среди прочих)
`Contractor.Bank.Name` $^{3}_{B}$
В контексте класса `GOODS` определены пути (среди прочих)
`Turnover` $^{1}_{S}$
`Turnover.Conragent` $^{2}_{R}$

Пути, имеющие тип ссылки или тип набора, имеют путевые продолжения Будем называть такие пути нетерминальными. Нетерминальные пути могут иметь множество путевых продолжений. Например путь `GOODSMOVEMENTS.Contractor` (ссылочный тип `CONTRACTOR`) имеет, среди прочих, следующие путевые продолжения (их краткая характеристика содержит знак "+" в верхнем индексе)
`.Name` $^{+1}_{B}$
`.INN` $^{+1}_{B}$
`.Bank` $^{+1}_{R}$
`.Bank.Name` $^{+2}_{B}$
Путевые продолжения начинаются на точку. Этот синтаксический прием позволяет отделить имена, определенные как продолжения используемого к команде пути, от имен, определенных в текущем контексте.

Пути, имеющие базовый тип, являются терминальными, т.е. не имеют путевых продолжений.

Любое содержащееся в пути имя класса или ссылки при необходимости может быть дополнено выражением отбора объектов
имя_класса_или_ссылки[*набор_критериев*]

Здесь критерии (аналоги критериев WHERE части выражения SELECT в SQL) применяются к путевым продолжениям скалярного типа (т.е. к скалярным компонентам и к атрибутам компоенетов-набора). Критерии могут сочетаться традиционными для SQL операциями AND и OR. Кроме того, для компонентов-наборов имеет смысл так называемый межстроковый_AND имеющая более низкий, по сравнению с операциями OR, AND, NOT , приоритет. Эта операция записывается просто запятой ",". Например, для того, что бы получить ссылку на объект класса GOODSMOVENTS, содержащий как строку с артикулом "`tShirt01`", так и строку с с артикулом"`Hat03`", можно использовать следующее выражение
```
GOODSMOVEMENTS[.Items.Art = "tShirt01", .Items.Art = "Hat03"]
```

Выражения отбора по объектам могут вкладываться и сочетаться произвольным образом. Например, синтаксически корректным будет следующий путь
```
GOODS[
      .Art BETWEEN …,
      .Turnover.Contractor[.Bank.BIC = …]]
.Turnover
.Contractor[.Name LIKE Name]
```

В последней строке сравниваются значения путевого продолжения `.Name` пути `GOODS.Turnover.Contractor` и переменной `Name`, которая должна быть определена в текущем контексте.

Общий принцип, определяющий использование путей в командах DML и DQL звучит следующим образом: любой нетерминальный путь может трактоваться как имя табличного вида, содержащего атрибуты, имена которых представляют некоторые скалярные путевые продолжения этого пути. Формальное обоснование этого преобразования представлено в [5].

Например, выражение "`GOODSMOVEMENTS[.Date >= '2010.01.01'].Contractor`" является путем, определяющим множество контрагентов, для которых после 01 января 2010 года выполнялись "движения" товаров. Этот путь допускает (среди прочих) путевые продолжения "`.Name` ", "`.INN` " и "`.Bank.Name` ". Это позволяет утверждать, что в нашей примере, среди многих прочих, можно обратиться к следующему табличному виду

```
GOODSMOVEMENTS[.Date >= '2010.01.01'].Contractor   \\ это имя вида
(.Name , .INN , .Bank.Name )                       \\ это атрибуты вида
```

Совокупность имени вида (т.е. пути) и его атрибутов (используемых путевых продолжений) образует полную сигнатуру этого вида.

Среди прочих, доступны также табличные виды со следующими сигнатурами

```
BANKS
( .Name , .BIC)

BANKS
( .Name)

GOODS
( .Art , .Turnover.DocNo , .Turnover.Date, .Turnover.Pieces )

GOODS.Turnover
( .Contractor.Name, .DocNo , .Date, .Pieces )

GOODS[.Art LIKE "Hat%"].Turnover
( .Contractor.Name, .DocNo , .Date, .Pieces )
```

… и многие-многие другие.

Будем называть такие табличные виды О-видами.

