Модельно-языковые средства управления данными

Гринев Максим Николаевич

Москва 2003

Список таблиц

Оглавление

Введение

Актуальность темы

В настоящее время язык XML используется как основное средство унифицированного представления данных различной степени структурированности. Это приводит к росту объемов XML-данных, которыми необходимо управлять. Одним из ключевых компонентов технологии управления XML-данными является декларативный язык запросов XQuery. Однако имеющиеся средства, основанные на языке XQuery, характеризуются низкой эффективностью, что делает невозможным их практическое использование для управления XML-данными больших объемов. Кроме того, другой важной проблемой использования XML-данных является предоставление конечным пользователям средств доступа в терминах, приближенных к предметной области. Разработка таких средств стимулируется ростом популярности языка UML, который предназначен для описания модели предметной области и для которого существуют отображения на язык XML. Решение этих проблем и определяет актуальность диссертационной работы.

Цель и задачи работы

Целью диссертационной работы является *исследование и разработка средств эффективной поддержки моделей данных UML и XML в рамках единой многоуровневой системы баз данных*. Для достижения этой цели поставлены следующие задачи:

1. Разработка языка UQL, позволяющего формулировать запросы к данным в терминах диаграмм классов UML.

2. Разработка метода трансляции UQL-запросов в запросы на языке XQuery.

3. Разработка методов логической оптимизации выполнения XQuery-запросов.

Основные результаты работы

1. Разработан язык UQL, позволяющий формулировать запросы к данным в терминах диаграмм классов UML.
2. Разработан и реализован метод поддержки языка UQL через трансляцию UQL-запросов в запросы на языке XQuery.
3. Разработаны и реализованы методы логической оптимизации XQuery-запросов.

Научная новизна работы

Научной новизной обладают следующие результаты диссертационной работы:

- переопределена семантика языка OCL с целью создания языка запросов UQL, обеспечивающего доступ к данным в терминах концептуальных схем, которые определяются на языке диаграмм классов UML;
- разработан метод трансляции запросов на языке UQL в запросы на стандартном языке запросов к XML-данным XQuery;
- предложен оригинальный подход к логической оптимизации XQuery-запросов, основанный на использовании метода перезаписи, и методы такой оптимизации в рамках этого подхода.

Практическая значимость

Разработанный язык запросов UQL может служить основой для создания подсистемы поддержки запросов, определяемых в терминах диаграмм классов UML, в различных системах управления данными. К таким системам можно отнести СУБД и системы интеграции данных.

Разработанные методы логической оптимизации выполнения XQuery-запросов могут быть использованы для повышения эффективности

подсистем поддержки XQuery-запросов в XML СУБД и системах интеграции данных на базе XML.

Разработан прототип системы, которая поддерживает UQL-запросы и XQuery-запросы на разных уровнях в единой системе баз данных и включает логический оптимизатор XQuery-запросов. Этот прототип был использован в качестве основы для создания в ИСП РАН промышленной системы виртуальной интеграции BizQuery.

Доклады и публикации

Основные положения работы докладывались на пятой и шестой международных конференциях Advances in Databases and Information Systems (ADBIS) (2001 г. и 2002 г.), на шестьдесят третьем, семьдесят третьем и семьдесят шестом семинарах Московской Секции ACM SIGMOD (2000 г., 2001 г., 2002 г.), на научном семинаре ИСП РАН (2002 г.), на семинаре "Современные сетевые технологии" (2002 г.), на втором семинаре "XML-академия" компании Software AG (2000 г.).

По материалам диссертации опубликовано пять печатных работ [47, 48, 49, 50, 51].

Структура и объем диссертации

Работа состоит из введения, трех глав, заключения и списка литературы. Общий объем диссертации 116 страниц. Список литературы содержит 51 наименование.

Краткое содержание работы

Первая глава является обзорной и содержит изложение принципов, методов и средств, которые лежат в основе разработок, описанных в последующих двух главах. Глава состоит из двух разделов. В первом разделе прослеживается эволюция развития моделей данных; отмечается, что работы последних лет были направлены, главным образом, на разработку моделей данных с повышенным уровнем семантики и методов поддержки таких моделей в системах баз данных; выявляются и обсуждаются факторы, препятствовавшие широкому распространению таких моделей; намечается подход для

преодоления препятствий, вызванных этими факторами. Во втором разделе обсуждаются существующие методы логической оптимизации реляционных и некоторых других запросов; приводится обзор языка запросов XQuery, предназначенного для формулирования запросов к XML-данным; на примерах исследуются возможности и особенности логической оптимизации XQuery-запросов.

Вторая глава посвящена описанию разработанного автором декларативного языка запросов к данным в терминах диаграмм классов UML. Потребность в таком языке обосновывается путем указания возможных областей его применения. Описывается сам язык UQL, предлагается метод его реализации через трансляцию UQL-запросов в запросы на языке XQuery.

Третья глава посвящена описанию предлагаемого автором общего подхода к логической оптимизации выполнения запросов на языке XQuery и разработанных автором методов в рамках этого подхода.

В заключении перечисляются основные результаты работы.

Глава 1

Модельно-языковые средства управления данными и оптимизация запросов на модельном уровне

Настоящая глава является обзорной и содержит изложение принципов, методов и средств, которые лежат в основе разработок, описываемых в следующих двух главах. Кроме того, в этой главе содержится изложение предварительных рассуждений и исследований автора, в которых намечены подходы и методы, обсуждаемые в последующих главах.

1.1 Семантически-ориентированные модели данных

Этот раздел посвящен историческому анализу исследований, направленных на повышение уровня семантики моделей данных, и попыток построения систем баз данных на основе таких моделей. Материал этого раздела организован следующим образом. В первом подразделе вводятся основные понятия, необходимые для однозначного понимания последующего материала. Эта вынужденная мера объясняется сильной перегруженностью различными смыслами многих понятий, связанных с моделями данных, в контексте методологии построения систем баз данных. В следующем подразделе прослеживается эволюция моделей данных в направлении повышения уровня семантики. Начиная рассмотрение с первых моделей данных периода 60-х годов, мы остановимся на середине 90-х годов с целью подвести итоги и определить факторы, препятствовавшие развитию и использованию моделей данных с повышенным уровнем семантики. В третьем подразделе по результатам подведенных итогов обсуждаются способы устранения негативных

факторов в условиях появления во второй половине 90-х годов новых перспективных технологий.

1.1.1 Основные понятия

Под *моделью данных*, в строгом смысле этого термина, будем понимать совокупность трех компонентов: (1) набор элементов, в терминах которых представляются данные в этой модели данных; (2) набор допустимых операций над этими элементами; и (3) набор ограничений целостности, которым должны удовлетворять элементы при представлении любых данных. Совокупность компонентов, входящих в модель данных, принято называть *структурной составляющей* модели данных. Совокупность операций - *манипуляционной составляющей*. Совокупность ограничений целостности - *целостной составляющей*.

Такое определение модели данных было впервые введено Э. Коддом в фундаментальной публикации [2]. Отметим, что сегодня известны модели данных и без манипуляционной составляющей, например, полуструктурированная модель данных Object Exchange Model (OEM) [12], поэтому будем считать наличие манипуляционной составляющей не строго обязательной. Таким образом, будем использовать термин модель данных и в нестрогом смысле, подразумевая при этом обязательность наличия только структурной составляющей и предполагая, что над всякой структурной составляющей можно определить некоторую манипуляционную составляющую, получив тем самым модель данных в строгом смысле этого термина. Такое двоякое определение термина модели данных поможет не акцентироваться на наличии или отсутствии в модели данных манипуляционной составляющей в тех случаях, когда это не имеет особого значения для передачи сути обсуждаемого предмета. В тех же случаях, когда это будет необходимо, будем явно указывать, в каком из двух смыслов был употреблен этот термин.

Под *моделью* или *схемой* будет понимать описание базы данных средствами *языка описания структуры базы данных*, который основан на концепциях структурной составляющей некоторой модели данных.

Под *моделированием данных* будем понимать процесс создания модели (схемы), то есть процесс построения описания модели (схемы) на языке, основанном на концепциях структурной составляющей некоторой модели данных.

1.1.2 Эволюция моделей данных с повышенным уровнем семантики

Работа над первыми моделями данных началась во второй половине 60-х гг. Наиболее значительными разработками в этой области стали *сетевая модель данных* CODASYL [1] и *иерархическая модель данных* компании IBM. Эти модели были реализованы в широко известных системах баз данных IDMS компании DF Gundrich Chemical Company (1971 г.) и IMS компании IBM (1969 г.) соответственно. С использованием этих систем было успешно разработано большое количество информационных систем. Тем не менее, эти модели данных имеют общепризнанный недостаток, который состоит в том, что представление данных в этих моделях является сильно зависимыми от способа их хранения. Следствием зависимости от способа хранения являются ограниченные возможности оптимизации запросов, трудности реализации распределенных систем, а также трудности определения и проверки ограничений целостности. Появление в 1970 году реляционной модели данных [2] стало ответом на потребность в модели данных, обеспечивающей независимость от способа хранения. В реляционной модели данных это было достигнуто через полагание в ее основу математических концептов более абстрактного уровня. Поэтому переход от сетевой и иерархической модели данных к реляционной — суть *повышения уровня абстракции.*

Направленность последующих работ в области развития моделей данных можно охарактеризовать стремлением *повысить уровень семантики.* Повышение уровня семантики означает расширение возможностей выражения свойств реального мира средствами модели данных. Одной из первых разработок в этом направлении можно является предложенная П. Ченом в 1976 году модель данных "сущность-связь" (ER) [3]. В основе ER модели данных лежат понятия *сущности*, которая представляет объекты реального мира; *бинарной связи* между сущностями, которая определяется свойства своих концов; и *атрибута* сущности. Повышение уровня семантики по сравнению с реляционной моделью осуществлялось, главным образом, за счет явного определения связей и их свойств. Важным свойством ER модели данных является определение графической нотации для лежащих в ее основе понятий, поэтому эту модель данных называют *ER диаграммами.* Так, например, сущности представляются в виде прямоугольника, связи - ромбом,

атрибуты - кругами, а связи между ними дугами. Отметим, что предложенная П. Ченом модель данных не является таковой в строгом смысле, поскольку она не содержала манипуляционной части. Позднее появились расширенные версии этой модели данных, среди которых есть и такие, которые можно считать достойными этого термина в строгом смысле [4, 5].

Другой вид моделей данных с повышенным уровнем семантики берет свое начало от разработки, предпринятой Дж. Смитом и Д. Смитом [6] и датируется 1977 годом. Предложенная Смитами модель данных была основана на базовых понятих абстракции данных: *обобщение (generalization)* и *агрегация (aggregation)*. Развитие подхода, основанного на использовании принципов абстракции данных, привело к появлению целого ряда моделей данных, за которыми закрепилось название *семантические*. Среди семантических моделей данных наибольшую известность приобрела модель данных Semantic Database Model (SDM) [7], разработанная М. Хамером и Д. МакЛеодом. SDM поддерживала полный набор фундаментальных видов абстракции применительно к атрибутированным сущностям. Список этих видов абстракции следующий:

- *Агрегация* представляет связь между заданными отдельными сущностями как некоторую единую сущность более высокого уровня. Заданные сущности при этом являются компонентами этой новой сущности. Эта связь может быть описана как "часть-целое".

- *Идентификация* позволяет поддерживать индивидуальность сущности в независимости от значения ее атрибутов.

- *Классификация* обеспечивает разбиение всех сущностей на подмножества таким образом, что каждая сущность попадает только в одно из подмножеств, и такие подмножества называют классом сущностей. Абстракции классификации соответствует связь "член-класс".

- *Обобщение* через связь "подкласс-суперкласс" позволяет рассматривать сущности некоторого класса (подкласса) как подмножество сущностей другого класса (супер класса).

- *Наследование атрибутов* обеспечивает наличие у сущности подкласса атрибутов супер класса.

Модель данных Смитов и SDM не являются моделями данных в строгом смысле, поскольку не содержат манипуляционной составляющей. Основанная на SDM модель данных, содержащая манипуляционную составляющую и средства определения ограничений целостности, была предложена в работе [8] в 1988 году. Манипуляционная составляющая этой модели была представлена декларативным языком манипулирования данных, который позволял запрашивать, модифицировать, добавлять и изменять сущности. Эта модель была реализована в промышленной СУБД SIM [8], разработанной компанией Unisys.

Отметим, что поддержка механизмов абстракции является основной, но не отличительной чертой семантической модели данных, поскольку в период появления этих моделей данных и позднее велись работы по расширению ER модели данных Чена поддержкой механизмов абстракции. Одной из наиболее известных расширений ER модели данных Чена является разработка Р. Баркера опубликованная в 1990 году [9]. Заимствование ER моделью данных принципов абстракции делает ее принципиально мало отличимой от семантических моделей данных.

Другим видом моделей данных с повышенным уровнем семантики является *функциональные модели данных*. Наиболее известной функциональной моделью является DAPLEX [10], разработанная Д. Шиманом в 1981 году. В отличие от большинства видов моделей данных с повышенным уровнем семантики функциональные модели данных основываются на минимальном числе понятий. Этих понятий два: сущность и функция. Функции могут быть однозначными и многозначными. Определение сущностей, их атрибутов и связей между ними в функциональной модели осуществляется через определение функций. Так, например, книга (BOOK) может быть определена следующим образом:

```
DECLARE BOOK() ==>> Entity
DECLARE Title(BOOK) ==> String
DECLARE Publisher(BOOK) ==> PUB_HOUSE
DECLARE Author(BOOK) ==> PERSON
```

В приведенном примере "==»" используется для определения многозначной функции, в то время как "==>" используется для определения однозначной функции. Как можно заключить из этого примера связи в функциональной модели также определяются через

функции. Например, функция Author представляет связь книги с ее авторами.

Функциональная модель данных не представляет явных средств абстракции данных, но пользователь может определить функции, реализующие механизмы абстракций. Например, функции без параметром возвращают все сущности некоторого вида, поэтому такие функции можно считать реализующими классификацию. Запросы в функциональной модели определяются в терминах вызовов функций. Большая выразительная мощность запросов достигается за счет возможности композиции функций.

Отметим, что функциональная модель данных по своей природе является моделью данных в строгом смысле, поскольку манипуляционная составляющая является ее неотъемлемой частью. Это объясняется ключевой ролью понятия функции, лежащей в основе манипуляционной составляющей.

Рассматривая модели данных с повышенным уровнем семантики невозможно не упомянуть объектно-ориентированные модели данных. Активное развитие этих моделей данных пришлось на вторую половину 80-х и первую половину 90-х гг. Несмотря на то, что основной целью разработки этих моделей данных было желание моделировать сложные структуры данных, концептуально объектно-ориентированные модели данных, с одной стороны, близки к семантическим моделям данных, поскольку поддерживают механизмы абстракции, с другой стороны, отражают поведенческий аспект моделируемого реального мира, что определяет их близость к функциональным моделям данных [1].

Остановимся на рассмотрении исследований и разработок середины 90-х с целью подведения итогов. В истории развития моделей данных было прослежено два перехода: повышение уровня абстракции и повышение уровня семантики. Мы рассмотрели более подробно модели данных с повышенным уровнем семантики. Было показано, что среди таких моделей существуют полноценные модели данных, которые можно взять за основу реализации системы баз данных. Отмечалось существование реализаций, в том числе и промышленных, например, система SIM компании Unisys.

[1] Фактически, именно наличие связи поведения с данными в объектно-ориентированных моделей данных является основной чертой, отличающей эти модели данных от семантических. В то же время расширение функциональной модели явной поддержкой механизмов абстракции данных можно считать приводящим к объектно-ориентированной модели данных. Примером этого является модель данных, положенная в основу системы IRIS [11]

С другой стороны можно смело констатировать тот факт, что системы баз данных, основанные на моделях данных с повышенным уровнем семантики, не получили широкого распространения. Рассмотрим причины такого итога. Таких причин, по мнению автора, три.

1. Из приведенного выше обзора можно сделать вывод, что период, в который разрабатывались модели данных с повышенным уровнем семантики, характеризуется появлением большого числа подходов к моделированию данных и их острой конкуренцией. Каждая из разработанных моделей претендовала на роль универсальной, при этом выразительные средства, положенные в их основу, фактически были ортогональны и дополняли друг друга. Сложившаяся ситуация исключала возможность прорыва одного из подходов на позиции лидера и закрепления за ним права называться стандартным, что способствовало рассеиванию усилий в доведении предложенных подходов до уровня промышленного использования.

2. На перспективы распространения моделей данных концептуального уровня значительное влияние оказывает наличие или отсутствие развитой методологии их использования в качестве инструмента проектирования баз данных при разработки ИС, а также степень интеграции такой методологии (и моделей данных) в общую методологию (и модельные средства) проектирования всех аспектов ИС. В период, когда предпринимались попытки реализации моделей данных с повышенным уровнем семантики, такие методологии еще только формировались и не достигли достаточного уровня зрелости.

3. При реализации всех упомянутых в обзоре систем баз данных, основанных на моделях данных с повышенным уровнем семантики, применялся подход построения с нуля. Такой подход требовал разработки большого числа новых методов, для которых не существовало прямых аналогов. Это приводило к необходимости больших инвестиций в разработку новой технологии, которая хотя и казалась многообещающей, но, как всякая новая технология, не могла гарантировать, что вложенные в нее разработку усилия окажутся оправданными.

1.1.3 Перспективы моделей данных с повышенным уровнем семантики

В заключение предыдущего раздела был приведен перечень факторов, которые, по мнению автора, вплоть до середины 90-х препятствовали появлению и распространению систем баз данных, основанных на моделях данных с повышенным уровнем семантики. Настоящий раздел посвящен обсуждению возможных путей устранения этих негативных факторов в условиях появления во второй половине 90-х новых перспективных технологий, таких как Unified Modeling Language (UML) и Extensible Markup Language (XML).

Ответом на трудности, вызванные первыми двумя факторами, может быть использование модели данных, основанной на языке UML. Обоснуем это утверждение.

К середине 90-х годов из всех описанных в предыдущем разделе подходов к моделированию данных продолжали использоваться только два: подход, основанный на использовании расширенных ER диаграмм, и подход, основанный на использовании объектно-ориентированных принципов моделирования. Оба подхода были представлены рядом моделей данных, основанных на них языков моделирования и методологий использования последних. Попытки прийти к стандартам в рамках первого подхода так и не увенчались успехом на момент написания настоящей работы, более того, в последние годы работа в этом направлении практически сошла на нет. В истории второго подхода прорыв произошел в 1997 году, когда компания Rational Software смогла сплотить разработчиков трех наиболее известных объектно-ориентированных языков моделирования: Грейди Буча и Джеймса Рамбо и Айвара Джекобсона. Эти разработчики ставили перед собой задачу создания унифицированного языка моделирования, который можно было бы принять за основу стандарта. Результатом их работы стал язык UML, который позднее был принят в качестве стандарта [41] консорциумом Object Management Group (OMG). Поддержка языка UML реализована в ряде продуктов, среди которых наиболее известным стал Rational Rose компании Rational Software. Кажется, можно утверждать, что к настоящему времени язык UML стал наиболее популярным и перспективным языком моделирования. В связи с этим, построение модели данных с повышенным уровнем семантики на основе языка UML

позволит добиться консенсуса, необходимого для распространения систем баз данных, основанных на этой модели. Таким образом, UML позволяет преодолеть трудности, вызванные первым негативным фактором.

Язык UML включает в себя набор подъязыков, с использованием которых можно моделировать большинство аспектов проектируемой ИС, включая и аспект управления данными. В этот набор в частности входят следующие языки:

- диаграмм вариантов использования;
- диаграмм последовательности;
- кооперативные диаграммы;
- диаграмм классов;
- диаграмм состояний;
- диаграмм компонентов;
- диаграмм размещения.

Заметим, что среди перечисленных языков находится язык диаграмм классов, который используется для моделирования данных. Существует ряд продуктов, поддерживающих проектирование схемы баз данных на языке UML и отображение разработанной схемы в среду целевой СУБД, как правило реляционной. При таком отображении фактически происходит переход от представления в модели данных концептуального уровня к представлению в модели данных уровня логического. Наиболее известным из таких продуктов является упоминавшийся выше Rational Rose.