Важно понимать, что когда мы говорим об О-видах, выражение вроде "`GOODS.Turnover`" рассматриваются нами всего лишь как имя (в данном случае табличного вида), то есть как строка, которая отлична от другого строки, например от "`GOODS[.Art LIKE "Hat%"].Turnover`". То же самое можно сказать о выражениях "`.Art`" , "`.Turnover.DocNo`" , "`.Turnover.Date`". Используя их как имена столбцов табличного вида, мы рассматриваем их лишь как отличающиеся друг от друга строки. Все эти имена *несут смысл* ранее определенной сложной структуры данных, хотя эта такая структура в табличных О-видах конечно отсутствует. Таким образом, в табличном представлении данных, используются те же имена и последовательности имен, что были заданы в спецификации классов.

Это позволяет осуществить переход от определяемых спецификацией классов сложных иерархических структур данных к табличным представлением этих же данных в рамках одного пространства имен. Более того – для пользователя этот переход происходит практически незаметно.

Таким образом, в RxO-системе все данные о совокупности объектов множества классов представляются как множество табличных O-видов, причем между множеством классов и множеством возможных O-видов нет однозначного соответствия. Один класс так или иначе соответствует множеству разных O-видов. В одном O-виде может быть представлены данные множества разных классов (связанных ссылками и\или участвующих в наследовании). Связь между множеством классов и множеством табличных O-видов основывается на именах.

Конечно, описываемые O-виды не определяются в строгом смысле этого слова - так, как определяются виды в традиционном SQL. Подразумевается, что O-виды *могут быть вычислены*. Любая команда, обращающаяся к данным, так или иначе будет содержать пути и продолжения путей, которые, в совокупности, образует сигнатуры O-видов. RxO-система анализирует эти пути и, если они удовлетворят сформулированному общему принципу, создает (вычисляет) вид и выполняет команду.

В процессе вычисления вида компоненты, так или иначе упомянутые в сигнатуре, связываются с реализациями этих компонентов (позднее связывание). С учетом того, что RxO-система допускает наследование классов и переопределение реализаций, с каждым из компонентов могут быть одновременно связаны и хранимые и вычисляемые по разным алгоритмам реализации. По сути табличные O-виды являются полиморфными (см далее " Множественное наследование и полиморфизм").

К данным, представленным в форме табличных O-видов, применяются обычные SQL команды DQL и, с некоторыми ограничениями, DML.

Создание и существование объектов.

Для создания объектов используется команда NEW, которая может быть использована либо отдельно, либо в коде метода, в том числе как правая часть выражения, инициализирующего ссылочные переменные.

```
NEW имя_класса [WITH...]
refVar := NEW имя_класса [WITH...]
```

Необязательная часть `WITH` содержит конструирующие выражения, позволяющее инициализировать компоненты объекта во время создания (в прототипе эта возможность реализована только для скалярных компонентов).

```
NEW BANKS WITH SET // создаем новый объект
    .Name := "CitttiBank",  // инициализируем компоненты
    .BIC := "999999999999999";
```

Отметим, что создавая объект, мы не обязаны сохранять ссылку на него. Созданные таким образом объекты не «теряются». Относясь к классу, они всегда доступны при групповых операциях с классом. Все созданные объекта хранятся до тех пор, пока не будут явно уничтожены командой `DESTROY`, например

```
DESTROY BANKS[.BIC IS NULL];//кстати, такой объект невозможен: BIC – ключ
```

Ее синтаксис - `DESTROY` $путь_R$

Здесь путь$_R$ означает, что в качестве аргумента может использоваться любой путь ссылочного типа. Отметим, что если на объекты существуют ссылки из других частей системы, они не могут быть уничтожены.

Единичная ссылка на любой объект всегда может быть получена оператором
ONE OF *путь$_R$*
например

```
NEW CONTRACTORS         // создаем новый объект
WITH SET
  .Name := "Levis",
  .Bank := (ONE OF BANKS[.Name = "CitttiBank"]), //получаем ссылку на объект
  .BankAcc := "40601000000000000001",
  .INN := "77777777777";
```

Команда NEW может быть вложена.

```
NEW CONTRACTORS // создаем новый объект
WITH SET
  .Name := "X3 retail group",
  .Bank :=               // инициализируем компонент ссылочного типа ссылкой
  (NEW BANKS WITH SET    // на вновь создаваемый объект
    .Name := "VoTaBe25",
    .BIC := "88888888888888888"),
  .BankAcc := "40601000000000000333",
  .INN := "770000077777";
```

Создадим несколько объектов класса GOODS, описывающего товары.

```
NEW GOODS WITH SET .Art := "tShirt01", .PricePL := 1;
NEW GOODS WITH SET .Art := "Trousers05", .PricePL := 5;
NEW GOODS WITH SET .Art := "Hat03", .PricePL := 3;
```

Изменение данных.