Кроме того, перечисленный набор языков является достаточным для определения на его базе законченной методологии проектирования ИС[2]. Одним из примеров такой методологии является Rational Unified Process [43]. Эта методология поддерживает все аспекты разработки ИС: от выявления и фиксации требований до внедрения разработанной ИС. Таким образом, возможность использования языка UML для моделирования различных аспектов ИС при ее разработке, а также методологическая поддержка процесса разработки, позволяет, опираясь при разработке

[2]Стандарт языка UML определяет только сам язык и не фиксирует никакой методологии его использования при проектировании систем.

модели данных с повышенным уровнем семантики на язык UML, избежать трудностей, вызванных вторым негативным фактором.

Преодолеть трудности, вызванные третьим негативным фактором, поможет использование концепции многоуровневой архитектуры СУБД. Впервые в общем виде эта концепция была сформулирована и детально проработана в отчете ANSI/X3/SPARC [16]. При решении обсуждаемой проблемы имеет смысл опереться на несколько другой набор уровней, чем тот, который предложен в этом отчете. Рассмотрим архитектуру СУБД, состоящую из следующих уровней: концептуального, логического и физического. На каждом из этих уровней СУБД поддерживает некоторую модель данных. Нашей задачей является разработка СУБД, в архитектуре которой присутствует концептуальный уровень. Этого можно добиться разработкой надстройки, которая поддерживает модель данных концептуального уровня, над некоторой СУБД, основанной на модели логического уровня. Будем называть такую СУБД базовой. При использовании описанного подхода задача разработки необходимой СУБД сводится к выбору модели данных логического уровня, на которой будет основываться базовая СУБД, и разработке правила отображения необходимой модели данных концептуального уровня на модель данных базовой СУБД. При удачном выборе модели данных базовой СУБД правила отображения могут оказаться несложными, и, следовательно, реализация надстройки будет относительно не дорогостоящей, а сама надстройка достаточно эффективной. Если выбирать среди имеющихся на сегодня стандартных моделей данных, то в качестве модели данных базовой СУБД можно естественно использовать модель данных языка XML [35, 31], поскольку гибкость ее структурной составляющей и выразительность манипуляционной, позволяет строить несложные отображения на эту модель данных многих других моделей данных. Таким образом, опираясь на концепцию многоуровневой архитектуры СУБД и реализуя поддержку модели данных концептуального уровня в надстройке над СУБД, основанной на модели данных логического уровня, можно избежать трудностей, вызванных третьим негативным фактором.

Подведем итог выше сказанному. В этом разделе обсуждались перспективы подхода, суть которого заключается в использовании модельных принципов языка UML в качестве основы для разработки модели данных концептуального уровня (в строгом смысле термина модель данных), и способ реализации систем баз данных, поддерживающих такую

модель данных, основанный на отображении этой модели на модель данных XML. Намеченный подход и способ реализации развивается в главе 2.

1.2 Оптимизация запросов к XML-данным

1.2.1 Основные принципы оптимизации запросов и оптимизация на модельном уровне

На ранних этапах развития технологии баз данных модельно-языковые средства СУБД были сильно привязаны к физической организации данных. В то время единственной возможностью оптимизации запросов было использование методов оптимизации, близких к тем, которые применяются при компиляции программ, написанных на императивных языках. Появление моделей данных с повышенным уровнем абстракции от физических особенностей среды хранения (такие модели данных часто называют декларативными) привело к расширению пространства возможных способов выполнения запросов, что позволило применять принципиально новые методы оптимизации. К числу таких моделей данных относится и реляционная, которая сегодня является абсолютным лидером по числу практических применений основанных на ней реализаций.

Впервые серьезное внимание вопросам оптимизации запросов при построении систем реляционных баз данных было уделено в проекте System R [17, 18]. По результатам этого проекта сформировались основные принципы оптимизации запросов. Большинство из этих принципов активно используются в настоящее время, причем при разработке не только реляционных систем. Кратко рассмотрим эти принципы и основные соображения, которые привели к формированию этих принципов. Отправной идеей было представление оптимизации запросов как двухэтапного процесса. На первом этапе по запросу строилось *пространство поиска*, которое состояло из набора возможных *планов выполнения* запроса. Алгоритм построения пространства поиска называли *алгоритм перечисления*. На втором этапе к каждому плану выполнения, входящему в пространство поиска, применялась *оценочная функция*, которая на основании состояния базы данных вычисляла *стоимость выполнения плана*. Оптимальным планом считался план с

минимальной стоимостью. При этом стремились добиться того, чтобы (1) пространство поиска включало планы с наименьшей стоимостью, и (2) оценочная функция была точной. Оказалось, что реализация, удовлетворяющая этим требованиям, хотя и приводит к оптимальному (в строго математическом смысле) плану, тем не менее, не может быть использована на практике, поскольку время оптимизации запроса превышало для многих практически важных запросов время выполнения произвольно выбранного плана. Рассмотрим по каждому из пунктов, чем это было вызвано, и использование каких подходов позволило получить практически ценную реализацию оптимизатора.

Начнем с рассмотрения второго пункта. Точность оценочной функции не может быть достигнута без анализа состояния базы данных, что сводит на нет все усилия по оптимизации запросов, поскольку такой анализ почти эквивалентен выполнению плана. Решением проблемы стало использование статистических данных о состоянии базы данных, что позволило получить оценки близкие к точным за приемлемое время. Становление подхода к оценке запросов, основанного на статистике, было позднее закреплено в работе Г. Питецкого-Шапиро[22]. В этой работе были предложены законченные методы оценки, которые лежат в основе большинства современных реализаций оптимизаторов реляционных систем.

Теперь рассмотрим первый пункт. Для того, чтобы гарантировать, что пространство поиска содержит план с наименьшей стоимостью, в общем случае необходимо построить все возможные планы. Именно вследствие декларативности реляционной модели данных пространство всех возможных планов даже для относительно несложных запросов слишком велико, и, как следствие, алгоритм перечисления работает слишком долго. Необходимой мерой борьбы с неприемлемо большими затратами времени на построение пространства поиска стало полагание в основу алгоритма перечисления набора эвристик, которые позволили сузить пространство поиска. Набор эвристик подбирался таким образом, чтобы для большинства практически важных запросов пространство поиска содержало оптимальный план.

Таким образом, при помощи описанных подходов по каждому из пунктов был достигнут компромисс между временем выполнения оптимизации и гарантированностью получения оптимального плана.

Остановимся более подробно на алгоритмах перечисления и

рассмотрим принципы работы этих алгоритмов, сложившиеся на сегодняшний день [19, 20, 21]. Как было сказано выше, задачей этого алгоритма является построение на основе эвристик пространства поиска, состоящее из планов выполнения оптимизируемого запроса. Эвристические методы, определяющие работу алгоритма перечисления, обычно делят на два класса. Основная мотивация такого разделения состоит во введении промежуточного уровня представления запроса, называемого *логическим представлением.* Логическое представление выражается через операции, определенные в терминах структурной составляющей модели данных. Таким образом, к первому классу относят методы, которые по логическому представлению запроса строят одно или несколько эквивалентных логических представлений. Ко второму классу относят методы, которые по заданному логическому представлению строят набор реализующих его планов выполнения, которые подлежат оценке с целью выбра оптимального. Оказывается, что большую и наиболее сложную часть работы по построению пространства поиска, связанную с трансформациями, можно провести в терминах логического представления, что существенно облегчает разработку алгоритма перечисления, благодаря возможности абстрагироваться от многочисленных особенностей реализации. Отметим, что трансляция логического представления во множество планов выполнения является относительно несложной задачей, поскольку такой переход обычно обладает свойством локальности, которое состоит в том, что каждая операция логического представления транслируется без учета контекста ее нахождения в представлении.

Рассмотрим виды трансформации логического представления, применяемые при построении пространства поиска.

- *Опустить предикаты.* Название этого вида трансформации следует воспринимать в терминах графовой интерпретации логического представления как дерева операций, при этом ориентация дерева в пространстве такова, что при обходе дерева от корня к листьям происходит движение сверху вниз (то есть корень выше листьев). "Опустить предикат" означает изменить порядок операций таким образом, чтобы применить его как можно раньше. Раннее применение предиката уменьшает размер результатов промежуточных вычислений, что приводит к уменьшению времени выполнения запроса.

- *Опустить проекцию.* Этот вид оптимизации аналогичен предыдущему с той разницей, что при трансформации вместо операции применения предиката участвует операция проекции. Такая трансформация позволяет не только уменьшить размер результатов промежуточных вычислений, но и избежать избыточных вычислений в том случае, если в результате проекции исключаются данные, вычисляемые динамически в процессе выполнения запроса.

- *Упрощение представления.* Под упрощением представления понимается, главным образом, вычисление подвыражений на этапе оптимизации без доступа к данным, если это возможно.

- *Повысить уровень декларативности представления запроса.* Несмотря на то, что речь идет об оптимизации декларативных запросов, уровни декларативности логических представлений запроса могут различаться. Например, вложенный подзапрос, зависящий от переменных внешнего запроса, может быть эквивалентен композиции операций проекции и соединения. Исходное представление запроса навязывает план выполнения запроса методом вложенных циклов, в то время как эквивалентное представление допускает и другие методы выполнения, например сортировки и слияния. Таким образом, этот вид трансформации направлен на расширение пространства поиска планами, среди которых может оказаться оптимальный. [23] - первая работа, в которой предлагался такой вид трансформации для оптимизации SQL-запросов. Описание наиболее развитых методов осуществления таких трансформаций можно найти в работе [24]. Отметим, что повышение уровня декларативности логического представления запросов потребовало расширения операций логического представления новыми декларативными операциями, такими, например, как полусоединение или расширенное внешнее соединение.

- *Изменение порядка операций.* Этот вид трансформации содержит обширный набор преобразований, опирающихся на свойства коммутативности и ассоциативности операций логического представления. К наиболее важным преобразованиям такого вида можно отнести изменение порядка операций соединения [25, 26] или изменение порядка операций соединения и группировки [27]. Важно отметить, что построение всех возможных порядков

операций, например, в случае перестановки операций соединения, приводит к чрезмерному расширению пространства поиска, что влечет за собой неприемлемо большие временные затраты на выбор оптимального плана. Поэтому для этого вида трансформации характерно использование эвристик, направленных на построение только наиболее обещающих порядков операций.

Все перечисленные виды трансформации, кроме последнего, являются однозначными, то есть приводят к единственному логическому представлению, и, следовательно, предполагается, что они "не ухудшают" представление запроса. Отметим, что это, вообще говоря, не верно [28], но практика использования оптимизаторов говорит о том, что такие виды преобразований полезны для большинства практически важных запросов [19, 20, 21] и помогают добиться увеличения производительности систем на порядки.

В настоящее время на арене технологий баз данных появилась новая модель данных XML[35], а также определенный над этой моделью данных декларативный язык запросов XQuery [31]. Эта модель данных и язык запросов предназначены для управления XML-данными [36]. Построение эффективных систем управления XML-данными невозможно без разработки развитых средств оптимизации запросов. Хотя описанный выше подход к оптимизации запросов был развит, главным образом, при построении реляционных систем, естественно предположить, что лежащие в его основе базовые принципы могут быть взяты за основу при разработке методов оптимизации запросов для других декларативных модельно-языковых средств, таких, например, как XML модель данных и язык XQuery. Тогда в рамках описанного подхода основную сложность при разработке оптимизатора составляет разработка алгоритма перечисления и оценочной функции, причем ключевым моментом при разработке алгоритма перечисления является трансформация логических представлений запроса. Третья глава данной работы посвящена описанию разработанных автором общего подхода к трансформации логических представлений XQuery-запросов, а также методов в рамках этого подхода. Поскольку язык XQuery является относительно новым (первая версия спецификации этого языка появилась только в 2000 году), представляется необходимым дать общую характеристику этого языка и связанных с ним модельных средств. Следующий раздел содержит такую характеристику. В разделе, следующем после описания общей характеристики языка

XQuery, приводятся результаты анализа того, в какой мере применимы перечисленные выше виды трансформации реляционных запросов к запросам на языке XQuery, и какие дополнительные виды трансформаций могут быть полезны при оптимизации запросов на этом языке. В заключительном разделе настоящей главы производится краткий обзор существующих работ по оптимизации XQuery-запросов.

1.2.2 Модель данных XML и язык запросов XQuery

Популярность языка XML во многом определяется его гибкостью при представлении разнородных данных, то есть данных, имеющих сложную и/или нерегулярную структуру. Эти свойства языка XML приводят к попыткам решить на его основе следующие задачи управления данными:

- построение систем интеграции разнородных данных;
- построение систем управления данными, которые имеют полуструктурированную природу, или данными, обладающими сложной структурой.

Обязательным требованием, выдвигаемым к таким системам, является поддержка декларативных языков запросов. В ответ на это требование появился ряд языков запросов к XML-данным, среди которых наибольшую известность приобрели следующие: XQL [15], XML-QL [14] и Lorel [12]. Эти языки имели взаимные преимущества при определении различных видов запросов, и ни один из них не мог претендовать на роль универсального языка запросов XML-данных, достойного стать стандартом. Ситуация изменилась с появлением в 1999 году языка Quilt [30], который впоследствии был доработан и переименован в XQuery. XQuery вобрал в себя лучшее из перечисленных выше языков и покрывает своими возможностями большинство практически важных видов запросов. Работа над языком XQuery ведется концорциумом World Wide Web (W3C) и ожидается скорое приобретение этим языком статуса рекомендации (наивысшего статуса консорциума W3C). Этот язык считается сегодня наиболее перспективным среди языков запросов к XML-данным, поэтому большинство исследований и разработок последних двух лет направлены на разработку методов реализации именно этого языка [42, 44, 45, 46]. В следующих подразделах содержится краткое описание языка XQuery и

модели данных, лежащей в его основе[3].

Модель данных XML

Язык XQuery основан на модели данных XML, определенной в спецификации [35]. Эта модель данных определяет абстрактное представление одного или нескольких XML-документов или их фрагментов. Основополагающим понятием этой модели данных является понятие последовательности. Для того чтобы избежать путаницы при использовании термина последовательность в общем смысле и в смысле, определенном в модели данных XML, будем использовать для обозначения последнего смысла термин x-последовательность. *X-последовательность* (sequence) — это упорядоченный набор нулевого или большего числа объектов. Объект (item) может быть узлом или атомарным значением. *Атомарное значение* (atomic value) — экземпляр одного из встроенных типов данных, определенных во второй части спецификации "XML Schema" [38], таких как строки, целые числа, десятичные числа и даты. *Узел* (node) соответствует одному из шести видов: элементы, атрибуты, тексты, документы, комментарии и команды обработки. Узел, относящийся к виду "элементы", может иметь другие узлы в качестве потомков, что позволяет образовывать иерархию узлов. Элементы и атрибуты имеют имена и типизированные значения. *Типизированное значение* (typed value) - это последовательность из нуля или большего числа атомарных значений. Узлы имеют *уникальные идентификаторы*, которые определяют их индивидуальность (т.е. два узла можно различить, даже если они имеют одинаковые имена и значения), но атомарные значения такой индивидуальностью не обладают. Для всех узлов иерархии имеется глобальный порядок, называемый *порядком документа* (document order), в соответствии с которым каждый узел предшествует своему потомку, и каждый узел предшествует своему правому брату в XML-дереве. Порядок документа соответствует порядку, в котором следовали бы узлы, если бы иерархия узлов представлялась в формате XML. Порядок документа между узлами в разных иерархиях определяется в реализации, но он должен быть последовательным, т.е. все узлы одной иерархии должны располагаться либо до, либо после всех узлов другой

[3]Отметим, что термин "модель данных" используется здесь как результат непосредственного перевода названия спецификации консорциума W3C, в которой описана эта модель ("XQuery и XPath Data Model"). На самом деле, вопреки сложившейся терминологии баз данных (см. раздел 1.1.1), эта модель содержит только структурную составляющую.

иерархии. Глобальный порядок документа и локальный порядок узлов в некоторой x-последовательности могут быть различными.

X-последовательности могут быть *неоднородными*, т.е. могут содержать смесь узлов и атомарных значений разного типа. Однако x-последовательность никогда не может быть элементом другой x-последовательности. Все операции, создающие x-последовательность, определены так, что результат операции — одноуровневая x-последовательность. Не проводится различие между объектом и x-последовательностью единичной длины, т.е. узел или атомарное значение считаются идентичными x-последовательности единичной длины, содержащей этот узел или атомарное значение.

Допускаются *x-последовательности нулевой длины*, и иногда они используются для представления отсутствующей или неизвестной информации, во многом так же, как в реляционных системах используются неопределенные значения.

Помимо x-последовательностей в модели данных XML определяется специальное значение, называемое *значением ошибки* (error value), которое является результатом вычисления выражения, содержащего ошибку. Значение ошибки не может присутствовать в x-последовательности вместе с каким-либо другим значением.

XML-документы могут быть преобразованы в термины модели данных XML с помощью процесса, называемого *проверкой корректности по схеме* (schema validation). Этот процесс выполняет грамматический разбор документа, проверяет его корректность в соответствии с некоторой схемой и представляет документ в виде иерархии узлов и атомарных значений, помеченных типом, полученным из схемы. Если входной документ не имеет схемы, проверка его корректности выполняется в соответствии с используемой по умолчанию рекомендательной схемой, которая присваивает родовые типы — узлы маркируются как anyType, а атомарные величины как anySimpleType.

Результат запроса может быть преобразован из модели данных XML в XML-представление с помощью процесса, называемого *сериализацией* (serialization). Следует отметить, что результат запроса не всегда является правильно построенным XML-документом. Например, запрос может возвращать атомарное значение или x-последовательность элементов, не имеющих общего предка.

Язык XQuery

XQuery — функциональный язык, состоящий из нескольких видов выражений, из которых могут строиться произвольные композиции. Рассмотрим виды XQuery выражений.

Выражение пути. Выражение пути в XQuery базируется на синтаксисе языка XPath [32]. Выражение пути состоит из серии шагов, разделенных символом слэш (“/”). Результат каждого шага - x-последовательность узлов. Значение выражения пути - x-последовательность узлов, которая формируется на последнем шаге. Шаг определяется тремя компонентами: *осью* (axis), *тестом* (test) и *предикатом* (predicate). Синтаксис записи шага следующий: `<ось>::<тест>[<предикат>]`. Каждый шаг вычисляется в *контексте*, называемом также динамическим контекстом. Опишем правила формирования контекста для вычисления очередного шага и правила вычисления самого шага. Результатом вычисления шага над x-последовательностью, которую назовем A, является объединение результатов вычисления шагов для каждого узла из A с удалением дубликатов на основании уникальных идентификаторов и последующей сортировкой в порядке документа. Вычисление шага для узла (будем называть этот узел *текущим*) осуществляется в несколько этапов, на каждом из которых строится промежуточная x-последовательность. На первом этапе происходит перемещение от текущего узла по иерархии узлов в направлении, определяемом осью. В XQuery поддерживается шесть осей: `child` (x-последовательность дочерних узлов), `descendant` (x-последовательность всех узлов потомков), `parent` (отец узла), `attribute` (x-последовательность узлов-атрибутов), `self` (текущий узел) и `descendant-or-self` (x-последовательность всех потомков, дополненная самим узлом). Второй этап состоит в исключении из x-последовательности, полученной на первом этапе, узлов, не удовлетворяющих тесту. Тестом в XQuery может быть имя узла или символ “*”, обозначающий любое имя. Проверка на удовлетворение тесту состоит в сравнении имени узла с тестом. На третьем этапе происходит фильтрация x-последовательности узлов, полученных на предыдущем этапе, на основании результата вычисления предиката для каждого узла. При этом предикат вычисляется в контексте, который состоит из следующих компонентов: контекстного узла (узла для которого вычисляется предикат), его позиции в x-последовательности и количества элементов x-последовательности.

Доступ из предиката к компонентам контекста осуществляется через вызов предопределенных функций `fn:context-item()`, `fn:position()` и `fn:last()` соответственно. Доступ к контекстному узлу можно также получить, перемещаясь по оси `self`. Приведем пример выражения пути, результатом которого является x-последовательность описания всех товаров, предлагаемых к продаже Смитом:

```
document("items.xml")/child::*/
child::item[self::*/child::seller="Smith"]
/child::description
```

Другой пример, выбрать пятый товар:

```
document("items.xml")/child::*/
child::item[fn:position()=5]
```

Существует сокращенная форма записи выражений пути, в которой ось `child` можно не указывать; ось `descendant` опускается и перед этим шагом ставится не один, а два слеша; переход по оси `parent` записывается через две точки, при этом предполагается, что тестом будет `*`; вместо оси `attribute` ставится символ `@`. Кроме того, шаг с осью `self` в предикате можно опустить. Пример выражения пути в сокращенной форме записи:

```
document("items.xml")//item[seller="Smith"]/
description/../@status
```

Операции сравнения значений. Операции `eq`, `ne`, `lt`, `le`, `gt`, `ge` сравнивают два атомарных значения и возвращают ошибку, если любой из операндов является x-последовательностью, состоящей из более чем одного элемента. Если один из операндов узел, то прежде, чем выполнить сравнение, операция сравнения значений извлекает его значение.