Для изменения значений объектов используются команды UPDATE, INSERT и DELETE, использующие в качестве параметров пути. Они имеют традиционный SQL синтаксис. Команда UPDATE применяется ко всем типам компонентов – и к ссылкам и к наборам (R+S в нижнем индексе).
UPDATE путь$_{R+S}$ SET {путь$^{+1}_{B+R}$:= ...}(,n)...
(здесь путь$^{+1}_{B+R}$ означает, что присваивание выполняется для путевых продолжений базового или ссылочного типа, имеющих длину в одно имя).

команды INSERT и DELETE применяются только для наборов.
INSERT INTO путь$_S$ ({путь$^{+1}_{B+R}$:= ...}(,n))
DELETE FROM путь$_S$

Например, команда

```
INSERT INTO GOODSMOVEMENTS[.DocNo = " InMoveEx"].Items (.Art, .Pieces)
VALUES("tShirt01", 10);   //"движение" с таким номером отсутствует
```

добавляет в компонент Items всех объектов класса GOODSMOVEMENTS, у которых поле DocNo равно "InMoveEx" строку со значением ("tShirt01", 10). Поскольку поле DocNo является ключевым, эта команда будет выполнена максимум для одного объекта.

Все описанные в данном разделе команды могут выполняться как в глобальном контексте, так и в методах и применятся как к классам, так и к this-локальным компонентам и переменным.

Помимо этого, в процедурах, реализующих компоненты и методы классов, для скалярных компонентов класса и для локальных скалярных переменных может использоваться операцию присваивания (см. например реализацию метода `DoMovement`).

`локальный_путь`$_{B+R}$ `:= ...`

Операция присваивания может применяться к локальным путям любой длины, при условии, что они не содержат имен компонентов-наборов.

Выполнение методов классов осуществляется при помощи команды

`EXEC путь`$_{M}$ `([параметры])`

Создадим два объекта класса `GOODSMOVEMENT`, описывающего «движения» товаров, и добавим в них записи о перевозимых товарах.

```
NEW GOODSMOVEMENTS WITH SET
  .DocNo := "InMove01",
  .Contractor := ONE OF CONTRACTORS[.Name = "Levis"],
  .DocDate :='2010.01.01', //старая поставка
  .Comment := " 1st SPL";
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("tShirt01", 10);
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("Hat03", 10);
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove01"].Items (.Art, .Pieces)
VALUES("Trousers05", 10);
```

```
NEW GOODSMOVEMENTS WITH SET
  .DocNo := "InMove02",
  .DocDate :='2011.02.02', //новая поставка
  .Contractor := ONE OF CONTRACTORS[.Name = "Levis"];
INSERT INTO GOODSMOVEMENTS[.DocNo = "InMove02"].Items (.Art, .Pieces)
VALUES("tShirt01", 20);
```

Запросы.

Подобно командам DML, синтаксис SELECT-выражений, используемый для запросов, практически не отличается от используемого в SQL, за исключение того, что используются произвольные нетерминальные пути (как имена используемых в части FROM перечисляющей табличные источники данных) и любые их путевые продолжения (как имена столбов-атрибутов этих табличных источников).

Мы уже выполняли запрос

```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Comment,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces,
  #g.PricePL
FROM
GOODSMOVEMENTS #gm JOIN GOODS #g ON #gm.Items.Art = #g.Art;
```

В этом запросе (далее – Q#1) используются табличные виды со следующими сигнатурами:

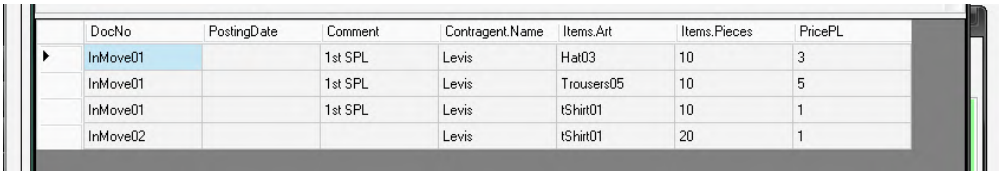
```
GOODSMOVEMENTS
  (.DocNo, .PostingDate, #gm.Comment, .Contractor.Name,
   .Items.Art, .Items.Pieces)
```

и

```
GOODS
```

```
  (.Art, .PricePL)
```

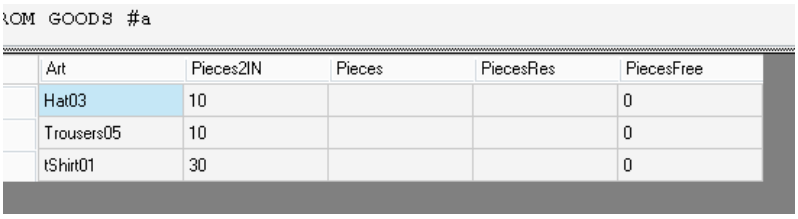
Основываясь на сигнатурах, система в первую очередь вычислит требуемые O-виды. Затем к ним могут быть применены любые обычные для таблиц операции (здесь JOIN). Результат представлен на следующей иллюстрации.



DocNo	PostingDate	Comment	Contragent.Name	Items.Art	Items.Pieces	PricePL
InMove01		1st SPL	Levis	Hat03	10	3
InMove01		1st SPL	Levis	Trousers05	10	5
InMove01		1st SPL	Levis	tShirt01	10	1
InMove02			Levis	tShirt01	20	1

Приведем пример другого запроса (далее Q#2), возвращающего данные о статусе товаров

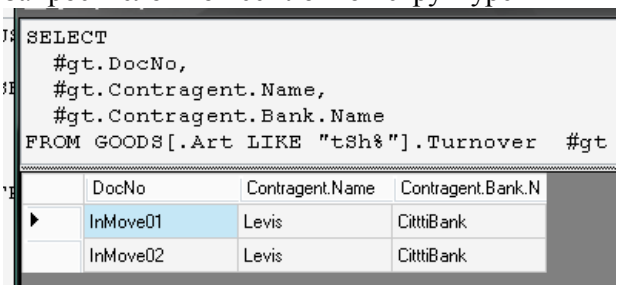
```
SELECT
  #a.Art,
  #a.Pieces2IN,  // pieces to be received
  #a.Pieces,     // pieces on stock
  #a.PiecesRes,  // pieces reserved to be shipped
  #a.PiecesFree  // pieces free to ship
FROM GOODS #a
```



```
ROM GOODS #a
```

Art	Pieces2IN	Pieces	PiecesRes	PiecesFree
Hat03	10			0
Trousers05	10			0
tShirt01	30			0

Запрос в сложной ссылочной структуре

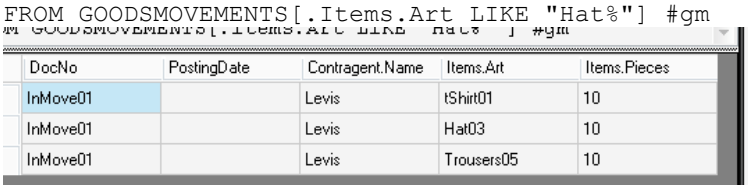


```
SELECT
  #gt.DocNo,
  #gt.Contragent.Name,
  #gt.Contragent.Bank.Name
FROM GOODS[.Art LIKE "tSh%"].Turnover  #gt
```

DocNo	Contragent.Name	Contragent.Bank.N
InMove01	Levis	CitttiBank
InMove02	Levis	CitttiBank

Условие, используемое в следующем запросе, задается выражением отбора объектов в пути, формирующем O-вид. Результатом являются данные об объектах, которые удовлетворяют заданному условию.

```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces
FROM GOODSMOVEMENTS[.Items.Art LIKE "Hat%"] #gm
```



DocNo	PostingDate	Contragent.Name	Items.Art	Items.Pieces
InMove01		Levis	tShirt01	10
InMove01		Levis	Hat03	10
InMove01		Levis	Trousers05	10