Операции сравнения узлов. Операции `is` и `notis` определяют идентичность двух узлов на основании сравнения уникальных идентификаторов. Например, `$node1 is $node2` принимает значение `true`, если переменные `$node1` и `$node2` связаны с одним и тем же узлом.

Логические операции. XQuery поддерживает следующие логические операции: `and`, `or`, `not`.

Арифметические операции и агрегатные функции. XQuery поддерживает традиционный набор арифметических операций: `+`, `-`, `*`, `div` и `mod`, а также набор агрегатных функций: `sum`, `avg`, `count`, `max` и

`min`, которые применяются к x-последовательности чисел и возвращают числовой результат.

Переменные и связывание. Переменная (variable) в XQuery — имя, начинающееся со знака доллар ("$"). Переменная может быть связана со значением и использоваться в выражении для представления этого значения. Один из способов связывания состоит в использовании LET-выражения, которое позволяет связать переменную со значением выражения. LET-выражение - частный случай выражения FLWR (сокращение от первых букв for, let, where, return), которое обеспечивает дополнительные способы связывания. Следующий пример выражения возвращает x-последовательность (1,2,3).

```
let $a:=1, $b:=2, $c:=3
return ($a,$b,$c)
```

Конструкторы узлов. XQuery обеспечивает средства трансформации через поддержку конструкторов элементов и атрибутов. Конструкторы бывают обычными и вычисляемыми. Для них поддерживаются разные формы записи. Вычисляемые конструкторы являются более общим видом конструкторов. При использовании обычного конструктора создаваемый XML-элемент записывается в соответствии с синтаксисом XML, за исключением того, что выражения, заключенные в фигурные скобки, которые могут задавать только значение элементов и атрибутов, вычисляются, а не трактуются как часть XML-текста. Приведем пример.

```
<highbid status="{$s}">
   <itemno>{$i}</itemno>
   <bid-amount>
   {max($bids[itemno=$i]/bid-amount)}
   </bid-amount>
</highbid>
```

Вычисляемые конструкторы отличаются от обычных тем, что выражения можно использовать для задания не только значений элементов и атрибутов, но и их имен. Для записи вычисляемых конструкторов используются ключевые слова `elemenet` или `attribute`, за которыми следуют два выражения в фигурных скобках - первое определяет имя элемента или атрибута, а второе - его значение. Приведем пример.

```
element {name($e)} {$e/@*,$e/price*2}
```

Итерация. XQuery позволяет выполнять итерацию над x-последовательностью объектов, по очереди связывая переменную с каждым значением и вычисляя выражение для каждого связывания переменной. Результат итерации соответствует порядку элементов в x-последовательности, по которой происходит итерация. Итерация задается оператором `for`. Результатом следующего примера является x-последовательность `(3,4)`.

```
for $n in (2,3) return $n+1
```

В операторе `for` можно указывать более одной переменной. Такие выражения рассматриваются как вложенные итераторы, где порядок записи связываний переменных определяет вложенность итераторов. Самое левое связывание соответствует самому внешнему циклу итерации.

Операторы `for` и `let` - частные случаи выражения FLWR. В наиболее общем виде выражение FLWR состоит из произвольной последовательности `for` и `let` выражений, за которыми следует необязательный оператор `where`, а затем оператор `return`. X-последовательность операторов определяет область видимости связанных переменных: выражение может содержать переменные, для которых связывание было выполнено в предыдущих операторах. Общее правило вычисления FLWR выражения следующее. Операторы `for` и `let` производят связывание переменных в соответствии с описанными ранее правилами. Для каждого набора связываний вычисляется выражение, находящееся в операторе `where`. Для тех наборов связываний, для которых значение этого выражения истинно, вычисляется выражение в операторе `return`. Результаты вычислений выражений в операторе `return`, составляют результирующую x-последовательность. Приведем пример выражения FLWR, результатом которого является x-последовательность из элементов `popular-item`, каждый из которых содержит номер товара и описание. Элемент `popular-item` создается для каждого товара, который имеет более десяти ставок.

```
for $i in document("items.xml")/*/item
let $b:=document("bids.xml")/*/bid[itemno=$i/itemno]
where count($b)>10
return
<popular-item>
   {$i/itemno,$i/description}
```

```
</popular-item>
```

По умолчанию, порядок элементов в результирующей x-последовательности при вычислении выражения FLWR соответствует порядку элементов в x-последовательностях, по которым производятся итерации. Перед выражением FLWR можно поставить префиксный оператор `unordered`, указывающий, что порядок результата не имеет значения. Такое указание повышает гибкость реализации, позволяя оптимизировать вычисление выражения FLWR.

Операции над x-последовательностями, требующие поддержки уникальных идентификаторов. К таким операциям в языке XQuery относятся `intersect`, `union` и `except`. Операция `intersect` порождает x-последовательность, в которую включены все узлы, имеющиеся в обоих операндах. Операция `except` позволяет получить x-последовательность, которая содержит все узлы, которые есть в первом операнде, но отсутствуют во втором. Операция `union` объединяет x-последовательности, являющиеся ее операндами. Важно отметить, что в приведенных определениях предполагается, что сравнение узлов производится через сравнение их уникальных идентификаторов.

Кванторные выражения. Кванторные выражения позволяют проверить некоторое условие, устанавливая, истинно ли оно для хотя бы одного элемента x-последовательности (квантор существования) или для всех элементов x-последовательности (квантор всеобщности). Результатом всегда является `true` или `false`. Подобно FLWR-выражению, при вычислении значения кванторного выражения происходит поочередное связывание переменной с каждым элементом x-последовательности. Для каждого связывания переменной вычисляется проверочное выражение. Ключевое слово `some` используется в кванторном выражении для обозначения квантора существования. Ключевое слово `every` используется в кванторном выражении для обозначения квантора всеобщности. Продемонстрируем общую форму записи кванторных выражений на примере квантора существования (квантор всеобщности может быть записан заменой ключевого слова `some` на ключевое слово `every`):

```
some $n in (5,7,9,11) satisfies $n>10
```

Функции в XQuery. В XQuery предусмотрена библиотека предопределенных функций [34]. Кроме того, XQuery предоставляет возможность определять пользователям их собственные функции

на самом языке XQuery. При определении функции пользователь указывает имя функции, типы и имена формальных параметров и тип возвращаемого функцией значения. В следующем примере определена функция `highbid`, параметром которой является XML-элемент (`element`). Тип возвращаемого функцией значения - десятичное число (`decimal`). Функция интерпретирует свой параметр как элемент `item` и извлекает номер товара. Затем она находит и возвращает самую крупную ставку (`bid-amount`), которая была зафиксирована для товара с этим номером.

```
define function highbid(element $item) returns decimal
{
max(document("bids.xml")//bid[itemno=$item/itemno]
/bid-amount)
}
highbid(document("bids.xml")//bid[itemno="1234"])
```

Система типов. Система типов языка XQuery определена в спецификациях [37, 38]. В эту систему типов входит набор предопределенных типов таких, например, как `integer` или `decimal`, а также типы, определенные в схеме адресованных запросом данных. При создании запроса иногда необходимо сослаться на некоторый тип. Например, как уже было отмечено, при определении функции требуется описать типы параметров функции и ее результата. Для ссылки на предопределенный тип необходимо просто указать его имя, например, `integer`. Для ссылки на другие типы используются родовые ключевые слова. Ключевое слово element обозначает любой элемент, ключевое слово `attribute` обозначает любой атрибут, `node` - любой узел, а `item` - любой объект (то есть узул или атомарное значение). За ключевыми словами `element` и `attribute` может следовать имя, которое в большей степени ограничивает тип узла. Например, `element ship` обозначает не любой элемент, а элемент с именем `ship`. За ссылкой на тип может следовать один из трех индикаторов присутствия: "*" означает "ноль или больше", "+" означает "один или больше", а "?" означает "ноль или один". Проиллюстрируем использование индикаторов присутствия.

```
element memo?  -  означает возможное появление элемента
с именем memo
element*  -  означает любое число любых XML-элементов
```

Правила вывода типов и процесс проверки корректности XQuery-запроса. Правила проверки корректности XQuery-запросов по схеме адресуемых этим запросом данных определены в [31, 33]. Основную часть этих правил составляют правила *механизма вывода типов*. Механизм вывода типов позволяет по имеющемуся запросу и схеме адресованных этим запросом данных определить схему получаемых в результате запроса данных. Наличие таких правил позволяет считать язык XQuery статически типизируемым, однако заметим, что применение механизма вывода типов ко многим запросам может дать наиболее общий тип anyType, что сводит на нет в общем случае преимущества статической типизации языка XQuery.

1.2.3 Предварительное исследование возможностей логической оптимизации XQuery-запросов

Этот раздел посвящен исследованию возможностей логической оптимизации. При проведении этого анализа мы будем отталкиваться от видов трансформации, применяемых при оптимизации реляционных запросов. Эти виды трансформации перечислены на странице 20. Мы не ограничимся только этими видами трансформации и рассмотрим также трансформации, характерные для оптимизации XQuery-запросов. Исследование будем проводить на примерах. Для записи примеров будем использовать следующую форму: "запрос-1 => запрос-2", где стрелка "=>" будет означать трансформацию запроса-1 в запрос-2.

```
for $i in document("items.xml")/items/item,
    $b in document("bids.xml")/bids/bid
where
      $i/itemno=$b/itemno
            and
      $i/preserve-price>100
            and
      $b/bid-amount>200
return ($i,$b)
=>
for $i in document("items.xml")/items/
          item[preserve-price>100],
    $b in document("bids.xml")/bids/bid[bid-amount>200]
```

```
where $i/itemno=$b/itemno
return ($i,$b)
```

Применение такой трансформации приводит к *опусканию предикатов* `preserve-price>100` и `bid-amount>200` в выражения, определяющие x-последовательности, по которым будет проводится итерация переменных `$i` и `$b`. Очевидно, что такая трансформация не может увеличить время выполнения запроса. В том же случае, когда селективность опущенных предикатов будет невысокой, такая трансформация может существенно уменьшить время выполнения запроса, поскольку уменьшается количество связываний при итерациях переменных `$i` и `$b`. Этот вид трансформации аналогичен одноименному виду, применяемому при оптимизации реляционных запросов.

Другим полезным видом опускания предиката является трансформация, при которой применение предиката осуществляется по возможности до вычисления конструкторов. Рассмотрим следующий пример.

```
for $i1 in
(for $i in document("items.xml")/items/item
return
<item>
    <seller>{$i/seller}</seller>
    <price>{$i/reserve-price*2}</price>
</item>)
where $i1/seller="Smith"
return $i1
=>
for $i in document("items.xml")/items/
          item[seller="Smith"]
return
<item>
    <seller>{$i/seller}</seller>
    <price>{$i/reserve-price*2}</price>
</item>
```

В этом примере в результате трансформации получаем запрос, при вычислении которого конструкторы XML-элементов будут вычисляться возможно меньшее число раз (на сколько меньшее зависит от

селективности предиката `seller="Smith"`), чем в исходном запросе. Напомним, что при вычислении конструктора осуществляется глубокое копирование всех вложенных XML-элементов с генерацией для них новых уникальных идентификаторов, что делает конструктор очень дорогостоящей операцией. Поэтому уменьшение числа вычислений конструкторов, как в рассмотренном только что примере, является очень существенной оптимизацией.

Перейдем к следующему примеру.

```
(for $b in document("bids.xml")/bids/bid
return
<bid>
    <seller>
    {document("items.xml")/items/item[itemno=$b/itemno]/
    seller/text()}

    </seller>
    {$b/bidder}
</bid>)/bidder
=>
document("bids.xml")/bids/bid/bidder
```

Этот пример трансформации можно считать аналогичным *опусканию проекции* при оптимизации реляционных запросов. Очевидно, что применение такой трансформации позволит существенно сократить время выполнения запроса, поскольку не вычисляется ни только дорогостоящие операции конструирования XML-элементов, но и подзапрос:

```
document("items.xml")/items/item[itemno=$b/itemno]/
seller/text()
```

Будем исходить из предположения, что если какое-либо подвыражение можно вычислить в статике, то это следует сделать. Поскольку в общем случае вычисление в статике сопоставимо по затрате вычислительных ресурсов вычислению в динамике, но бывают случаи, когда вычисление в статике оптимизирует выполнение запроса. Например, если вычисляемое подвыражение находится в операторе return, то упрощение этого подвыражения путем его вычисления приведет к уменьшению времени выполнения запроса, поскольку это подвыражение не будет многократно

вычисляться в динамике при итерации. Рассмотрим подтверждающий это пример трансформации.

```
for $b in document("bids.xml")/bids/*
return
   element {name($b)} {$b/*}
=>
for $b in document("bids.xml")/bids/bid
return
   element {"bid"} {$b/*}
```

Такая трансформация возможна в том случае, если применение механизма вывода типов к подзапросу `document("bids.xml")/bids/*` покажет, что типом возвращаемого результата является `element bid`. В этом случае, типом переменной `$b` является `element bid`, и, следовательно, значение подвыражения name($b) равно "bid".

Следующий пример демонстрирует трансформацию, направленную на *повышение уровня декларативности запроса.*

```
unordered for $b in document("bids.xml")/bids/bid
where
   some $i in
      document("items.xml")/items/item[itemno=$b/itemno]
   satisfies $i/seller="Smith"
return $b

=>
fn:distinct-nodes(
for $b in document("bids.xml")/bids/bid,
    $i in document("items.xml")/items/item[seller="Smith"]
where $i/itemno=$b/itemno
return $b)
```

Этот пример особенно интересен тем, что наглядно показывает нехватку выразительных средств языка XQuery для представления

оптимизированного запроса в виде, из которого нетрудно извлечь преимущества оптимизации. Действительно, в приведенном примере трансформации фактически происходит переход от вложенного подзапроса к операции соединения, но поскольку такой операции нет в языке XQuery, то по полученному запросу будет трудно это увидеть. Таким образом, этот пример демонстрирует необходимость осуществления оптимизации в терминах расширенного, по отношению к языку XQuery, набора операций.

Наличие в запросе вызовов XQuery функций определяемых пользователем может существенно усложнить оптимизацию запроса. Это показывает следующий пример фрагмента запроса.

```
define function output(element $s) returns anyType
{if node-kind($s)="text"
 then $s
 else
    element {name($s)}
    {$s/@*,
    for $k in $s/*
    return if(name($k)="reserve-price")
           then <price>{$k/node()}</price>
           else output($k)}}

for $i in output(document(items.xml)/items)/items/item
where $i/price/node()>20
return $i
```

Если функция `output` будет вызываться из нескольких позиций в запросе (например, когда есть вызовы и за пределами этого фрагмента), то в приведенной формулировке в принципе невозможно опустить предикат `$i/price/node()>20` ниже конструкторов, находящихся в теле функции. Это привело бы к изменению тела функции, а, следовательно, и результатов вызова этой функции из других позиций в запросе. Эффективность оптимизации запросов, содержащих вызовы XQuery-функций, определяемых пользователем, может быть повышена посредством осуществления трансформации запросов, при которых происходит открытая вставка тел функций.

Рассмотренные выше примеры позволяют сделать следующее основное заключение. Применяя методы логической оптимизации к XQuery-запросам, можно добиться существенного уменьшения времени их выполнения.

1.2.4 Обзор работ по оптимизации XQuery-запросов

В связи с тем, что язык XQuery появился всего несколько лет назад, существует небольшое число работ, посвященных логической оптимизации запросов на этом языке. В этих работах затрагиваются только некоторые аспекты логической оптимизации, причем преимущественно на уровне идей, и не одна из этих работ не предлагает законченного подхода к решению этой задачи.

В работе [42] приводится "предположительный и не полный" набор правил трансформации XQuery-запросов в терминах предлагаемой в статье XQuery-алгебры. Приведенный набор правил ориентирован, главным образом, на уменьшение промежуточных результатов вычисления XQuery-запросов. Наиболее разработанным на сегодня подвидом логической оптимизации XQuery-запросов можно считать семантическую оптимизацию. К этому подвиду логической оптимизации относят правила трансформации запросов с использованием информации из схемы адресуемых запросов данных. В работе [44] приводятся примеры использования такой трансформации с целью упрощения XQuery-запроса. В работе [45] приводятся примеры различных видов трансформации XQuery-запросов, среди которых: семантическая оптимизация, опускание предиката, переупорядочивание операций соединения, избавление от избыточных операций сортировки в порядке документа. Кроме того, стоит отметить работу [46]. В этой работе описаны правила нормализации XQuery-запросов к виду, более удобному для трансляции в запросы на языке SQL. Такая трансляция используется при построении систем интеграции данных с поддержкой языка XQuery. Приведенные в этой работе правила могут быть полезны при оптимизации XQuery-запросов.

1.3 Выводы

Современные тенденции развития информационных технологий, такие как широкое использование языков моделирования концептуального

уровня для спецификации схем баз данных и потребность в средствах, облегчающих разработку приложений для использования баз данных конечными пользователями, создают предпосылки к разработке модельно-языковых средств доступа к данным в терминах концептуальных схем. Перспективным походом к разработке и реализации таких средств является выбор языка UML в качестве основы для разработки полноценной модели данных концептуального уровня и использование модели данных XML для отображения на нее модели данных концептуального уровня с целью получения недорогой и эффективной реализации последней. При таком подходе к реализации модели данных концептуального уровня требуются эффективные методы поддержки модельно-языковых средств семейства XML. Ключевым средством из этого семейства является язык запросов XQuery. По результатам проведенных автором исследований можно сделать вывод, что существенного повышения эффективности обработки запросов на языке XQuery можно добиться при помощи методов логической оптимизации, аналогичных тем, которые используются для оптимизации запросов к реляционным базам данных.

Глава 2

Декларативный язык запросов данных в терминах UML

Настоящая глава посвящена описанию разработанного автором языка запросов UML Query Language (UQL), который позволяет формулировать запросы к данным в терминах диаграмм классов UML, а также метода реализации этого языка через трансляцию UQL-запросов в запросы на языке XQuery.

2.1 О необходимости разработки языка запросов к данным в терминах UML

На сегодняшний день широкое распространение получило использование высокоуровневых языков моделирования при разработке информационных систем (ИС). Одним из лидеров среди таких языков является UML (Unified Modeling Language) [41]. Описание на языке UML будем называть UML-моделью. В последнее время UML все чаще используется не только для описания функциональных требований и архитектуры системы, но и для определения концептуальной схемы данных, с которыми работает ИС. Для определения концептуальной схемы данных обычно используется подмножество языка UML, называемое диаграммами классов[1]. Определенная на этом подмножестве UML-модель рассматривается как концептуальная схема данных. При этом данные, соответствующие UML-модели (концептуальная схема), мы будем называть UML-данными.

При использовании языков моделирования (в частности, языка UML)

[1]Далее в этой главе будем рассматривать только это подмножество языка UML и, соответственно, под UML-моделью будем понимать некоторую диаграмму классов, если это не будет оговорено особо.

для определения концептуальной схемы данных характерен следующий сценарий разработки ИС: разрабатывается модель предметной области, которая описывается на языке UML; из этой модели выделяется часть, которую рассматривают как концептуальную схему управляемых данных (как правило, эта часть описана с использованием подмножества языка UML - диаграмм классов); по концептуальной схеме автоматически (или полуавтоматически под управлением разработчика) генерируется логическая схема данных в терминах модели данных, поддерживаемой целевой СУБД, которую предполагается использовать для управления данными; затем, при реализации компонентов ИС, работающих с данными, используются средства доступа к данным в терминах логической схемы, предоставляемые этой СУБД. При таком сценарии разработки ИС, если необходимо предоставить пользователю доступ к данным в терминах предметной области, требуется обратный переход от логической схемы к концептуальной. Этот переход реализуется при разработке компонентов, работающих с данными. На сегодняшний день не существует средств, позволяющих каким-либо образом облегчить реализацию такого перехода. Как следствие, для осуществления последнего этапа приведенного выше сценария разработки требуется большой объем ручного программирования со всеми его издержками. Эту проблему можно решить путем поддержки на системном уровне доступа к данным в терминах предметной области (то есть в терминах концептуальной схемы описанной на UML). В этой главе описываются предлагаемые автором модельно-языковые средства и способ их реализации для поддержки такого доступа.