Это же условие применяется в WHERE части SELECT выражения к уже вычисленному О-виду. Результатом являются строки, удовлетворяющие условию

```
SELECT
  #gm.DocNo,
  #gm.PostingDate,
  #gm.Contractor.Name,
  #gm.Items.Art,
  #gm.Items.Pieces
FROM GOODSMOVEMENTS  #gm
WHERE #gm.Items.Art LIKE "Hat%"
```

DocNo	PostingDate	Contragent.Name	Items.Art	Items.Pieces
InMove01		Levis	Hat03	10

Множественное наследование и полиморфизм.

Спецификация класса-наследника является объединением множеств наследуемых и собственных компонентов и методов. Реализации наследуемых компонентов и методов могут быть переопределены. Для методов и в запросах используется механизм позднего связывания. Всё это позволяет развивать модель предметной области путем наследования классов и переопределения реализаций. Проиллюстрируем это в нашем примере.

Добавим новый класс, содержащий записи о денежных операциях

```
CREATE CLASS VALUERECORDS
{
  VRDocNo STRING;       // номер фин.документа
  VRDate DATETIME;      // дата записи
  VRComment STRING;     // комментарии
  ExpectedAmount FLOAT;// ожидаемая сумма
  ValueAmount FLOAT;    // сумма
};
```

Создадим класс SALES, содержащий информацию о продажах. Поскольку продажи связывают товарные и денежные операции, объявим его наследником классов GOODSMOVEMENTS и VALUERECORDS.

```
CREATE CLASS SALES
EXTENDED GOODSMOVEMENTS, VALUERECORDS
{
  SaleItems SET OF   //компонент-набор: данные о проданных товарах
  {
    Art STRING;       //артикул
    Price FLOAT;      //цена
    Pieces INTEGER;   //штуки
  }KEY uniqArtPrice (Art,Price);
}
REFERENCE ToUniqArt SaleItems.(Art) ON GOODS.UniqArt;
```

Компоненты класса VALUERECORDS пока не имеют собственных реализаций. Определим их в классе SALES.

```
ALTER CLASS SALES
REALIZE
VRDocNo STRING   // в качестве номера документа о финансовой операции...
AS
{
  RETURN DocNo;  //...используем номер товарного заказа
};
```

```
ALTER CLASS SALES
REALIZE
  VRComment STRING // в финансовых комментариях указываем, что ...
AS
{
  RETURN " FinDoc#" + DocNo + "(" + Comment + ")";//...речь идет о продаже
номер...
};
```

```
ALTER CLASS SALES
REALIZE
  VRDate DATETIME //датой финансовой операции ...
AS
{
  RETURN PostingDate; //...является дата выполнения отгрузки
};
```

```
ALTER CLASS SALES
REALIZE
  ValueAmount FLOAT // значение суммы ...
AS
STORED              // ...хранится в системе
;
```

```
ALTER CLASS SALES
REALIZE
  ExpectedAmount FLOAT //ожидаемая сумма ...
AS
{                      //вычисляется ...
  RETURN
    SELECT SUM (-#pi.Pieces*#pi.Price)  //как сумма произведений кол-во*цена
    FROM SaleItems #pi;              //по строкам продажи
};
```

Отметим, что компонент `Items`, определенный в классе `GOODSMOVEMENTS`, напрямую связан с вновь определенным компонентом `SaleItems` – сколько товаров продано, столько и должно быть отгружено в товарной операции. Что бы отразить это, реализуем компонент `SaleItems`, содержащий данные о проданном товаре, как хранимый.

```
ALTER CLASS SALES
REALIZE
  SaleItems SET OF
  {
    Art STRING;
    Price FLOAT;
    Pieces INTEGER;
  }KEY uniqArtPrice (Art,Price)
AS STORED;
```

Переопределим реализацию наследованного компонента `Items`, содержащего данные о товаре, участвующем в товарной операции. Напомним, что в базовом классе он был

определен как хранимый. Теперь он будет вычисляться на основании данных о продаваемых товарах

```
ALTER CLASS SALES
REALIZE
  Items SET OF
  {
    Art STRING;
    Pieces INTEGER;
  }KEY uniqArt (Art)
AS
{
  RETURN
  SELECT
    #pi.Art,
    SUM(-#pi.Pieces)
  FROM SaleItems #pi
  GROUP BY #pi.Art;
};
```