Выше были приведены общие соображения, из которых следует потребность разработки средств доступа к данным в терминах концептуальной схемы. Перед тем, как перейти к описанию этих средств, рассмотрим возможные конкретные их применения.

Во-первых, использование языков моделирования становится особенно привлекательным для определении глобальной концептуальной схемы при интеграции данных из различных источников. Эта привлекательность обусловлена не столько богатством языков моделирования, сколько возможностью определить схему в простых и наглядных терминах (которые в случае UML имеют и наглядную графическую нотацию). Определение схемы в простых и наглядных терминах может оказаться очень полезным в случае, когда число интегрируемых источников велико, и когда источники имеют сложные схемы. При этом полученная глобальная

концептуальная схема, вероятно, будет содержать большое количество сущностей и связей между ними, поэтому переход к более низкоуровневому логическому представлению схемы может существенно усложнить формулирование запросов к интегрируемым данным. Приведенные рассуждения говорят о потребности в средствах доступа в терминах концептуальной схемы в контексте задачи интеграции данных. Подробнее о применении предложенных в этой главе средств к решению задачи интеграции данных смотри в [51].

Во-вторых, в последние годы развивается направление исследований и промышленных разработок, которые связаны с генерацией пользовательских интерфейсов по описаниям моделей на языке UML [40]. Если предположить наличие средств генерации по UML-модели интерактивных пользовательских интерфейсов для доступа к данным, то такие средства можно было бы расширить поддержкой автоматической генерации запросов по результатам взаимодействия пользователя с интерфейсом. Например, по диаграмме классов UML можно сгенерировать набор форм, в которые пользователь будет вводить критерии запросов к данным, соответствующим UML-модели. По результатам заполнения этих форм система может автоматически сгенерировать запрос в терминах используемой UML-модели.

2.2 Язык UQL

В этом разделе описывается язык UML Query Language (UQL), разработанный для формулирования запросов к UML-данным. Язык UQL построен на основе языка объектных ограничений OCL (Object Constraint Language)[41], предназначенного для ограничения пространства допустимых объектов, соответствующих структуре UML-модели, что обеспечивает возможнсть более полного выражение в модели семантики моделируемой предметной области. Накопленный опыт разработки языков запросов и языков ограничений говорит о том, что обычно для языков этих двух классов используются аналогичные выразительные средства. Это наблюдение привело к идее разработки языка запросов UQL на базе языка ограничений OCL. Тем не менее, недостаточно изменить только общую семантику OCL-выражения, чтобы получить язык запросов, удовлетворяющий всем необходимым требованиям. Комментарии по поводу того, почему это так, и в чем заключаются

необходимые отличия языков UQL и OCL, будут приводиться по ходу описания языка. Язык UQL, как и OCL, был разработан для использования пользователями или разработчиками прикладных приложений, которые не имеют серьезной математической подготовки. UQL является строго и статически типизированным языком, что позволяет поддерживать средства статической проверки запросов, которые желательны для контроля запросов, разработанных специалистами такого класса. Отметим, что язык UQL позволяет формулировать только запросы и не поддерживает средств выражения модификации данных.

2.2.1 Модель данных языка UQL

Модель данных, на которой основан языка UQL, базируется на спецификации языка UML [41][2]. В этой спецификации определены структурные элементы двух уровней: уровня схемы и уровня данных. К уровню схемы относятся такие понятия, как класс, типизированный атрибут класса и связь между классами. Именно в этих терминах определяется UML-модель. К уровню данных относятся следующие понятия: объект (под объектом понимается экземпляр некоторого класса), значение атрибута, экземпляр связи [3]. Уровень данных будем также называть *моделью UML-данных*. В этой модели данных схема (UML-модель) представляет собой граф, в вершинах которого находятся классы, и ребра которого являются связями между классами. Данные, соответствующие UML-модели, представляют собой совокупность атрибутированных объектов, где каждый объект является экземпляром определенного в UML-модели класса, и связей между ними, где каждая такая связь является экземпляром некоторой связи, определенной в модели. Для разработки языка UQL требовалось несколько расширить уровень данных (то есть модель UML-данных). В связи с этим перечислим только основные ее положения (включая и внесенные расширения):

- Объектом - называется экземпляр некоторого класса, определенного в UML-модели. Каждый объект имеет уникальный идентификатор, который отличает его от других объектов, вне зависимости от возможного равенства значений соответствующих атрибутов. Будем

[2]Здесь термин модель данных используется не в строгом смысле для обозначения только структурной составляющей.

[3]Будем называть экземпляр связи просто связью в тех случаях, когда это не вызывает двусмысленности.

предполагать, что уникальный идентификатор имеет строковое представление, которое сохраняет свойство уникальности.

- Предполагается, что определение каждого класса состоит из определения атрибутов класса и определения связей. В определение атрибута входит его имя и тип значения. В определении связи указывается имя класса, с которым установлена связь; множественность связи с объектами этого класса ("ноль или один", "строго один", "ноль или более", "один и более"); требование уникальности связанных объектов или его отсутствие; упорядоченность связанных объектов. Определение класса рассматривается как определение типа. Объект некоторого типа, определяемого классом, является значением типов, определяемых его суперклассами.

- Предполагается, что каждый класс имеет экстент, который представляет собой множество всех объектов (экземпляров) этого класса. Экстент класса включает объекты подклассов.

Отметим, что язык UQL не поддерживает средств работы с методами, которые могут быть определены в классе. Такие определения не принимаются во внимание.

2.2.2 Система типов языка UQL

Каждый UQL-запрос имеет свою собственную систему типов, состав которой зависит от UML-модели, к которой адресован этот запрос. Типы, составляющие такую систему, делятся на две группы:

- *Типы, определенные в UML-модели* (зависящие от адресуемой модели). Каждый определенный в модели класс является типом[4]. Имя типа совпадает с именем класса.

- *Предопределенные в языке UQL типы* (не зависящие от адресуемой модели и входящие в систему типов любого запроса): набор скалярных и агрегатных типов.

К предопределенным агрегатным типам относятся следующие:

[4]Таким образом, под классом понимается тип, определяемый набором атрибутов класса, а не множество объектов этого класса. Для обозначения последнего выше было введено понятие "экстент".

- Set(X) — множество, содержащее элементы некоторого типа X без дубликатов. Если X — это некоторый класс, то под дубликатами понимаются объекты, имеющие один и тот же идентификатор. Если X — это скалярный тип, то дубликаты определяются при помощи определенной для этого типа операции сравнения по значению.

- Bag(X) — мультимножество, содержащее элементы некоторого типа X, среди которых могут быть дубликаты. Порядок элементов мультимножества не определен.

- Sequence(X) — упорядоченное мультимножество (далее называемое последовательность) элементов некоторого типа X.

Агрегатные типы являются параметрическими. При определении типов переменная X была использована для обозначения параметра типа. В качестве значения параметра агрегатного типа могут выступать только скалярные типы и типы, определенные в UML-модели. Таким образом, используя наличие параметров в агрегатных типах, нельзя строить вложенные агрегатные типы.

По определению все агрегатные типы являются гомогенными, поскольку значение любого агрегатного типа включает в себя объекты одного и того же типа, который является фактическим параметром этого агрегатного типа. Значения агрегатного типа могут состоять из объектов классов, имеющих общий суперкласс, если этот суперкласс является фактическим параметром агрегатного типа, поскольку такие объекты относятся к нескольким классам, в число которых входит этот общий суперкласс. Отсутствие в системе типов гетерогенных агрегатных типов не кажется ограничением по следующим двум причинам. Во-первых, только несколько операций, таких как `count` (количество элементов, составляющих значение агрегатного типа), `isEmpty` (является ли количество элементов, составляющих значение агрегатного типа, равным 0), может быть определено над значениям гетерогенных агрегатных типов. Во-вторых, как кажется, гетерогенные агрегатные типы были бы полезны только для обработки иррегулярностей в UML-модели. Например, некоторые адреса людей могли бы быть представлены в виде строк, в то время как другие являлись бы объектами некоторого класса, и могло бы потребоваться сосчитать адреса. Но UML-модель имеет четко выраженную тенденцию быть регулярной.

К предопределенным скалярным типам относятся следующие: boolean, integer, real, double, string. Не будем подробно обсуждать скалярные типы, считая, что можно определить другой набор скалярных типов без влияния на другие аспекты языка UQL, кроме автоматического приведения типа (о нем см. ниже).

Помимо параметрического полиморфизма агрегатных типов, в UQL поддерживается другой тип полиморфизма - *автоматическое приведение типа (coercion).* Автоматическое приведение типа используется для автоматического преобразования действительного параметра некоторой операции к типу соответствующего формального параметра по предопределенным правилам. Предопределенные правила автоматического приведения типа представлены в таблице 2.1.

Тип	Приведенный тип
Integer	Real
Real	Double

Таблица 2.1: Правила приведения типа

Следующие подразделы содержат определение операций над значениями описанных типов. При этом определяются как синтаксис, так и семантика каждой операции.

2.2.3 Операции над значениями агрегатных типов

Операции над значениями агрегатных типов имеют ряд общих синтаксических особенностей. Продемонстрируем их на шаблонных выражениях.

```
<aggr-value-expr>=>op(iter | <uql-expr>,...,<uql-expr>,...)
```

Выражение `<aggr-value-expr>`, определяющее значение агрегатного типа, к которому применяется операция `op`, записывается перед операцией и отделяется от нее последовательностью символов `=>`. Такой вид записи заимствован из языка OCL и используется для записи композиции как последовательности шагов вычисления, где каждый шаг разделяется символами `=>`. Значение выражения `<aggr-value-expr>` является одним из действительных параметров операции, хотя и не указывается среди других параметров, перечисленных в скобках после имени операции. Значение выражения `<aggr-value-expr>` будем называть *входным*

параметром. Параметром некоторых операций является выражение, которое будет применяться к каждому элементу агрегата. Для таких параметров определяются переменные-итераторы (`iter`), отделяемые от выражения символом | и используемые в таком выражении-параметре для ссылки на текущий элемент агрегата при итерировании. В разделе 2.2.6 будет показан способ сокращенной записи таких выражений-параметров без явного определения итераторов.

Определим операции над значениями типа Set(X). У таких операций входной параметр является множеством значений типа X.

`count():Integer` возвращает число элементов во входном множестве.

`isEmpty():Boolean` возвращает “ложь”, если входное множество содержит хотя бы один элемент.

`select(expr:Boolean): Set(X)` возвращает подмножество входного множества, для элементов которого результатом вычисления `expr` является “истина”.

`collect(expr:Y):Bag(Y)` возвращает мультимножество, полученное в результате объединения результатов вычисления `expr` для каждого элемента входного множества. Тип `Y` результата вычисления `expr` определяет тип элементов результирующего мультимножества.

`exist(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления `expr` является "истина" хотя бы для одного элемента входного множества.

`forAll(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления `expr` является "истина" для всех элементов входного множества.

Кроме того, над значениями типов Set(Integer), Set(Real) и Set(Double) определены агрегатные операции `sum()`, `avg()`, `max()`, `min()`, которые возвращают сумму, среднее арифметическое, максимум и минимум элементов входного множества соответственно. Тип результата определяется типом-параметром агрегатного типа входного множества.

Определим операции над значениями типа Bag(X). У таких операций входной параметр является мультимножеством значений типа X.

`count():Integer` возвращает число элементов во входном мультимножестве.

`isEmpty():Boolean` возвращает “ложь”, если входное мультимножество содержит хотя бы один элемент.

`select(expr:Boolean): Bag(X)` возвращает подмультимножество входного мультимножества, для элементов которого результатом вычисления `expr` является "истина".

`collect(expr:Y):Bag(Y)` возвращает мультимножество, полученное в результате объединения результатов вычисления `expr` для каждого элемента входного мультимножества. Тип `Y` результата выражения `expr` определяет тип элементов результирующего мультимножества.

`exist(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления `expr` является "истина" хотя бы для одного элемента входного мультимножества.

`forAll(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления `expr` является "истина" для всех элементов входного мультимножества.

Кроме того, над значениями типов Bag(Integer), Bag(Real) и Bag(Double) определены агрегатные операции `sum()`, `avg()`, `max()`, `min()`, которые возвращают сумму, среднее арифметическое, максимум и минимум элементов входного мультимножества соответственно. Тип результата определяется типом-параметром агрегатного типа входного мультимножества.

Определим операции над значениями типа Sequence(X). У таких операций входной параметр является последовательностью значений типа X.

`count():Integer` возвращает число элементов во входной последовательности.

`isEmpty():Boolean` возвращает “ложь”, если входная последовательность содержит хотя бы один элемент.

`select(expr:Boolean): Bag(X)` возвращает мультимножество, состоящее из элементов входной последовательности, для которых результатом вычисления `expr` является "истина".

Замечание 1 *Операция* `select`*, по сути, производит фильтрацию входной последовательности. При определении операции select типом результата было выбрано мультимножество, а не последовательность, поскольку предполагается, что порядок элементов после проведения фильтрации не может быть известен разработчику запроса, так как он не знает, в каких позициях находились исключенные в результате фильтрации элементы. Поэтому нет смысла сохранять*

порядок элементов после фильтрации. В некоторых случаях отсутствие требования сохранения порядка дает возможность оптимизатору выполнить запрос более эффективными методами.

`collect(expr:Y):Bag(Y)` возвращает мультимножество, полученное в результате объединения результатов вычисления expr для каждого элемента входной последовательности. Тип `Y` результата выражения `expr` определяет тип элементов результирующего мультимножества.

Замечание 2 *При определении операции* `collect` *типом результата было выбрано мультимножество а не последовательность, поскольку в противном случае пришлось бы потребовать, чтобы типом результата выражения expr была последовательность.*

`exist(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления `expr` является "истина" хотя бы для одного элемента входной последовательности.

`forAll(expr:Boolean):Boolean` возвращает "истину", если результатом вычисления expr является "истина" для всех элементов входной последовательности.

`at(i:Integer):X` возвращает i-ый элемент входной последовательности. Нумерация элементов начинается с 1.

`last(i:Integer):X` возвращает последний элемент входной последовательности.

Кроме того, над значениями типов Sequence(Integer), Sequence(Real) и Sequence(Double) определены агрегатные операции `sum()`, `avg()`, `max()`, `min()`, которые возвращают сумму, среднее арифметическое, максимум и минимум элементов входной последовательности соответственно. Тип результата определяется типом-параметром агрегатного типа входной последовательности.

2.2.4 Операции над объектами классов

В языке UQL поддерживаются следующие операции над объектами любого класса:

- операция берет в качестве параметра объект и имя атрибута и возвращает значение этого атрибута;

- операция берет объект и имя связи и возвращает все объекты, доступные по указанной связи от указанного объекта. Тип результата определяется следующими правилами:
  - результат будет объектом, если множественность связи "один";
  - результат будет множеством объектов связанного класса, если связь неупорядочена и есть требование уникальности;
  - результат будет мультимножеством объектов связанного класса, если связь неупорядочена и нет требования уникальности;
  - результат будет последовательностью объектов связанного класса, если связь упорядочена.
- операция `type():String` берет объект и возвращает имя типа (непосредственного класса) этого объекта;
- операция `oid():String` берет объект и возвращает его уникальный идентификатор, представленный в качестве значения типа String[5]

В синтаксисе языка UQL эти операции выражаются с использованием так называемой "точечной нотации", где в качестве "точки" используются символы "." и "!". Например:

- `<obj-expr>.age` возвращает значение атрибута age, если результат выражения `<obj-expr>` имеет тип (класс), в котором определен атрибут age. В противном случае возникает ошибка типа.
- `<obj-expr>!employee` возвращает все объекты, доступные по связи с именем `employee`, если результат выражения `<obj-expr>` имеет тип (класс), в котором определена связь employee. В противном случае возникает ошибка типа.
- `<obj-expr>.type()` возвращает значение имя класса объекта, полученного в результате вычисления выражения `<obj-expr>`, если `<obj-expr>` возвращает объект. В противном случае возникает ошибка типа.

Обратим внимание на то, что в каждом из трех перечисленных видов выражений при помощи только синтаксического анализа можно определить, какую операцию над объектом имел в виду составитель

[5]Такое представление возможно и сохраняет свойство уникальности. Об этом смотри 2.2.1.

запроса. Действительно, отличительной чертой первого вида выражения является наличие символа ".", отличительной чертой второго вида является знак "!", отличительной чертой третьего вида выражения является наличие пары скобок. Возможность только средствами синтаксического анализа определять намерения пользователя позволяют выполнять UQL-запросы, не проводя при этом вывода типов, за счет чего появляется возможность создания легковесных реализаций языка. В этом состоит одно из отличий языка UQL от языка OCL. В последнем вместо символа "!" используется символ ".", и, как следствие, статический вывод типов необходим для того, чтобы понять, имел в виду пользователь получение значения атрибута или переход по связи.

2.2.5 Общая семантика UQL-запроса

Любой запрос на языке UQL представляет собой выражение, являющееся композицией операций над значениями типов, на которых базируется язык UQL. Таким образом, общая семантика запроса на языке UQL следующая: результат запроса есть результат вычисления определяющего его выражения. Отметим, что общая семантика OCL-выражения отлична от только что сформулированной. В языке OCL выражение, состоящее из композиции операций языка, всегда пишется в контексте некоторого класса, который явно указывается. При этом выражение трактуется как ограничение на объекты класса. Подобный подход можно было бы использовать и в языке UQL. Контекстный класс определял бы класс, объекты которого мы хотим получить в качестве результатов запроса, а выражение определяло бы фильтрующий предикат над объектами этого класса. Однако этот подход является неудобным. Проблема состоит не в том, что в результате запроса мы всегда получаем объекты только одного класса (с точностью до наиболее общего класса в иерархии наследования). Попытка снять это ограничение привела бы к невозможности поддержки статического вывода типа. Проблема в том, что класс, объект которого мы хотим получить, может быть неизвестен пользователю непосредственно. Приведем подтверждающий пример. Как уже отмечалось, язык UQL предполагается использовать, в том числе, и при создании графических поисковых интерфейсов. Одним из широко распространенных видов интерфейсов являются каталоги. При любой организации каталогов пользователь будет просматривать объекты

класса и переходить по связи от конкретного объекта к связанным с ним объектам. Если взять за основу подход, заимствованный из языка OCL, то при генерации запроса, осуществляющего переход по связи, необходимо будет выяснить по модели на какой класс указывает связь, по которой необходимо осуществить переход. Затем потребуется сформулировать запрос в следующем виде: получить объекты класса (контекстного класса), на которые, как было выяснено, указывает связь, и которые связаны с заданным объектом. В семантике языка UQL этот запрос можно выразить так: перейти от заданного объекта по связи. Последняя формулировка представляется более предпочтительной, что в контексте языков запросов говорит в пользу подхода, положенного в основу языка UQL и против подхода языка OCL. Обратим также внимание на необходимость указания "заданного объекта" при использовании обоих подходов. Единственно возможный способ такого указания в задании уникального идентификатора требуемого объекта (точнее в поиске объекта по заданному уникальному идентификатору). Для этого необходимо поддерживать доступность уникального идентификатора объекта на уровне языка, то есть поддерживать возможность получения уникального идентификатора в качестве значения некоторого типа. Как было показано в разделе 2.2.4, UQL поддерживает такую возможность.

2.2.6 Сокращенные формы записи и правила разрешения неоднозначности

Определение переменных-итераторов в параметрах некоторых операций, определенных над значениями агрегатных типов, может быть опущено в случае, когда это не приводит к неоднозначности. В тех местах запроса, где переменные-итераторы опущены, автоматически вводятся неявные переменные-итераторы. Опишем механизм ввода неявных переменных-итераторов и правила разрешения неоднозначности (то есть неявной подстановки неявно введенных переменных) на покрывающем все случаи примере. Рассмотрим выражение:

```
extent("Company")=>select(c| c.foundationYear=1995)=>
forAll(employee=>exist(lastName=name)))
```

Переход по связи `employee` и получение значений атрибутов `lastName` и `name` осуществляется без явного определения переменных-итераторов,

поэтому вводятся две неявные переменные. Первая неявная переменная типа Company определяется для параметра операции `forAll` (назовем эту переменную `iter1`), и вторая неявная переменная типа `Person` — для параметра операции `exist` (назовем эту переменную `iter2`). Разрешение неоднозначности происходит следующим образом:

- Из подвыражения перехода по связи `employee` доступна только неявная переменная `iter1`, поэтому будет произведена неявная подстановка `iter1.employee`.
- В аргументе операции `exist`, в котором содержатся подвыражения получения значений атрибутов `lastName` и `name`, доступны две неявные переменные `iter1` и `iter2`. Поскольку в классе `Company` определен атрибут с именем `name` и не определены ни атрибут, ни связь с именем `lastName`, а в классе `Person` определен атрибут с именем `lastName` и не определены ни атрибут, ни связь с именем `name`, будет произведена неявная подстановка `iter2.lastName=iter1.name`.
- Если бы оба атрибута `name` и `lastName` были определены в одном из классов (например, в классе `Person`), то в этом случае возникла бы неоднозначность, приводящая к ошибке времени компиляции. Эту неоднозначность можно преодолеть только явным определением переменных-итераторов.

Другой сокращенной формой записи является группировка при помощи скобок общих подвыражений заканчивающихся "." или "=>". Покажем эту возможность на примерах. Запросы

```
p.age>20 and p.employer.age<60
```

и

```
p.employee=>count()<3 and not p.employee=>isEmpty()
```

могут быть записаны следующим образом соответственно

```
p.(age>20 and employer.age<60)
```

и

```
p.employee=>(count()<3 and not isEmpty())
```

Понятно, что такая сокращенная форма может быть всегда однозначно преобразована в несокращенную на этапе компиляции запроса.

2.2.7 О статической типизируемости языка UQL

Язык UQL является статически типизируемым языком. Под этим понимается то, что тип любого UQL-выражения может быть определен с точностью до наиболее общего типа в иерархии наследования на этапе компиляции запроса (то есть в статике). Статическая типизация языка UQL подтверждается тем, что для всех операций языка UQL верно следующее утверждение: по типу входных параметров однозначно определяется тип результата. Фактически этого удалось добиться благодаря использованию только гомогенных агрегатных типов. То, что тип определяется с точностью до наиболее общего типа, объясняется потребностью поддерживать операцию перехода по связи от объекта к объектам различных классов, которые, тем не менее, должны быть объединены общей иерархией наследования.

Свойство статической типизации языка UQL позволяет обеспечить средства статической проверки запросов. Такие средства особенно важны для языков, предназначенных для специалистов без серьезной математической подготовки.

2.3 Реализация языка UQL через отображение в XQuery

Для практического использования предложенных в предыдущих разделах концепций и модельно-языковых средств необходимо добиться их эффективной реализации. Прямолинейным подходом к решению этой задачи является разработка и реализация набора всех необходимых для этого методов. Однако большинство требуемых методов не имеют сегодня непосредственных аналогов, и, следовательно, их пришлось бы разрабатывать практически с нуля, что потребовало бы больших инвестиций в разработку. Другой, более экономичный подход основан на отображении разработанных модельно-языковых средств на некоторую модель данных, для которой уже существуют реализации. Будем называть эту модель базовой.

Отображение на базовую модель должно включать в себя три группы правил. К первой группе относятся правила представления UML-данных в терминах структурной составляющей базовой модели. Ко второй группе относятся правила трансляции UQL-запросов в выражения на языке

запросов базовой модели. К третьей группе относятся правила обратного отображения данных из терминов базовой модели в термины UML-данных[6].

Реализация подхода, основанного на отображении, сводится к построению надстройки над существующей системой. Будем называть эту систему базовой. В задачу надстройки входит следующее: транслировать полученные на входе запросы в запросы, понятные базовой системе; передавать их на выполнение базовой системе; и осуществлять необходимые преобразования результатов работы базовой системы. При условии несложности трансляции и преобразований такой подход к реализации может оказаться более выгодным по сравнению с прямолинейным, поскольку позволяет сэкономить на разработке многих методов, которые уже поддерживаются базовой системой. Таким образом, приведенные рассуждения говорят в пользу второго подхода.

В качестве базовой модели предлагается выбрать модель данных XML (точнее язык XQuery [31] и структурную модель, на которой он основан [35]). Такой выбор обусловлен следующими соображениями. Во-первых, как будет показано в следующих подразделах, можно построить необходимое отображение модели UML-данных на модель данных XML, и такое отображение получается несложным. Это позволяет разработать простые правила трансляции UQL-запросов в соответствующие XQuery-запросы. Во-вторых, результаты выполнения запросов имеет смысл возвращать в формате XML, поскольку этот формат данных является широко распространенным, и, следовательно, нет необходимости преобразовывать результаты выполнения запросов базовой системой.

В следующих двух подразделах описывается отображение модели UML-данных на модель XML и правила трансляции UQL-запросов в XQuery запросы, основанные на этом отображении.

2.3.1 Отображение модели UML-данных на модель данных XML

Отображение модели UML-данных на XML модель обеспечивает представление в формате XML данных, соответствующих произвольной UML-модели. Это отображение суть отображение атрибутированного

[6]Правила третьей группы определяют отображение, обратное отображению, задаваемому правилами первой группы.

графа[7] на атрибутированное дерево. Понятно, что такое отображение неоднозначно. С целью дать представление о вариантах возможных отображений отметим, что узлы графа переходят в узлы дерева, а ребрам графа могут быть поставлены в соответствие ребра дерева, и, кроме того, ребра графа могут быть представлены ссылкой по значению. Наличие различных вариантов отображения требует определения критерия для выбора одного из них. Таким критерием будем считать простоту отображения UQL-запросов в XQuery-запросы, которая во многом определяется выбором способа отображения UML-данных в XML. Простота UQL-XQuery отображения является основой быстрой реализации трансляции запросов. При этом не будем учитывать сложность получаемых XQuery-запросов, предполагая наличие развитых средств их оптимизации. В этом разделе определяются правила отображения UML-данных на модель данных XML. Это отображение представляется наилучшим с точки зрения правил трансляции запросов. К вопросу трансляции запросов мы вернемся в следующем разделе.

Перейдем к описанию выбранного способа UML-XML отображения. Этот способ таков, что схема получаемых XML-данных зависит от UML-модели. Это было сделано для того, чтобы перенести как можно больше семантики из UML-модели в схему XML-данных. Поэтому, определяя способ UML-XML отображения, необходимо определить правила построения схемы XML-данных по UML-модели. В соответствии с построенной схемой будут храниться значения UML-модели. Пусть имеется UML-модель с именем `model_name`, содержащая N классов со следующими именами `class_name1`, `class_name2`, ..., `class_nameN`. Класс с именем `class_nameK`, где K — некоторое число из диапазона от 1 до N включительно, содержит Km атрибутов с именами `attr_nameK1`, ..., `attr_nameKm` и Kl связей с именами `ref_nameK1`, ..., `ref_nameKl`. Будем считать, что все данные, соответствующие UML-модели, будут храниться в одном XML-документе. Ниже приводится описание схемы этого XML-документа на языке DTD [36].

```
<!ELEMENT data (class_name1|class_name2|...|class-nameN)*>
...
<!ELEMENT class_nameK (attr_nameK1, ... , attr_nameKm, refs)>
```

[7]Представление данных в модели UML значений можно рассматривать как граф, у которого вершины — объекты UML классов, ребра — связи между объектами, являющиеся экземплярами связей между UML-классами

```
<!ATTLIST class_nameK
          id ID #REQUIRED>
<!ELEMENT attr_nameK1 (#PCDATA)>
...
<!ELEMENT attr_nameKm (#PCDATA)>
<!ELEMENT refs (ref_nameK1, ... , ref_nameKl)>
<!ELEMENT ref_nameK1 (idref)*>
...
<!ELEMENT ref_nameKl (idref)*>
<!ELEMENT idref (#PCDATA)>
```

Выбранный подход к UML-XML отображению можно описать следующим образом. Объекты классов представляются XML-элементами, имя которых есть имя класса объекта (XML-элементы с именами `class_name1`, `class_name2`, ..., `class-nameN`). Атрибуты объекта представляются в виде подэлементов элемента, соответствующего объекту, при этом имя подэлемента совпадает с именем атрибута (XML-элементы с именами `attr_nameK1`, ..., `attr_nameKm`). Уникальный идентификатор объекта представляется в виде значения XML-атрибута с именем `id` типа `ID`. Связи представляются в виде XML-элементов, имя каждого из которых есть имя связи, и содержание которых составляют значения XML-атрибутов с именем `id` тех элементов, которые представляют связанные объекты (XML-элементы с именами `ref_nameK1`, ..., `ref_nameKl`). Таким образом, все связи моделируются при помощи ссылки по значению атрибута `id`. Чтобы отличать XML-элементы, представляющие атрибуты объекта, от XML-элементов, представляющих связь, последние группируются в содержание XML-элемента с именем `refs`, который находится на том же уровне, что и элементы представляющие атрибуты объекта.

2.3.2 Правила трансляции UQL-запросов в XQuery-запросы

В этом разделе определяются правила трансляции UQL-запросов в XQuery-запросы в соответствии с правилами отображения UML-XML, определенными в разделе 2.3.1. Общая семантика UQL-запроса такова, что результатом запроса является результат вычисления выражения, определяющего запрос, поэтому определение языка UQL сводилось к определению операций, композиция которых является

выражением, определяющим запрос. Аналогично, для определения правил трансляции UQL-запроса достаточно определить правила трансляции каждой операции языка UQL. Тогда процесс трансляции запроса будет представлять собой рекурсивное применение таких правил.

Ниже приводится перечень правил трансляции каждой операции языка UQL в выражение на языке XQuery. При этом будем считать, что все данные UML-модели хранятся в XML-документе с именем "data" в соответствии с правилами, определенными в разделе 2.3.1.

Применение правил отображения к UQL-выражению `expr` будем обозначать `[[expr]]`. В фигурных скобках будут приводиться дополнительные комментарии. Выражение `T(expr)` означает тип результата UQL-выражения `expr`. Под выражением `$new-varN`, где `N` - некоторое натуральное число, будем понимать имя переменной, не совпадающее с именами ни одной переменной в получаемом в результате трансляции XQuery-запросе.

```
[[objExpr.attName]] {attName - имя атрибута objExpr} =>
[[objExpr]]/attName

[[objExpr!refName]] {refName - имя связи objExpr} =>
FOR $v1 IN document("data")/data/*
WHERE
    SOME $idref1 IN [[objExpr]]/refs/refName/idref
    SATISFIES $v1/@id=$idref1/node()
RETURN $v

[[varName]] => $varName

[[aggrTypeExpr=>count()]] => count([[aggrTypeExpr]])

[[aggrTypeExpr=>isEmpty()]] => empty([[aggrTypeExpr]])

[[aggrTypeExpr=>select(v | boolExpr)]]
{T(aggrTypeExpr)=Set(X)|Bag(X)} =>
unordered FOR [[v]] IN [[aggrTypeExpr]]
WHERE [[boolExpr]]
RETURN [[v]]
```

```
[[aggrTypeExpr=>select(v | boolExpr)]]
{T(aggrTypeExpr)=Sequence(X)} =>
FOR [[v]] IN [[aggrTypeExpr]]
WHERE [[boolExpr]]
RETURN [[v]]

[[aggrTypeExpr=>collect(v | expr)]] =>
FOR [[v]] IN [[aggrTypeExpr]]
RETURN [[expr]]

[[aggrTypeExpr=>forAll(v | boolExpr)]] =>
EVERY [[v]] IN [[aggrTypeExpr]]
SATISFIES [[boolExpr]]

[[aggrTypeExpr=>exist(v | boolExpr)]] =>
SOME [[v]] IN [[aggrTypeExpr]]
SATISFIES [[boolExpr]]

[[aggrTypeExpr=>sum()]] => sum([[aggrTypeExpr]])

[[aggrTypeExpr=>avg()]] => avg([[aggrTypeExpr]])

[[aggrTypeExpr=>max()]] => max([[aggrTypeExpr]])

[[aggrTypeExpr=>min()]] => min([[aggrTypeExpr]])

[[aggrTypeExpr=>at(indexExpr)]] =>
[[aggrTypeExpr]][ [[indexExpr]] ]

[[aggrTypeExpr=>last(indexExpr)]] =>
[[aggrTypeExpr]][position()=last()]
```

Рекурсивное применение приведенных правил позволяет отобразить любой UQL-запрос в XQuery-запрос. Требуемую простоту трансляции можно считать достигнутой. Однако обратим внимание на то, что для непосредственной реализации этих правил требуется поддержка механизма статического вывода типов (см. выше правило трансляции select-выражений). Однако это не является обязательным для

полной поддержки языка и было сделано лишь с целью получения потенциально более эффективного XQuery-запроса. Отсутствие требования упорядоченности результата за счет использования оператора `unordered` в случае применения операции `select` к множеству или мультимножеству дает больше возможности для оптимизации получаемого XQuery-запроса, особенно в случае наличия вложенного подзапроса в WHERE-операторе. Таким образом, требование поддержки механизма статического вывода типа может быть снято с сохранением корректности реализации языка UQL за счет безусловной трансляции select-выражения в XQuery-выражение без применения оператора `unordered`.

2.4 Выводы

Путем переопределения общей семантики языка объектных ограничений OCL и привнесения в этот язык статической типизируемости автору удалось разработать язык запросов UQL, который позволят формулировать запросы к данным в терминах диаграмм классов UML и может быть использован конечным пользователем. Предложенный метод реализации разработанного языка запросов, основанный на отображении диаграмм классов UML на модель данных XML и трансляции в соответствии с этим отображением запросов на этом языке в XQuery-запросы, позволяет строить недорогие и эффективные системы, поддерживающие этот язык.

Глава 3

Логическая оптимизация запросов на языке XQuery

Настоящая глава посвящена описанию предлагаемого автором общего подхода к логической оптимизации выполнения запросов на языке XQuery и разработанных автором методов в рамках этого подхода.

3.1 Общий подход к логической оптимизации XQuery-запросов

3.1.1 Постановка задачи логической оптимизации XQuery-запросов и обоснование оправданности такой постановки

Под оптимизацией запросов обычно понимается совокупность действий, направленных на улучшение некоторых характеристик выполнения запросов. В данной работе мы следуем традиционному подходу, в котором основной характеристикой выполнения запроса является время его выполнения. Улучшение этой характеристики состоит в сокращении времени. Теоретически итеративный процесс оптимизации запроса можно представить следующим образом. Оптимизация производится над логическим представлением запроса. Логическое представление запроса основано на операциях, которые определены исключительно в терминах модели данных. Базируясь на таком логическом представлении, производя различные преобразования и постепенно уточняя уровень представления, оптимизатор строит все возможные физические планы выполнения запроса, которые выражаются в терминах однозначно интерпретируемых операций реализационного уровня. На следующем этапе производится количественная оценка стоимости выполнения каждого физического плана. Эта оценка основывается на использовании некоторого

набора стоимостных функций, вычисляемых над статистическими характеристиками базы данных. Оптимальным считается физический план с наименьшим значением оценки стоимости.

Такой идеализированный подход неприменим на практике, поскольку построение всех возможных физических планов требует чрезмерно большого числа преобразований, что приводит к недопустимо длительному процессу оптимизации. В действительности из множества всех возможных преобразований выделяется подмножество тех преобразований, которые сокращают число возможных вариантов генерации физического плана запроса, но гарантируют высокую вероятность того, что среди оставшихся вариантов содержится оптимальный. Тогда в ходе оптимизации выделенные преобразования применяются к исходному логическому представлению, и в результате получается некоторое новое промежуточное представление запроса, которое расценивается как более близкое к оптимальному, чем исходное. К промежуточному представлению применяются оставшиеся преобразования, которые формируют оцениваемые физические планы запросов. Заметим, что в число выделенных преобразований входят те преобразования, которые обычно расцениваются как эвристические методы оптимизации. Кроме того, преобразования из выделенного подмножества не выводят за рамки логического представления. Поэтому эти преобразования можно определить в терминах логического представления, что упрощает построение и применение преобразований. С этим связано распространенное название подхода: применение выделенных видов преобразований называют логической оптимизацией. Оставшиеся преобразования обеспечивают переход от логического представления к физическому, и поэтому их применение называется физической оптимизацией.

Базируясь на описанных общих принципах построения оптимизаторов, сформулируем теперь более точно задачу логической оптимизации в контексте оптимизации запросов на языке XQuery.

Определение 1 Логическим представлением *XQuery-запроса называется формулировка XQuery-запроса в виде выражения, представляющего собой композицию операций, которые определены в терминах некоторой модели данных XML.*

Замечание: Упомянутое в определении выражение является

композицией операций, и естественна его графовая интерпретация в виде дерева. В дальнейшем мы часто будем неявно опираться на эту древовидную природу логического представления. Для определенности будем считать, что дерево ориентировано сверху (от корня) вниз.

Определение 2 *Пусть Q - это логическое представление запроса, а S - схема, к которой адресован запрос. Тогда* задачей логической оптимизации *будем называть задачу нахождения отображения R, которое осуществляется на основе выполнения преобразований и ставит в соответствие любой паре (Q, S) эквивалентное логическое представление $Q_{opt} = R(Q, S)$, обладающее улучшенными временными характеристиками (для большинства запросов).*

Замечание 3 *Далее в тексте под терминами* запрос *и* выражение *будем понимать некоторое логическое представление какого-либо запроса или его фрагмента, кроме тех случаев, которые оговариваются особо.*

Определение 3 *Логические представления называются* эквивалентными, *если они приводят к физическим планам запроса, при выполнении которого вырабатывается один и тот же результат.*

Определение 4 *Отображение R в постановке задачи логической оптимизации будем называть* логическим оптимизатором *или просто оптимизатором.*

Для завершения постановки задачи логической оптимизации необходимо определить, какие виды преобразований относятся к преобразованиям, улучшающим временные характеристики обработки большинства запросов. Как уже отмечалось, подбор таких преобразований осуществляется эвристически, и можно лишь перечислить эти преобразования и некоторым образом обосновать, что их можно считать улучшающими для большинства запросов. Такие виды преобразований будем называть *видами логической оптимизации.*

Виды логической оптимизации:

1. *Опустить предикат по дереву логического выражения ниже конструкторов.* Это способствует сокращению объема результатов промежуточных вычислений, к которым будут применяться конструкторы XML-элементов. Такое сокращение объема существенно уменьшает время выполнения запроса, потому

что конструктор является очень дорогостоящей операцией, при вычислении которой производится глубокое копирование узлов с генерацией заново их уникальных идентификаторов.

2. *Опустить предикат ниже вложенных итераторов.* Это способствует сокращению объема результатов промежуточных вычислений, которые будут участвовать в вычислении итераторов. Такое сокращение объема позволяет уменьшить число итераций для каждого итератора, за счет чего существенно уменьшается время вычисления совокупности вложенных итераторов.

3. *Вычислить подвыражения в статике, когда это возможно.* Этот вид оптимизации, во-первых, сокращает представление запроса, что упрощает его дальнейшую оптимизацию, и, во-вторых, устраняет потребность в повторных вычислениях во время выполнения, если такое подвыражение находится в теле итератора.

4. *Повысить уровень декларативности запроса.* Это позволяет расширить пространство возможных физических планов планами, более близкими к оптимальным.

5. *Избавиться от конструкторов, вычисление которых не требуется для получения результата запроса.* Это позволяет избегать лишних вычислений дорогостоящих конструкторов XML-элементов.

6. *Произвести открытую вставку тел определяемых пользователем функций XQuery.* Такая оптимизация позволяет уменьшить время вычисления запроса за счет снижения накладных расходов на повторные вызовы функции, если эти вызовы находятся в теле итератора, и повысить эффективность последующей оптимизации, поскольку при открытой вставке происходит, по сути, привязка подзапроса, оформленного в виде функции XQuery, к контексту вызова функции.

3.1.2 Использование техники перезаписи в качестве средства описания решения задачи логической оптимизации и как основы для реализации

Согласно постановке задачи логической оптимизации по одному логическому представлению запроса необходимо построить другое

представление, обладающее некоторыми "улучшенными" свойствами. Такой переход от представления к представлению естественно рассматривать как процесс перезаписи выражения, определяющего логическое представление. Естественность подобной трактовки приводит к мысли о выборе правил перезаписи как основного средства описания решения задачи логической оптимизации. Как говорилось в главе 1 подход к решению задачи логической оптимизации для различных языков запросов с использованием правил перезаписи применяется наиболее часто и хорошо себя зарекомендовал, поэтому в настоящей работе этот подход был выбран за основу. Приведем основные понятия теории перезаписи из [29], на основе которых ниже будет переформулирована задача логической оптимизации, что позволит свести ее к построению правил перезаписи и доказательству свойств этих правил.