Метод `DoPost` , выполняющий учет продаж, тоже должен быть переопределен

```
ALTER CLASS SALES
REALIZE PostIt(inDate DATETIME)
AS
{
IF(PostingDate IS NULL) THEN
  {
  Comment := "Sale is POSTED! " + Comment; //отметка в тов. комментарии
  PostingDate := inDate;                   //указываем дату учета
  ValueAmount :=                           //находим окончательную сумму
    SELECT SUM (-#pi.Pieces*#pi.Price)
    FROM SaleItems #pi;
  }
};
```

Теперь, когда класс SALES полностью реализован, можно внести в систему информацию о двух продажах.

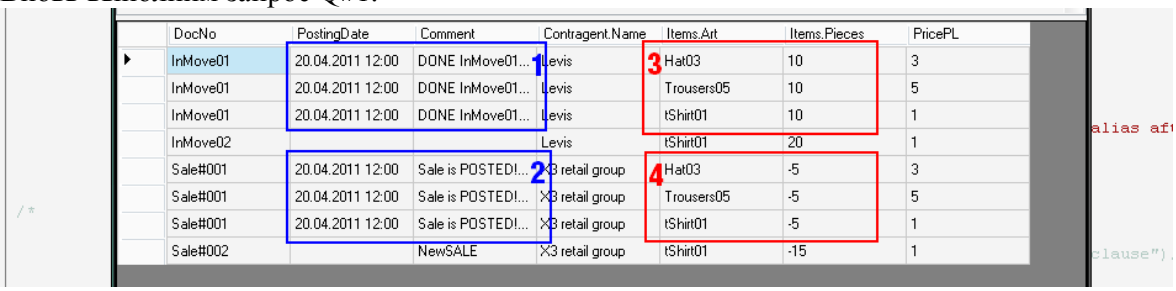
```
NEW SALES WITH SET
  .DocNo := "Sale#001",
  .Comment := "OldSALE",
  .Contractor := ONE OF CONTRACTORS[.Name = "X3 retail group"],
  .DocDate :='2010.11.11'  //старая продажа
;
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("tShirt01", 5, 0.8);
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("Hat03", 5, 2.5);
INSERT INTO SALES[.DocNo = "Sale#001"].SaleItems  (.Art, .Pieces, .Price)
VALUES("Trousers05", 5, 4);
```

```
NEW SALES WITH SET
  .DocNo := "Sale#002",
  .Comment := "NewSALE",
  .Contractor := ONE OF CONTRACTORS[.Name = "X3 retail group"],
  .DocDate :='2011.04.04'  //новая продажа
;
INSERT INTO SALES[.DocNo = "Sale#002"].SaleItems  (.Art, .Pieces, .Price)
VALUES("tShirt01", 15, 4);
```

Итак, в нашей БД существует два объекта класса `GOODSMOVEMENTS` и два объекта класса-наследника `SALES`. Выполним определенный в классе `GOODSMOVEMENTS` полиморфный метод `PostIt()` для объектов, датированных старыми годами.

```
EXEC GOODSMOVEMENTS[.DocDate<'2011.01.01'].PostIt('2011.04.20');
```

Вновь выполним запрос Q#1.