Определение 5 *Системой перезаписи называется структура* $R = (QLR, \{\rightarrow_\alpha \ | \alpha \in I\}$, *состоящая из множества всех возможных логических представлений* QLR *и бинарных отношений* $\rightarrow_\alpha$ *на* QLR, *индексированных на множестве* I.

Определение 6 *Если* $R = (QLR, \{\rightarrow_\alpha | \alpha \in I\}$ *есть некоторая система перезаписи, то* $\rightarrow_\alpha$ *называются правилами перезаписи из* R.

Определение 7 $q \in QLR$ *- называется нормальной формой в системе перезаписи* R, *если к* q *нельзя применить ни одного правила перезаписи из* R.

Определение 8 *Система перезаписи обладает свойством нормальной формы (normal form property - NF), если применение правил перезаписи к любому логическому представлению* $q \in QLR$ *приводит* q *к нормальной форме.*

Тогда в терминах теории перезаписи задача логической оптимизации формулируется следующим образом:

Определение 9 *Задачей логической оптимизации в терминах перезаписи будем называть задачу построения системы перезаписи, которая обладает свойством нормальной формы, и доказательства того, что в этой нормальной форме реализованы все виды логической оптимизации.*

Таким образом, решение задачи логической оптимизации сведено к построению системы перезаписи, правила которой реализуют все виды логической оптимизации, и доказательству ее свойств.

3.1.3 Классы правил перезаписи и этапы логической оптимизации

Имеет смысл сгруппировать правила перезаписи, описывающие решение задачи логической оптимизации, в классы в соответствии с двумя принципами: во-первых, в один класс должны входить правила, взаимосвязанные по некоторому признаку, и не должно существовать взаимосвязей по этому признаку между правилами из разных классов; во-вторых, классы правил должны быть сформированы таким образом, чтобы по результатам этой классификации можно было определить этапы логической оптимизации. Группировка правил в классы и последовательное применение классов правил позволяет рассматривать правила, применяемые на каждом этапе, как отдельную систему перезаписи, которая реализует один или несколько видов логической оптимизации, и доказывать свойства каждой системы в отдельности.

Ниже приводится перечень классов правил перезаписи, построенных в соответствии с изложенными принципами. Для каждого класса указываются признак общности, по которому правила были сгруппированы в этот класс, и виды логической оптимизации, которые предполагается реализовывать через правила этого класса. На базе вводимых классов определяются этапы оптимизации.

1. *Преобразование структуры запроса.* В этот класс входят правила, которые определяются только в терминах оптимизируемого логического представления. Как будет показано в разделе 3.6, при помощи этих правил реализуются следующие виды логической оптимизации: опустить предикат ниже конструкторов; избавиться от конструкторов, вычисление которых не требуется для получения результата запроса. Преобразование структуры запроса описывается в разделе 3.6.

2. *Повышение декларативности представления запроса.* При помощи правил этого класса, главным образом, осуществляется трансляция вложенных итераторов в операции соединения. При этом реализуются следующие виды оптимизации: повысить декларативность запроса;

опустить предикат ниже вложенных итераторов. Этот класс правил описывается в разделе 3.7.

3. *Семантическая перезапись.* К этому классу относятся правила, при помощи которых запрос оптимизируется на основании информации из схемы данных, к которым адресован этот запрос. Термин "семантическая оптимизация" используется по следующим соображениям. Традиционно под семантической оптимизацией понимается вид перезаписи, основанный на использовании схемы данных, к которым адресован запрос, или ограничений целостности, наложенных на запрашиваемые данные. На сегодняшний день в контексте управления XML-данными понятие ограничений целостности не разработано в должной степени. Поэтому пока имеет смысл говорить о семантической оптимизации только как о виде перезаписи с использованием информации из схемы данных. Основная цель семантической оптимизации может быть определена следующим образом: попытаться максимально редуцировать запрос за счет вычисления всего запроса или его подзапросов только на схеме данных (т.е. без обращения к данным). Посредством достижения этой цели реализуется следующий вид логической оптимизации: вычислить подвыражения, которые могут быть вычислены в статике. Семантическая оптимизация рассматривается в разделе 3.4.

4. *Открытая вставка тел XQuery-функций.* К этому классу правил отнесены правила, которые принимают во внимание вызовы XQuery-функций, определенных пользователем, и работают с телами этих функций. Дело в том, что, как показано в разделе 3.5, наличие в запросе вызовов определенных пользователем XQuery-функций уменьшает возможность оптимизации запроса: к такому запросу применимо меньшее число правил трех описанных выше классов по сравнению с эквивалентным запросом, в котором вызовы функций заменены на их тела. Замену вызова функции на ее тело с корректной подстановкой фактических параметров принято называть *открытой вставкой тела функции.* С помощью обсуждаемого вида перезаписи осуществляется открытая вставка тел определяемых пользователем нерекурсивных XQuery-функций и некоторого вида рекурсивных функций (важного на практике подвида структурной рекурсии, когда при каждом рекурсивном вызове происходит спуск

на один или несколько уровней вниз по дереву XML-данных). Таким образом, применение правил данного вида позволяет в некоторых случаях повысить возможность применения правил другого вида, т.е. повысить уровень оптимизируемости запросов за счет избавления от вызова определенных пользователем XQuery-функций. При помощи этого класса правил реализуется одноименный вид логической оптимизации. Открытая вставка тел XQuery-функций описывается в разделе 3.5.

Перечисление видов оптимизации, поддерживаемых каждым классом, показывает, что введенных классов правил оказывается достаточно для реализации всех видов логической оптимизации. Используя свойства классов правил перезаписи, можно определить на основе этих классов этапы логической оптимизации, задающие последовательность применения классов правил перезаписи:

1. Первый этап: открытая вставка тел XQuery-функций и семантическая перезапись. Применение правил перезаписи этих двух классов на одном этапе позволяет сократить объем оперативной памяти, требуемый для реализации открытой вставки тел XQuery-функций со структурной рекурсией, поскольку для тел таких функций характерны подвыражения, вычисляемые на схеме (то есть упрощаемые при помощи семантической перезаписи). Основная идея алгоритма совместного использования этих классов правил перезаписи состоит в том, что после каждого применения правила открытой вставки тела XQuery-функции к вставленному телу применяются правила семантической перезаписи с учетом соответствующего контекста. Эти два правила применяются на первом этапе, поскольку они помогают существенно повысить эффективность применения правил двух других классов.

2. Второй этап: преобразование структуры запроса. Правила этого класса повышают эффективность применения правил класса "повысить уровень декларативности формулировки запроса", поэтому их следует применять на более раннем этапе.

3. Третий этап: повышение уровня декларативности представления запроса.

В следующих разделах описывается решение задачи логической оптимизации в рамках предложенного общего подхода. В разделе 3.2, определяется логическое представление XQuery-запросов, определенное на основании спецификации базового подмножества языка, которое называется XQuery Core и определено в [33]. Не все правила, необходимые для реализации требуемых видов логической оптимизации, могут быть определены в терминах этого логического представления. Поэтому в разделе 3.3 определяется расширение стандартной модели данных XML [35], на которой базируется XQuery Core. В следующих разделах (3.4, 3.5, 3.6, 3.7) описываются четыре системы перезаписи, построенные на базе описанных выше классов правил перезаписи, и показывается, какие виды логической оптимизации реализуются при помощи каждой из систем перезаписи. В заключительном разделе 3.8 этой главы представлены результаты использования логического оптимизатора.

3.2 Логическое представление XQuery-запросов

В этом разделе определяются операции логического представления (далее сокращенно LR-операции) XQuery-запросов. Определение LR-операций осуществляется посредством отображения представляющих их синтаксических конструкций на выражения, опирающиеся на операции, для которых определена формальная семантика: XQuery Core [33] и операции, определенные в [34]. Таким образом, определяются как синтаксис, так и семантика LR-операций. Большинству, но не всем LR-операциям взаимно однозначно соответствуют операции XQuery Core или операции определенные в [34]. Различие состоит в основном в выборе префиксного синтаксиса для унифицированной записи операций в виде *имя-операции(аргумент1,...,аргументаN)*, а также в снятии семантической перегруженности некоторых операций XQuery Core, или наоборот во введении операций, которые отображаются в композицию нескольких операций XQuery Core и [34]. Это было сделано для создания набора операций, который более удобен для построения систем перезаписи.

Ниже следуют определения LR-операций, организованные в именованные группы. Имена групп будут использоваться в последующих разделах для ссылки на все операции группы. Применение правил отображения на XQuery Core к выражению логического представления expr будем обозначать $[\text{expr}]_{\text{x-core}}$.

Базовые операции логического представления

Мета операции

$$[\text{return}(e_1, \lambda(x|e_2))]_{\text{x-core}} = \text{for } \$x \text{ in } [e_1]_{\text{x-core}} \text{ return } [e_2]_{\text{x-core}} \quad (3.1)$$

$$[\text{seq}(e_1, e_2, ..., e_n)]_{\text{x-core}} = \text{op : concatenate}(e_1, \text{op : concatenate}(e_2, \text{op : concatenate}(\ldots, e_n))) \quad (3.2)$$

$$[\text{if}(e_1, e_2, e_3)]_{\text{x-core}} = \text{let fs : new} := [e_1]_{\text{x-core}} \text{ return if (fs : new) then } [e_2]_{\text{x-core}} \text{ else } [e_3]_{\text{x-core}} \quad (3.3)$$

$$[\text{ts}(e, \lambda(x| \text{cases}(\text{case}(ST_1, e_1), ..., \text{case}(ST_n, e_n), \text{def}(e_{n+1})]_{\text{x-core}} = \text{for } \$y \text{ in } [e]_{\text{x-core}} \text{ return typeswitch}(\$y) \text{ case } ST_1 \ \$x \text{ return } [e_1]_{\text{x-core}} \ldots \text{case } ST_n \ \$x \text{ return } [e_n]_{\text{x-core}} \text{ default } \$x \text{ return } [e_{n+1}]_{\text{x-core}} \quad (3.4)$$

Комментарии: Введение операции ts в таком виде обусловлено часто появляющейся потребностью применять typeswitch к x-последовательности. Применение же ts к объекту эквивалентно применению typeswitch к этому объекту.

Конструкторы

$$
\begin{aligned}
&[\text{element}(e_{name}, e)]_{\text{x-core}} = \\
&\qquad \text{let } \$name := [e_{name}]_{\text{x-core}} \text{ return} \\
&\qquad \text{let } \$e := [e]_{\text{x-core}} \text{ return} \\
&\qquad \text{let } \$attributes := \text{for } \$\text{fs} : \text{new in } \$e \text{ return} \\
&\qquad \text{typeswitch } \$\text{fs} : \text{new} \\
&\qquad \text{case attribute } \$a \text{ return } \$a \\
&\qquad \text{default return } () \\
&\qquad \text{let } \$\text{everythingelse} := \text{for } \$\text{fs} : \text{new in } \$e \text{ return} \\
&\qquad \text{typeswitch } \$\text{fs} : \text{new} \\
&\qquad \text{case attribute } \$a \text{ return } () \\
&\qquad \text{default } \$v \text{ return } \$v \\
&\qquad \text{return} \\
&\qquad \text{element } \{\$name\} \ \{\$\text{attributes}, \$\text{everythingelse}\}
\end{aligned}
\tag{3.5}
$$

$$
\begin{aligned}
&[\text{attribute}(e_{name}, e)]_{\text{x-core}} = \\
&\qquad \text{let } \$name := [e_{name}]_{\text{x-core}} \text{ return} \\
&\qquad \text{attribute } \{\$name\} \ \{[e]_{\text{x-core}}\}
\end{aligned}
\tag{3.6}
$$

Аксессоры

$$
[\text{name}(e)]_{\text{x-core}} = \text{fn} : \text{name}([e]_{\text{x-core}}) \tag{3.7}
$$

$$
[\text{node} - \text{kind}(e)]_{\text{x-core}} = \text{fn} : \text{node} - \text{kind}([e]_{\text{x-core}}) \tag{3.8}
$$

Как мы увидим далее, определенные выше аксессоры обладают некоторыми общими свойствами, поэтому для удобства ссылки на эти операции, как на группу, введем для них обобщенную операцию accessor.

$$[\text{child}(e, test)]_{\text{x-core}} = \text{for } \$x \text{ in } [e]_{x-core} \text{ return } \$x/\text{child} ::[test]_{x-core} \tag{3.9}$$

$$[\text{descendant}(e, test)]_{\text{x-core}} = \text{for } \$x \text{ in } [e]_{\text{x-core}} \text{ return } \$x/\text{descendant} - \text{or} - \text{self} ::[test]_{\text{x-core}} \tag{3.10}$$

$$[test]_{\text{x-core}} = \text{elem(name)} \mid \text{elem}(*) \mid \text{attr(name)} \mid \text{attr}(*) \mid \text{node()} \mid \text{text()} \tag{3.11}$$

Кванторы

$$[\text{some}(e_1, \lambda(x|e_2))]_{\text{x-core}} = \text{xf} : \text{not}(\text{xf} : \text{empty}(\text{for } \$x \text{ in } [e_1]_{\text{x-core}} \text{ return if } [e_2]_{\text{x-core}} \text{ then } \$x \text{ else } ())) \tag{3.12}$$

$$[\text{every}(e_1, \lambda(x|e_2))]_{\text{x-core}} = \text{xf} : \text{empty}(\text{for } \$x \text{ in } [e_1]_{\text{x-core}} \text{ return if } (\text{xf} : \text{not}([e_2]_{\text{x-core}})) \text{ then } \$x \text{ else } ()) \tag{3.13}$$

Операции над x-последовательностями

$$[\text{empty}(e)]_{\text{x-core}} = \text{xf} : \text{empty}([e]_{\text{x-core}}) \tag{3.14}$$

Агрегатные функции

$$[\text{count}(e)]_{\text{x-core}} = \text{fn} : \text{count}([e]_{\text{x-core}}) \tag{3.15}$$

$$[\text{avg}(e)]_{\text{x-core}} = \text{fn} : \text{avg}([e]_{\text{x-core}}) \tag{3.16}$$

$$[\text{max}(e)]_{\text{x-core}} = \text{fn} : \text{max}([e]_{\text{x-core}}) \tag{3.17}$$

$$[\text{min}(e)]_{\text{x-core}} = \text{fn} : \text{min}([e]_{\text{x-core}}) \tag{3.18}$$

$$[\text{sum}(e)]_{\text{x-core}} = \text{fn} : \text{sum}([e]_{\text{x-core}}) \tag{3.19}$$

Как мы увидим далее, агрегатные функции обладают некоторыми общими свойствами, поэтому для удобства ссылки на эти операции, как на группу, введем для них обобщенную операцию aggregate.

Операции над значениями атомарных типов

В разделах 5-12 спецификации [34] определяются функции и операторы над значениями атомарных типов, определенных в [38]. Языковые средства семейства XML, в том числе и язык XQuery, разрабатывались таким образом, чтобы система атомарных типов, на которых базируются языки, была ортогональной к этим языкам. В методах логической оптимизации, обсуждаемых в настоящей работе, не различаются операции над значениями атомарных типов. Поэтому введем обобщенную операцию fo (functions and operators), под которой будем понимать любую операцию из разделов 5-12 спецификации [34]. Впоследствии разработанные методы могут быть конкретизированы для оптимизации выражений с операциями над значениями атомарных типов благодаря свойству ортогональности системы атомарных типов языку XQuery. При этом такая конкретизация может быть сделана как для стандартной системы типов языка XQuery, так и для любой другой системы, которая будет обладать свойством ортогональности к XQuery.

Генераторы x-последовательностей

$$[\text{doc}(e)]_{\text{x-core}} = \text{one} - \text{doc}([e]_{\text{x-core}}) \tag{3.20}$$

Вспомогательные операции

Операция special-string-value-type(e) возвращает xs:anySimpleType(string-value(e)) (т.е результат вычисления string-value(e), приведенный к типу anySimpleType).

Операция special-name-type(e) возвращает e, приведенное к типу special-name, если тип e приводим к QName; в противном случае возвращается ошибка.

Операция special-node-kind-type(e) возвращает e, приведенное к типу special-node-kind, если тип e приводим к string; в противном случае возвращается ошибка.

Операция extended-name(e) возвращает e, приведенное к типу QName, если тип e - "special-name"; в противном случае возвращается результат вычисления name(e).

Операция extended-node-kind(e) возвращает e, приведенное к типу string, если тип e - "special-node-kind"; в противном случае возвращается результат вычисления node-kind(e).

И-операции

К группе И-операций отнесем операции над узлами, при определении которых используются операции над уникальными идентификаторами.

$$[\text{union}(e_1, e_2, \ldots, e_n)]_{\text{x-core}} = \\ \text{op} : \text{union}([e_1]_{\text{x-core}}, \text{op} : \text{union}(\ldots, [e_n]_{\text{x-core}})) \quad (3.21)$$

$$[\text{intersect}(e_1, e_2, \ldots, e_n)]_{\text{x-core}} = \\ \text{op} : \text{intersect}([e_1]_{\text{x-core}}, \text{op} : \text{intersect}(\ldots, [e_n]_{\text{x-core}})) \quad (3.22)$$

$$[\text{except}(e_1, e_2, \ldots, e_n)]_{\text{x-core}} = \\ \text{op} : \text{except}([e_1]_{\text{x-core}}, \text{op} : \text{except}(\ldots, [e_n]_{\text{x-core}})) \quad (3.23)$$

$$[\text{ddo}(e)]_{\text{x-core}} = \text{fs} : \text{distinct} - \text{doc} - \text{order}([e]_{\text{x-core}}) \quad (3.24)$$

Операции доступа к динамическому контексту

В спецификации языка XQuery [31] определено понятие *динамического контекста* выражения (*dynamic context, evaluation context*), который состоит из набора компонентов. В [34] определяются функции доступа к компонентам динамического контекста. Почти все компоненты динамического контекста являются глобальными для всего запроса, кроме компонентов, которые называются *фокусом* (*focus*) выражения. При использовании техники перезаписи необходимо особо учитывать фокус выражения, чтобы в результате перезаписи получить эквивалентное выражение. В то же время использование остальных компонент динамического контекста в выражении не может быть причиной перехода к неэквивалентному выражению по причине их глобальности для всего запроса. В этом разделе мы ограничимся перечислением функций, определенных в спецификации [34] и предназначенных для обращения к компонентам фокуса выражения, которые мы собираемся поддерживать при оптимизации запроса: fn:context-item(), fn:position(), fn:last(). В разделе 3.6.3 описывается метод поддержки этих функций при перезаписи. Эти операции определяются на структурных элементах расширенной модели данных, которые обсуждаются в следующем разделе.

3.3 Расширение модели данных XML и логического представления XQuery-запросов

В предыдущем разделе были определены такие операции логического представления запросов в терминах модели данных XML [35], для которых существуют аналоги в XQuery Core. Чтобы реализовать некоторые виды логической оптимизации, требуется ввести операции, для которых нет аналогов в XQuery Core и которые неудобно или неэффективно определять в терминах модели данных XML. К этой группе относятся те операции, для представления результатов которых необходимо объединить объекты в одну структуру. Сразу отметим, что x-последовательность не может быть использована в качестве такой объединяющей структуры, поскольку x-последовательности не могут быть вложенными, а выравниваются. Например, при попытке определить операцию соединения двух x-последовательностей необходимо каким-то образом представлять последовательность пар, которые состоят из элементов соединяемых

x-последовательностей. Последовательность может быть представлена в виде x-последовательности. Но какую структуру выбрать для представления пары? В работе [39], для этого использовались XML-элементы. Такой подход к решению проблемы кажется неприемлемым, поскольку при применении конструктора XML-элемента для построения пары произойдет глубокое копирование узлов, составляющих пару, с генерацией новых уникальных идентификаторов. Это не только неэффективно, но может также привести к неэквивалентному запросу, если обработка результатов соединения опирается на использование уникальных идентификаторов. В настоящей работе модель данных XML расширяется новой структурой, которая будет использоваться для решения рассмотренной проблемы, и определяются необходимые операции в терминах этой новой структуры. Назовем эту новую структуру *унией*[1]. Уния позволяет объединять объекты и строить последовательности таких объединений.