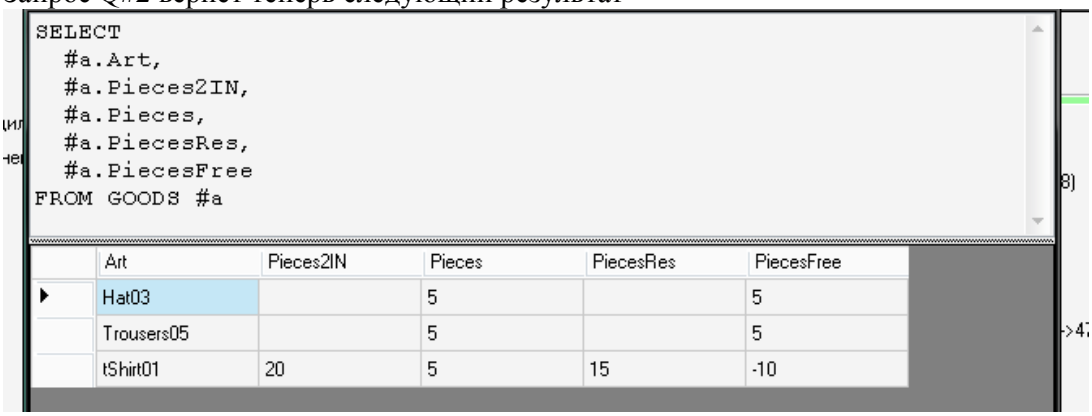
DocNo	PostingDate	Comment	Contragent.Name	Items.Art	Items.Pieces	PricePL
InMove01	20.04.2011 12:00	DONE InMove01...	Levis	Hat03	10	3
InMove01	20.04.2011 12:00	DONE InMove01...	Levis	Trousers05	10	5
InMove01	20.04.2011 12:00	DONE InMove01...	Levis	tShirt01	10	1
InMove02			Levis	tShirt01	20	1
Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	Hat03	-5	3
Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	Trousers05	-5	5
Sale#001	20.04.2011 12:00	Sale is POSTED!...	X3 retail group	tShirt01	-5	1
Sale#002		NewSALE	X3 retail group	tShirt01	-15	1

Столбцы `PostingDate` и `Comment`, представляющие соответствующие скалярные компоненты класса `GOODSMOVEMENT`, содержат значения, являющиеся результатом выполнения полиморфного метода `PostIt()`, имеющего несколько реализаций. На иллюстрации представлены результат исполнения реализаций метода `PostIt()` для класса `GOODSMOVEMENT` (область 1) и для класса `SALES` (область 2). Отметим, что сами компоненты `PostingDate` и `Comment` являются хранимыми и не меняли реализацию при наследовании.

Другим примером полиморфизма являются столбцы `Items.Art` и `Items.Pieces`, представляющие скалярные атрибуты компонента-набора `Items` класса `GOODSMOVEMENT`. Этот компонент изменил реализацию. Область 3 представляет значения так, как они хранятся в системе, а в области 4 представлены результаты группирующего запроса из данных, хранящихся в компоненте, существующем только в классе-наследнике `SALES`, как это определено реализацией компонента в этом класса.

Отметимтаростин, что запрос Q#1 не претерпел никаких изменений с момента своего первого использования, когда для класса `GOODSMOVEMENT` была задана только спецификация, а класс `SALES` вообще не существовал. И то, что значения, представленные в этом запросе, являются результатом достаточно сложных и разнородных вычислений над данными, описывающими состояния множества объектов разных классов, достигается с использованием традиционных ОО-механизмов наследования и полиморфизма, реализованного как для методов, так для значимых компонентов.

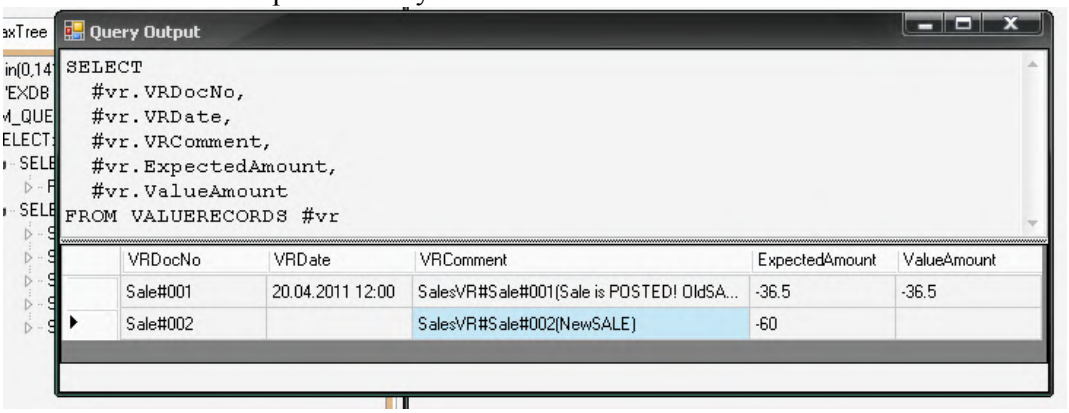
Запрос Q#2 вернет теперь следующий результат



```
SELECT
  #a.Art,
  #a.Pieces2IN,
  #a.Pieces,
  #a.PiecesRes,
  #a.PiecesFree
FROM GOODS #a
```

Art	Pieces2IN	Pieces	PiecesRes	PiecesFree
Hat03		5		5
Trousers05		5		5
tShirt01	20	5	15	-10

Выполним также запрос к классу `VALUERECORDS`



Напомним, что реализации компонентов этого класса были созданы только в классе наследнике `SALES`. Результатом запроса, как видно, являются данные об объектах этого класса.

> Заключение.

Реализуемый RxO-прототипом проход позволяет достичь нового уровня традиционной клиент-серверной архитектуры, когда сервер является независимой средой создания и персистентного существования управляемой объектной модели предметной области с возможностью получения данных о состоянии этой модели.

Отметим отсутствие каких-либо ограничений на создание и совместное использование традиционных для SQL табличных структур. Рассматриваемые языковые расширения можно использовать как надмножество существующих версий SQL, что делает возможной эволюцию существующих SQL-систем, позволяет использовать существующий SQL-код, не требует изменения существующих клиентских приложений и протоколов доступа к БД. Стоит обратить внимание на сравнительную простоту как самой идеи так и реализующих её языковых конструкций.

Предлагаемый подход в определенном смысле противоположен идее сохранения в БД произвольных объектов ОО-языков программирования, которую реализуют многочисленные ООСУБД и ORM-системы. Клиентское приложение использует данные о *других* объектах, описанных *другим* языком и существующих в *другой* системе. Однако именно это делает возможным создание независимой модели предметной области, данные о которой доступны из многих разнородных систем.

Литература.

[1] М. Аткинсон и др. "Манифест систем объектно-ориентированных баз данных", СУБД, No. 4/1995, http://www.osp.ru/dbms/1995/04/23.htm

[2] Стоунбрейкер М. и др. "Системы баз данных третьего поколения: Манифест", СУБД, No. 2/1996, http://www.osp.ru/dbms/1995/02/23.htm

[3] Х. Дарвин, К. Дейт. "Третий манифест", СУБД, No. 1/1996, http://www.osp.ru/dbms/1996/01/23.htm

[4] Donald D. Chamberlin and other. "A History and Evaluation of System R", Communication of ACM, Oct. 1981. http://www.cs.berkeley.edu/~brewer/cs262/SystemR.pdf

[5] Григорьев Е.А. "НадРеляционный Манифест". http://citforum.ru/database/articles/nrm/#6. (PFD версия- http://citforum.ru/database/articles/nrm/nrm.pdf)

[6] С.Д. Кузнецов, М.Н. Гринев "Ландшафт области управления данными: аналитический обзор" Институт системного программирования РАН, 2008 http://citforum.ru/database/data_management_overview/

[7] Кузнецов С.Д. "Объектно-реляционные базы данных: прошедший этап или недооцененные возможности?" (http://citforum.ru/database/articles/ordbms10/)

[8] Roberto V. Zicari (Editor) "Object Database Systems: Quo vadis " (http://www.odbms.org/download/039.03%20Zicari%20Object%20Database%20Systems%20-%20Quo%20vadis%20June%202008.PDF)

[9] William R. Cook "Integrating Programming Languages & Databases: What's the Problem? (draft)" http://www.odbms.org/download/039.03%20Zicari%20Object%20Database%20Systems%20-%20Quo%20vadis%20June%202008.PDF

[10] Peter Baumann "Object-Oriented or Object-Relational? An Experience Report from a High-Complexity, Long-Term Case Study." (http://www.odbms.org/download/oo-or-comparison.pdf)

[11] George Feuerlicht1 and Jaroslav Pokorný and Karel Richta "Object-Relational Database Design: Can YourApplication Benefit from SQL:2003?" (http://silverfish.iiitb.ac.in/ver0/nutch_crawled_pdfs/indexed/OO-to-ORDBMS.pdf)

[12] А. Эйзенберг, Дж. Мелтон "SQL:1999, ранее известный как SQL3" (пер. Кузнецов, С.Д.) http://citforum.ru/database/digest/sql1999.shtml