Определение 10 *Унией называется упорядоченный набор объектов.*

Для обозначения унии будем использовать квадратные скобки. Например, уния, состоящая из двух объектов a и b, обозначается $[a, b]$.

Определение 11 *Арностью унии называется число объектов в ней.*

Подобно тому, как в модели данных XML объект отождествляется с x-последовательностью, состоящей из одного элемента, которым является этот объект, отождествим объект с унией, которая состоит из одного элемента, которым является этот объект. Тогда мы можем считать, что все функции и операторы XQuery замкнуты на x-последовательности уний, которые состоят из одного объекта. Ниже приводится определения дополнительных операций, которые опираются на понятие x-последовательности уний, состоящих более чем из одного объекта. Логическое представление, определенное в предыдущем разделе, расширенное операциями из этого раздела, будем называть *расширенным логическим представлением, или сокращенно ELR.*

В определениях используется функция $f(x, y)$, которая возвращает порядковый номер объекта y в x-последовательности x[2]. Порядковые номера начинаются с 1.

[1] УНИЯ от лат. unio “соединяю”

[2] При определении функции предполагается, что y содержится в x

Определение 12 *X-соединением (xjoin) двух x-последовательностей уний A и B по условию $P(x, y)$ называется x-последовательность всевозможных уний вида $[a, b]$, которые удовлетворяют следующим условиям (определяющим порядок уний в результирующей x-последовательности)*[3]*:*

1. $a \in A, b \in B$
2. $P(a, b)$ *— истинно*
3. $\forall a_1,\ a_2 \in A:\ f(A, a_1) < f(A, a_2)$ *и* $\forall b_1,\ b_2 \in B:\ c_1 = [a_1, b_1], c_2 = [a_2, b_2] \in C \Rightarrow f(C, c_1) < f(C, c_2)$
4. $\forall a \in A,\ b_1 \in B,\ b_2 \in B:\ c_1 = [a, b_1], c_2 = [a, b_2] \in C$ *и* $f(B, b_1) < f(B, b_2) \Rightarrow f(C, c_1) < f(C, c_2)$

Определение 13 *Полу-X-соединением (semixjoin) двух x-последовательностей уний A и B по условию $P(x, y)$ называется x-последовательность уний вида $[a]$, которые удовлетворяют следующим условиям* [4]*:*

1. $a \in A$
2. $\forall a \in A,\ \exists b \in B : P(a, b)$ *— истинно*
3. $\forall a_1,\ a_2 \in A:\ f(A, a_1) < f(A, a_2) \Rightarrow f(C, a_1) < f(C, a_2)$

Определение 14 *Левым внешним X-соединением (louterxjoin) двух x-последовательностей уний A и B по условию $P(x, y)$ называется x-последовательность уний, которая получается из множества $\{[a, b] | a \in A, b \in B, P(a, b)$ — истинно$\} + \{[a, ()] | \forall b \in B, P(a, b)$ — ложно$\}$ при выполнении следующих условий (определяющих порядок уний в результирующей x-последовательности)*[5]*:*

1. $\forall a_1,\ a_2 \in A:\ f(A, a_1) < f(A, a_2)$ *и* $\forall b_1,\ b_2 \in B$ *или* $b_1,\ b_2 = ():\ c_1 = [a_1, b_1], c_2 = [a_2, b_2] \in C \Rightarrow f(C, c_1) < f(C, c_2)$
2. $\forall a \in A,\ b_1 \in B,\ b_2 \in B:\ c_1 = [a, b_1], c_2 = [a, b_2] \in C$ *и* $f(B, b_1) < f(B, b_2) \Rightarrow f(C, c_1) < f(C, c_2)$

[3] Определение можно обобщить до случая произвольного числа x-последовательностей уний.
[4] Определение можно обобщить до случая произвольного числа x-последовательностей уний.
[5] Определение можно обобщить до случая произвольного числа x-последовательностей уний.

Определение 15 *Анти-X-соединением (antixjoin) двух x-последовательностей уний A и B с условием $P(x, y)$ называется x-последовательность уний вида $[a]$, которые удовлетворяют следующим условиям* [6]*:*

1. $a \in A$
2. $\forall a \in A \quad$ *не* $\exists b \in B : P(a, b)$ *— истинно*
3. $\forall a_1,\ a_2 \in A :\ f(A, a_1) < f(A, a_2) \Rightarrow f(C, a_1) < f(C, a_2)$

Определение 16 *X-проекцией (xproject) x-последовательности A уний (одной арности n) по порядковым номерам $k_1, \ldots, k_n$ называется x-последовательность уний, которые получаются из уний исходной x-последовательности A путем исключения из них объектов, порядковые номера которых в этих униях не принадлежат множеству $k_1, \ldots, k_n$. Причем из результирующей x-последовательности уний удаляются дубликаты таким образом, что в подпоследвательности уний-дубликатов остается первая уния.*

Доступ к элементам унии будем поддерживать при помощи расширенной операции return через связывание с переменными объектов, составляющих унию. Расширим операцию return(e, f) следующим образом. Будем считать, что первым аргументом e операции return может быть x-последовательность из уний, состоящих из более чем одного объекта, но все унии состоят из одного и того же числа объектов. Второй аргумент f - функция, имеющая число формальных параметров равное числу объектов в униях x-последовательности. В соответствии с семантикой операции return, определенной в предыдущем разделе, результатом этой операции является конкатенация x-последовательностей, полученных в результате применения функции f к каждому элементу x-последовательности e. В расширенном варианте операции return будем также применять f к каждому элементу e (т.е. к каждой унии из e), беря элементы унии в качестве фактических параметров для функции f с учетом порядка элементов в унии и порядка формальных параметров функции f.

[6] Определение можно обобщить до случая произвольного числа x-последовательностей уний.

3.4 Семантическая оптимизация

В этом разделе описывается класс правил семантической оптимизации. Будем называть систему перезаписи, основанную на этих правилах, "системой перезаписи семантической оптимизации", или сокращенно СПСО. Общим для всех правил этого класса является то, что в их определении используется информация из схемы данных, к которым адресуется запрос. Использование информации из схемы данных осуществляется через применение механизма вывода схемы, определенного в спецификации [33]. Этот механизм позволяет по выражению и по схеме данных, к которым адресуется это выражение, вывести схему данных, получаемых в результате вычисления выражения. Результат применения механизма вывода схемы к выражению e будем обозначать $T(e)$. Для записи схемы будем использовать язык описания схем, определенный в [33]. Чтобы указать, что схема s описана на этом языке, будем использовать обозначение $[s]_{\text{XQuery\_type}}$. Неформально опишем основные конструкции языка описания схем, которые требуются для понимания приводимого далее перечня правил:

- $[\text{element}]_{\text{XQuery\_type}}$ - узел XML-элемент;
- $[\text{element n}]_{\text{XQuery\_type}}$ - узел XML-элемент с именем n;
- $[\text{element n of type t}]_{\text{XQuery\_type}}$ - узел XML-элемент с именем n и содержанием t;
- $[\text{attribute}]_{\text{XQuery\_type}}$ - узел XML-атрибут;
- $[\text{document}]_{\text{XQuery\_type}}$ - узел XML-документ;
- $[\text{text}]_{\text{XQuery\_type}}$ - текстовый узел;
- $[()]_{\text{XQuery\_type}}$ - пустая x-последовательность.

Замечание 4 *При определении правил перезаписи будем считать, что оптимизируется запрос, который прошел статическую проверку типов (то есть не будут рассматриваться случаи когда, например, операция nate применяется к x-последовательности, содержащей более одного элемента), хотя при реализации имеет смысл совмещать статическую проверку типа с семантической оптимизацией.*

Правила формулируются в контексте перезаписи, ссылка на который из правил осуществляется при помощи ключевого слова context. Выражение gen-var(context) означает переменную, имя которой отлично от имен переменных, содержащихся в контексте перезаписи. Выражение $e_1\{x \mapsto e_2\}_{\text{context}}$ означает выражение e_1, в котором заменены все свободные вхождения переменной x на выражение e_2. Причем предполагается, что такая замена производится корректно с учетом возможных перекрытий имен переменных из контекста context.

Ниже перечисляются правила перезаписи СПСО.

$$\text{return}(e_1, \lambda((x)e_2))\{T(e_1) = [()]_{\text{XQuery\_type}}\} \Rightarrow e_2\{x \mapsto e_2\}_{\text{context}} \quad (3.25)$$

$$\text{name}(e) \quad \Rightarrow \quad \begin{cases} \text{n}, T(e) = [\text{element n of type t}]_{\text{XQuery\_type}} \\ \text{n}, T(e) = [\text{attribute n of type t}]_{\text{XQuery\_type}} \\ \text{""}, T(e) = [\text{document} \mid \text{text} \mid ()]_{\text{XQuery\_type}} \end{cases} \quad (3.26)$$

$$\text{node} - \text{kind}(e) \quad \Rightarrow \quad \begin{cases} \text{"document"}, T(e) = [\text{document}]_{\text{XQuery\_type}} \\ \text{"element"}, T(e) = [\text{element}]_{\text{XQuery\_type}} \\ \text{"attribute"}, T(e) = [\text{attribute}]_{\text{XQuery\_type}} \\ \text{"text"}, T(e) = [\text{text}]_{\text{XQuery\_type}} \end{cases} \quad (3.27)$$

Отметим, что поскольку все эти правила можно реализовать за один обход дерева запроса, то будем считать доказанным для введенной системы перезаписи свойство нормальной формы.

Основным результатом применения перечисленных правил является сокращение запроса за счет вычисления подзапросов. При вычислении подзапросов используется информация из схемы данных с применением механизма вывода схемы результата подзапроса. Еще раз подчеркнем, что такое сокращение производится без обращения к данным. Это позволяет реализовать следующий вид логической оптимизации: вычислить подвыражения в статике, когда это возможно. Забегая вперед, отметим, что семантическая оптимизация обычно помогает существенно редуцировать запрос и тем самым упростить и ускорить оптимизацию запроса на последующих этапах, если перед этим к запросу применялась процедура открытой вставки тел XQuery-функций (см. следующий раздел).

3.5 Открытая вставка тел XQuery функций

Важным свойством языка XQuery является поддержка функций, определяемых пользователем на языке XQuery (далее в этом разделе будем называть их XQuery-функциями). Важность этого свойства заключается не столько в удобстве повторного использования фрагментов запросов за счет их определения в виде функции, сколько в возможности определения полиморфных рекурсивных функций, которые являются естественным средством формулирования запросов при наличии древовидной природы XML-данных.

Рассмотрим основные проблемы, связанные с оптимизацией запросов, которые содержат вызовы XQuery-функций. Наличие вызовов XQuery-функции (основной интерес представляют полиморфные функции) в оптимизируемом запросе препятствует оптимизации такого запроса, поскольку поведение такой функции может в значительной степени зависеть от контекста, из которого она была вызвана[7]. Зависимость поведения функции от контекста ее вызова приводит к низкой эффективности семантической оптимизации тела функции и к невозможности определения фактического типа результата функции при рассмотрении функции вне контекста вызова, что ведет к снижению эффективности семантической оптимизации подзапроса/запроса, содержащего вызов функции. Кроме того, наличие в запросе нескольких вызовов одной функции препятствует применению правил, направленных на опускание предиката в тело функции, поскольку изменение тела функции для одного из вызовов изменяет семантику других вызовов.

Одним из способов повысить уровень оптимизируемости запросов, содержащих вызовы XQuery-функций, является замена вызовов XQuery-функции в запросе на ее тело. За счет этого осуществляется встраивание тела XQuery-функции в контекст, и тем самым устраняются описанные выше сложности при оптимизации запроса. Замену вызова функции на ее тело называют *открытой вставкой тела функции* (*function inlining*). При разработке алгоритма открытой вставки тела функции необходимо учитывать, что в случае рекурсивной функции простая подстановка тела функции не приводит к избавлению от вызова этой функции. В этом случае в алгоритме подстановки тел функций нужно или избавиться

[7]Под контекстом вызова функции понимается подзапрос/запрос, содержащий вызов, и выражения, определяющие фактические параметры функции.

некоторым способом от рекурсивного вызова функций, если это возможно, или заменить вызовы функции столько раз, сколько рекурсивных вызовов будет произведено при выполнении запроса (в этом случае, даже если в запросе останутся вызовы функции, они не будут производиться в момент выполнения запроса). Ниже описывается разработанный автором алгоритм открытой вставки тел XQuery-функций, который решает эту задачу для практически важного класса функций. Функции из этого класса осуществляют структурную рекурсию, в которой на каждом шаге происходит спуск по XML-дереву на один или более уровней. Основная идея алгоритма состоит в оценке глубины XML-дерева, по которому предполагается спуск при рекурсивном вызове, на основании информации о схеме данных. Такая оценка позволяет осуществлять открытую вставку тела функции с погрешностью числа подстановок, не большей максимальной глубины XML-дерева, поскольку в подобных функциях выход из рекурсии произойдет не позднее достижения листа XML-дерева.

Описание алгоритма начнем с определения входных данных и обозначений. Входными данными являются определения XQuery-функций (для описания алгоритма достаточно определения одной функции $f(x_1, ..., x_n) = e(x_1, ..., x_n)$, где $e(x_1, ..., x_n)$ - некоторое выражение от параметров $x_1, ..., x_n$), и некоторый запрос, в котором могут содержаться вызовы этих функций.

Введем следующие обозначения:

- $\mu(type - expr)$ - глубина схемы, которая равна максимальному числу уровней в XML-деревьях, соответствующих данной схеме.
- $mask$ - маска, которая определяется для каждой функции и представляет собой вектор из нулей и единиц длинны n, где единица стоит в позиции, соответствующей аргументу функции, по которому происходило уменьшение глубины схемы на всех предыдущих вызовах этой функции.
- $T(e)$ - схема результата выражения e, определенная по схеме данных, к которым адресуется запрос (о выводе схемы смотри раздел 3.4).
- $\overline{x} = (x_1, \ldots, x_n)$ - вектор формальных параметров функции текущего вызова.
- $\overline{e} = (e_1, ..., e_n)$ - вектор фактических параметров текущего вызова.

- $v[i]$ - i-ый элемент вектора v.
- $T(\overline{e}) = (T(e_1), ..., T(e_n))$ - вектор, полученный применением функции $T(type - expr)$ к каждому элементу вектора e.
- $m(\overline{e}) = (\mu(T(e_1)), ..., \mu(T(e_n)))$ - вектор, полученный применением функции $\mu(type - expr)$ к каждому элементу вектора $T(\overline{e})$.
- $\{\{f(e_1, ..., e_n)\}\}$ - результат открытой вставки тела функции f вместо вызова $f(e_1, ..., e_n)$.

Алгоритм открытой вставки анализирует текущий вызов функции и принимает решение об очередной открытой вставке или остановке алгоритма. При анализе текущего вызова предполагается наличие информации о предыдущем вызове этой функции: $mask_{prev}$, $m(\overline{e}_{prev})$. Текущий вызов определяется следующим образом. В начале работы алгоритма обходится сверху-вниз дерево, соответствующее логическому представлению запроса. При этом каждый встреченный вызов XQuery-функции становится для алгоритма текущим. Если при обработке текущего вызова была произведена очередная открытая вставка, обходится сверху-вниз дерево, соответствующее выражению, на которое был заменен текущий вызов. При этом каждый встреченный вызов XQuery-функции становится для алгоритма текущим. Ниже перечисляются шаги, производимые при обработке текущего вызова:

1. Вычисляем $m_{cur} = m(\overline{e}_{cur})$.
2. Вычисляем $\Delta m = m(\overline{e}_{prev}) - m(\overline{e}_{cur})$.
3. Вычисляем новую маску $mask_{cur} = sgn(\Delta m * mask_{prev})$.
4. Если $\exists i : mask_{cur}[i] = 1$ (т.е. новая маска показывает, что при текущем вызове не было спуска ни по одному фактическому параметру), то алгоритм останавливается и оставляет вызов функции нетронутым, в противном случае текущий вызов заменяется на $\{\{f(e_1, ..., e_n)\}\}$.

Очевидно, что приведенный алгоритм остановится при любой последовательности вызовов любой функции, поскольку открытая вставка тела некоторой функции продолжается до тех пор, пока на каждом следующем рекурсивном вызове этой функции происходит уменьшение глубины одного или нескольких фактических параметров (т.е. происходит

спуск по значению одного или нескольких фактических параметров), но этот процесс уменьшения глубины обязательно остановится или по логике вычисления выражения, определяющего тело функции, или по достижении глубины 0[8].

3.6 Преобразование структуры запроса

В этой главе описывается система перезаписи, в соответствии с правилами которой преобразуется структура запроса, за счет чего реализуются два вида логической оптимизации. Эта система перезаписи включает наибольшее число правил, и эти правила относительно несложны. Общим для всех правил системы является то, что они определяются только в терминах логического представления запроса (в частности, без использования информации о схеме адресуемых данных). Набор правил таков (и это несложно показать на примере), что переход к нормальной форме невозможен за один обход выражения с применением правил, поскольку применение некоторых правил приводит к ситуации, когда появляется возможность для применения других правил, что требует повторного обхода выражения. Поэтому при разработке алгоритма применения правил этой системы следует учесть необходимость повторного обхода представления после применения правил. Кроме того, отмеченное свойство правил требует доказательства свойства нормальной формы системы перезаписи, чтобы показать конечность процесса применения правил.

Правила этой системы перезаписи определяются в терминах подвыражений, поэтому процедура применения правил состоит в следующем: находится подвыражение, к которому можно применить правило; затем правило применяется к найденному подвыражению; результат применения правила заменяет исходное подвыражение. Для обеспечения перехода к эквивалентному запросу в целом в результате применения некоторых правил к подвыражению запроса используется контекст перезаписи, который представляет собой множество всех переменных, доступных в этом подвыражении и определенных за пределами этого подвыражения. Использование такого контекста помогает избежать некорректного переопределения переменных при применении

[8]глубина не может быть по определению отрицательной и всегда будет достигнута благодаря конечности глубины схемы адресуемых запросом данных

правил.

Построение системы перезаписи проведем в три итерации. На первой итерации рассмотрим только подмножество LR-операций, в которое входят операции группы "Базовые операции логического представления". Для выражений, являющихся композицией этой группы операций, строится система перезаписи. Затем доказывается, что построенная система перезаписи обладает свойством нормальной формы и реализует в полной мере два вида логической оптимизации. На второй и третьей итерациях расширим рассматриваемое подмножество LR-операций за счет включения в него И-операций и операций поддержки динамического контекста соответственно. Система перезаписи, построенная на первой итерации, дополняется правилами сначала для И-операций, а затем для операций поддержки динамического контекста. Будет показано, что невозможно добиться реализации в полной мере двух видов логической оптимизации, которые реализовывались исходной системой, для расширенных систем перезаписи по причинам природы И-операций и наличия динамического контекста. Тем не менее, для достаточно широкого класса запросов это возможно. Таким образом, подмножество языка, реализуемое через базовые LR-операции, является хорошо оптимизируемым, в то время как расширение этого подмножества за счет включения И-операций и операций поддержки динамического контекста приводит к снижению оптимизируемости.

3.6.1 Привила для базовых операций

В этом разделе рассматриваются выражения, представляющие собой композицию базовых LR-операций. Будем называть такие выражения *базовыми*. Определим систему перезаписи базовых выражений, которую будем сокращенно называть СПС-Б (от "система перезаписи структуры базовых выражений"). Ниже приводится перечень правил перезаписи, составляющих СПС-Б. Приведенные правила осуществляют переход к эквивалентным логическим представлениям. Обоснование последнего утверждения приводится только для правил, для которых это не является очевидным.

$$\text{child}(\text{return}(e_1, \lambda(x|e_2)), \text{test}) \Rightarrow \\ \text{return}(e_1, \lambda(x|\text{child}(e_2, \text{test}))) \quad (3.28)$$

$$\text{child}(\text{seq}(e_1, ..., e_n), \text{test}) \Rightarrow \\ \text{seq}(\text{child}(e_1, \text{test}), ..., \text{child}(e_n, \text{test})) \quad (3.29)$$

$$\text{child}(\text{if}(c, e_1, e_2), \text{test}) \Rightarrow \\ \text{if}(c, \text{child}(e_1, \text{test}), (\text{child}(e_2, \text{test})) \quad (3.30)$$

$$\text{child}(\text{ts}(e_0, \lambda(x|\text{cases}(\text{case}(t_1, e_1), \ldots, \text{case}(t_n, e_n), \text{def}(e_{n+1}))), \text{test}) \Rightarrow \\ \text{ts}(e_0, \lambda(x|\text{cases}(\text{case}(t_1, \text{child}(e_1, \text{test})), \ldots, \\ \text{case}(t_n, \text{child}(e_n, \text{test})), \text{def}(\text{child}(e_{n+1}, \text{test}))))) \quad (3.31)$$

$$\text{child}(\text{element}(e_1, e_2), \text{test}) \Rightarrow \\ \begin{cases} \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{element}, \text{text})x), \text{def}(())))), \\ \text{test} = \text{node}(), x = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{element})x), \text{def}(())))), \\ \text{test} = \text{elem}(*), x = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{element name})x), \text{def}(())))), \\ \text{test} = \text{elem}(\text{name}), x = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{attribute})x), \text{def}(())))), \\ \text{test} = \text{attr}(*), x = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{attribute name})x), \text{def}(())))), \\ \text{test} = \text{attr}(\text{name}), x = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_2, \lambda(x|\text{cases}(\text{case}((\text{text})x), \text{def}(())))), \\ \text{test} = \text{text}(), x = \text{gen} - \text{var}(\text{context}) \end{cases} \quad (3.32)$$

$$\text{child}(\text{attribute}(e_1, e_2), \text{test}) \Rightarrow () \quad (3.33)$$

$$\text{some}(\text{return}(e_0, \lambda(x_1|e_1)), \lambda(x_2|e_2)) \Rightarrow \\ \begin{cases} \text{some}(e_0, \lambda(v|\text{some}(e_1', \lambda(x_2|e_2)))), \\ x_1 \in \text{context}, e_1' = e_1\{x_1 \mapsto v\}_{\text{context}}, v = \text{gen} - \text{var}(\text{context}) \\ \text{some}(e_0, \lambda(x_1|\text{some}(e_1, \lambda(x_2|e_2)))) \end{cases} \quad (3.34)$$

$$\text{some}(\text{element}(e_1, e_2), \lambda(x|e_3)) \Rightarrow$$
$$e_3\{x \mapsto \text{element}(e_1, e_2)\}_{\text{context}} \quad (3.35)$$

$$\text{some}(\text{attribute}(e_1, e_2), \lambda(x|e_3)) \Rightarrow$$
$$e_3\{x \mapsto \text{attribute}(e_1, e_2)\}_{\text{context}} \quad (3.36)$$

$$\text{some}(\text{if}(c, e_1, e_2), \lambda(x|e_3)) \Rightarrow \text{if}(c, \text{some}(e_1, \lambda(x|e_3), \text{some}(e_2, \lambda(x|e_3)) \quad (3.37)$$

$$\text{ts}(\text{seq}(e_1, ...e_n), \lambda(x|e)) \Rightarrow$$
$$\text{seq}(\text{ts}(e_1, \lambda(x|e)), \ldots, \text{ts}(e_n, \lambda(x|e))) \quad (3.38)$$

$$\text{ts}(\text{return}(e_0, \lambda(x_1|e_1)), \lambda(x_2|e_2)) \Rightarrow$$
$$\begin{cases} \text{return}(e_0, \lambda(v|\text{some}(e_1', \lambda(x_2|e_2)))), \\ x_1 \in \text{context},\ e_1' = e_1\{x_1 \mapsto v\}_{\text{context}},\ v = \text{gen} - \text{var}(\text{context}) \\ \text{some}(e_0, \lambda(x_1|\text{some}(e_1, \lambda(x_2|e_2)))) \end{cases} \quad (3.39)$$

$$\text{ts}(\text{ts}(e_0, \lambda(x_1|\text{cases}(\text{case}(t_1, e_1), \ldots, \text{case}(t_n, e_n), \text{def}(e_{n+1})))), f) \Rightarrow$$
$$\begin{cases} \text{ts}(e_0, \lambda(v|\text{cases}(\text{case}(t_1, \text{ts}(e_1', f)), \ldots, \text{case}(t_n, \text{ts}(e_n', f)), \\ \text{def}(\text{ts}(e_{n+1}', f))))), x_1 \in \text{context},\ e_i' = e_i\{x_1 \mapsto v\}_{\text{context}}, \\ v = \text{gen} - \text{var}(\text{context}) \\ \text{ts}(e_0, \lambda(x_1|\text{cases}(\text{case}(t_1, \text{ts}(e_1, f)), \ldots, \text{case}(t_n, \text{ts}(e_n, f)), \\ \text{def}(\text{ts}(e_{n+1}, f))))) \end{cases} \quad (3.40)$$

$$\text{ts}(\text{if}(c, e_1, e_2)), f) \Rightarrow$$
$$\text{if}(c, \text{ts}(e_1, f), \text{ts}(e_2, f)) \quad (3.41)$$

$$\text{return}(\text{return}(e, \lambda(x_1|e_1)), \lambda(x_2|e_2)) \Rightarrow$$
$$\begin{cases} \text{return}(e, \lambda(v|\text{return}(e_1', \lambda(x_2|e_2)))), \\ x_1 \in \text{context}, v = \text{gen} - \text{var}(\text{context}),\ e_1' = e_1\{x_1 \mapsto v\}_{\text{context}} \\ \text{return}(e, \lambda(x_1|\text{return}(e_1, \lambda(x_2|e_2)))) \end{cases} \quad (3.42)$$

$$\text{return}(\text{seq}(e_1, \ldots, e_n), f) \Rightarrow$$
$$\text{seq}(\text{return}(e_1, f), \ldots, \text{return}(e_n, f)) \quad (3.43)$$

$$\text{return}(\text{if}(c, e_1, e_2), \lambda(x|e_3)) \Rightarrow$$
$$\text{if}(c, \text{return}(e_1, \lambda(x|e_3)), \text{return}(e_2, \lambda(x|e_3))) \quad (3.44)$$

$$\textit{return}(\text{ts}(e, \lambda(x_1|\text{cases}(\text{case}(t_1, e_1), \ldots, \text{case}(t_n, e_n), \text{def}(e_{n+1})))),$$
$$\lambda(x_2|e_{n+2})) \Rightarrow$$
$$\begin{cases} \text{ts}(e, \lambda(v|\text{cases}(\text{case}(t_1, \text{return}(e'_1, \lambda(x_2|e_3))), \ldots, \\ \text{case}(t_n, \text{return}(e'_n, \lambda(x_2|e_3))), \text{def}(\text{return}(e'_{n+1}, \lambda(x_2|e_3)))))), \\ x_1 \in \text{context},\ v = \text{gen} - \text{var}(\text{context}),\ e'_i = e_i\{x_1 \mapsto v\}_{\text{context}} \\ \text{ts}(e, \lambda(x_1|\text{cases}(\text{case}(t_1, \text{return}(e_1, \lambda(x_2|e_3))), \ldots, \\ \text{case}(t_n, \text{return}(e_n, \lambda(x_2|e_3))), \text{def}(\text{return}(e_{n+1}, \lambda(x_2|e_3)))))) \end{cases} \quad (3.45)$$

$$\text{return}(\text{element}(e_1, e_2), \lambda(x|e_3)) \Rightarrow$$
$$e_3\{x \mapsto \text{element}(e_1, e_2)\}_{\text{context}} \quad (3.46)$$

$$\text{return}(\text{attribute}(e_1, e_2), \lambda(x|e_3)) \Rightarrow$$
$$e_3\{x \mapsto \text{attribute}(e_1, e_2)\}_{\text{context}} \quad (3.47)$$

$$\text{return}(\text{op}(e_1, \ldots, e_n), \lambda(x|e_{n+1})) \Rightarrow$$
$$e_2\{x \mapsto \text{op}(e_1, \ldots, e_n)\}_{\text{context}}$$
где op - любая операция из
accessor, some, every, empty, aggregate, fo, doc $\quad (3.48)$

$$\text{return}(e, f)\{\text{var?}(e)\} \Rightarrow f\{x \mapsto e\}_{\text{context}} \quad (3.49)$$

$$\text{accessor}(\text{if}(c, e_1, e_2)) \Rightarrow \text{if}(c, \text{accessor}(e_1), \text{accessor}(e_2)) \quad (3.50)$$

Следующие правила описываются в виде рекурсивных функций. Общей чертой этих правил является то, что при применении их к выражению это выражение обходится вглубь с сохранением состояния до тех пор, пока анализируемые подвыражения представлены мета-операциями. Если встречается конструктор, он заменяется на некоторое выражение, не содержащее этого конструктора. Если встречается какая-либо другая операция отличная от мета-операции и конструктора, то обход выражения продолжается без его модификации. Таким образом, основным результатом применения этих правил является избавление от конструкторов.

Правило *предварительного вычисления действительного логического значения (effective boolean value)* применяется ко всем выражениям, которые находятся в тех позициях запроса, где ожидается логическое выражение. К ним относятся следующие позиции, обозначенные символом "?": $\text{if}(?, e, e)$, $\text{some}(e, \lambda(x|?))$, $\text{every}(e, \lambda(x|?))$. Определим это правило в виде рекурсивной функции pre-effective-boolean-value (или, сокращенно, pe) с двумя параметрами expr и state, где expr — анализируемое выражение, state - одно из двух состояний "active" или "passive". Сокращенно будем называть это правило *PE-правилом*.

- Если expr имеет вид $\text{return}(e_1, \lambda(x|e_2))$, то вне зависимости от значения state функция pe возвращает $\text{return}(e_1, \lambda(x|pe(e_2, state)))$.
- Если expr имеет вид $\text{seq}(e_1, \ldots, e_n))$, то вне зависимости от значения state функция pe возвращает $\text{seq}(\text{pe}(e_1, \text{state}), \ldots, \text{pe}(e_n, \text{state}))$.
- Если expr имеет вид $\text{if}(e_1, e_2, e_3))$, то вне зависимости от значения state функция pe возвращает $\text{if}(\text{pe}(e_1, "active"), \text{pe}(e_2, \text{state}), \text{pe}(e_n, state))$.
- Если имеет вид $\text{ts}(e_1, \lambda(x|\text{cases}(\text{case}(tp_1, e_1), \ldots,$ $\text{case}(tp_n, e_n), \text{def}(e_{n+1}))))$, то вне зависимости от значения state функция pe возвращает $\text{ts}(e_1, \lambda(x|\text{cases}(\text{case}(tp_1, \text{pe}(e_1, \text{state})), \ldots,$ $\text{case}(tp_n, \text{pe}(e_n, \text{state})), \text{def}(\text{pe}(e_{n+1}, \text{state})))))$.
- Если expr имеет вид $\text{some}(e_1, \lambda(x|e_2))$, то вне зависимости от значения state функция pe возвращает $\text{some}(e_1, \lambda(x|\text{pe}(e_2, "active")))$.

- Если expr имеет вид every$(e_1, \lambda(x|e_2))$, то вне зависимости от значения state функция pe возвращает every$(e_1, \lambda(x|\text{pe}(e_2, \text{"}active\text{"})))$.
- Если expr имеет вид and(e_1, e_2), то вне зависимости от значения state функция pe возвращает and$(\text{pe}(e_1, \text{"}active\text{"}), \text{pe}(e_2, \text{"}active\text{"}))$.
- Если expr имеет вид or(e_1, e_2), то вне зависимости от значения state функция pe возвращает or$(\text{pe}(e_1, \text{"}active\text{"}), \text{pe}(e_2, \text{"}active\text{"}))$.
- Если expr имеет вид not(e), то вне зависимости от значения state функция pe возвращает not$(\text{pe}(e, \text{"}active\text{"}))$.
- Если expr имеет вид element(e_1, e_2) и state = "*active*", то функция pe возвращает true.
- Если expr имеет вид attribute(e_1, e_2) и state = "*active*", то функция pe возвращает true.
- Иначе будем считать, что $expr = \text{op}(e_1, \ldots, e_n)$ и функция pe возвращает $\text{op}(\text{pe}(e_1, \text{"}passive\text{"}), \ldots, \text{pe}(e_n, \text{"}passive\text{"}))$.

Правило *предварительного вычисления типизированных значений* (typed-value) применяется ко всем выражениям, которые находятся в тех позициях запроса, к которым неявно применяется процедура вычисления типизированного значения. К ним относятся следующие позиции, обозначенные символом "?": fo(?,?), avg(?), max(?), min(?), sum(?). Определим это правило в виде рекурсивной функции pre-typed-value(e,state) (или, сокращенно, pt(e,state)) с двумя параметрами expr и state, где expr - анализируемое выражение, и state может принимать значения "active" или "passive". Сокращенно будем называть это правило *PT-правилом.*

- Если expr имеет вид return$(e_1, \lambda(x|e_2))$, то вне зависимости от значения state функция pt возвращает return$(e_1, \lambda(x|pt(e_2, state)))$.
- Если expr имеет вид seq$(e_1, \ldots, e_n))$, то вне зависимости от значения state функция pt возвращает $\text{seq}(\text{pt}(e_1, state), \ldots, \text{pt}(e_n, state))$.
- Если expr имеет вид if$(e_1, e_2, e_3))$, то вне зависимости от значения state функция pt возвращает $\text{if}(\text{pt}(e_1, \text{"}passive\text{"}), \text{pt}(e_2, state), \text{pt}(e_n, state))$.
- Если expr имеет вид ts$(e_1, \lambda(x|\text{cases}(\text{case}(tp_1, e_1), \ldots,$ $\text{case}(tp_n, e_n), \text{def}(e_{n+1}))))$, то вне зависимости от значения state

функция pt возвращает ts($e_1, \lambda(x|$cases(case(tp_1, pt(e_1, state)), ..., case(tp_n, pt(e_n, state)), def(pt(e_{n+1}, state))))).

- Если expr имеет вид element(e_1, e_2), и state = "*active*", то функция pt возвращает special-string-value-type(pt(e_2, "*active*")).
- Если expr имеет вид attribute(e_1, e_2), и state = "*active*", то функция pt возвращает special-string-value-type(pt(e_2, "*active*")).
- Если expr имеет вид aggregate(e), то вне зависимости от значения state функция pt возвращает aggregate(pt(e, "*active*")).
- Если expr имеет вид fo($e_1, \ldots, e_n$), то вне зависимости от значения state функция pt возвращает fo(pt(e_1, "*active*"), ..., pt(e_n, "*active*")).
- Если expr имеет вид vc($e_1, \ldots, e_n$), то вне зависимости от значения state функция pt возвращает vc(pt(e_1, "*active*"), ..., pt(e_n, "*active*")).
- Иначе будем считать, что $expr$ = op($e_1, \ldots, e_n$), и функция pt возвращает op(pt(e_1, "*passive*"), ..., pt(e_n, "*passive*")).

Правило *предварительного вычисления значения аксессора* применяется ко всем выражениям, которые являются параметрами аксессоров. Определим это правило в виде рекурсивной функции pre-accessor(e,state) (или, сокращенно, pa(e,state)) с двумя параметрами expr и state, где expr - анализируемое выражение, и state может принимать значения "active", "passive", "name" или "node-kind". Будем называть это правило *PA-правилом*.

- Если expr имеет вид return($e_1, \lambda(x|e_2)$), то вне зависимости от значения state функция pa возвращает return($e_1, \lambda(x|$pa(e_2, state))).
- Если expr имеет вид seq($e_1, \ldots, e_n$)), то вне зависимости от значения state функция pa возвращает seq(pa(e_1, state), ..., pa(e_n, state)).
- Если expr имеет вид if(e_1, e_2, e_3)), то вне зависимости от значения state функция pa возвращает if(pa(e_1, "*passive*"), pa(e_2, state), pa(e_n, state)).
- Если expr имеет вид ts($e_1, \lambda(x|$cases(case(tp_1, e_1), ..., case(tp_n, e_n), def(e_{n+1})))), то вне зависимости от значения state функция pa возвращает ts($e_1, \lambda(x|$cases(case(tp_1, pa(e_1, state)), ..., case(tp_n, pa(e_n, state)), def(pa(e_{n+1}, state))))).

- Если expr имеет вид name(e), то вне зависимости от значения state функция ра возвращает extended-name(pa(e,"name")).
- Если expr имеет вид node – kind(e), то вне зависимости от значения state функция ра возвращает extended-node-kind(pa(e,"node-kind")).
- Если expr имеет вид element(e_1, e_2), и state ="name", то функция ра возвращает special-name-type(e_1).
- Если expr имеет вид attribute(e_1, e_2), и state ="name", то функция ра возвращает special-name-type(e_1).
- Если expr имеет вид element(e_1, e_2), и state ="node-kind", то функция ра возвращает special-node-kind-type("element").
- Если expr имеет вид attribute(e_1, e_2), и state ="node-kind", то функция ра возвращает special-node-kind-type("attribute").
- Иначе будем считать, что $expr$ = op($e_1, \ldots, e_n$), и функция ра возвращает op(pp(e_1,"*passive*"), ..., pp(e_n,"*passive*")).

Для оптимизации запроса РЕ-правило, РТ-правило и РА-правило следует применять последовательно в любом порядке к результату применения правил 3.28 -3.50 со значением параметра state, равным "passive" для всех трех правил.

Докажем, что СПС-Б обладает свойством нормальной формы.

Теорема 1 (о существовании нормальной формы) *СПС-Б обладает свойством нормальной формы.*

Доказательство 1 Поскольку РЕ-правило, РТ-правило и РА-правило можно применять после правил 3.28-3.50 и причем за один обход запроса, не будем рассматривать эти правила при доказательстве теоремы.

Таким образом, рассмотрим только правила с 3.28 по 3.50. Для удобства доказательства разделим множество этих правил на три непересекающиеся подмножества, объединение которых дает множество всех рассматриваемых правил:

- К первому подмножеству отнесем правила 3.28,3.29, 3.30,3.31, 3.34,3.37,3.38,3.39, 3.40,3.41,3.42,3.43, 3.44,3.45,3.50.
- Ко второму подмножеству отнесем правила 3.32,3.33.

- К третьему подмножеству отнесем правила 3.35,3.36, 3.46,3.47, 3.48,3.49.

При применении правил первого и второго подмножества количество операций в запросе не увеличивается, а при применении правил третьего подмножества количество операций в запросе хотя и может увеличиваться, но это увеличение не безгранично (оно ограничивается сверху суммой по каждой операции return количества свободных вхождений формального параметра в тело функции, являющейся вторым параметром операции return). Поэтому количество операций в представлении запроса в процессе и результате перезаписи ограничено сверху некоторым числом. Отсюда следует, что отсутствие свойства нормальной формы СПС-Б возможно только в том случае, когда одно правило будет применяться бесконечное число раз. Но это невозможно. Докажем последнее утверждение для каждого подмножества правил в отдельности.

Тот факт, что ни одно из правил первого подмножества не может применяться бесконечное число раз, докажем для правила 3.42 (для остальных правил этого подмножества доказательство аналогично). Для каждой пары вхождений операции return, оказавшихся в сочетании, необходимом для удовлетворения условия применимости этого правила, возможно только единственное применение этого правила, поскольку в результате применения будет получен подзапрос со следующей структурой: $\mathrm{return}(\ldots, \lambda(\ldots|\mathrm{return}(\ldots)))$. Легко видеть, что при применении любого из правил любой группы возможны только два варианта: сохранение положения вхождения операции return внутри тела безымянной функции либо исчезновение этого вхождения. Из последнего утверждения и ограниченности числа вхождений операции return в представление следует конечность числа применений правила 3.42.

Ни одно из правил второго подмножества не может применяться бесконечное число раз, поскольку применение этих правил ведет к уменьшению числа операций в представлении.

Ни одно из правил третьего подмножества не может применяться бесконечное число раз, потому что каждое применение правила ведет к исчезновению операции, наличие которой необходимо для удовлетворения условия применимости правила.

Теорема доказана.

Правила, составляющие СПС-Б, построены таким образом, чтобы

добиться реализации двух видов логической оптимизации: опустить предикат ниже трансформации и избавиться от конструкторов, вычисление которых не требуется для получения результата запроса. Поясним неформально, каким образом достигается реализация этих видов оптимизации. Следствием выбора логического представления и правил отображения XQuery-запроса на его операции является следующее представление применения предиката: $\mathrm{return}(e, \lambda(x|\mathrm{if}(p, e_1, ())))$, где p - некоторый предикат. Опустить предикат ниже трансформации означает избавиться от конструкторов в выражениях e и p. Правила 3.42,3.43,3.44,3.45,3.46,3.47 перемещают конструкторы из выражения e в другие позиции запроса (отметим, что в результате применения этих правил число конструкторов в предикате может расти). Удаление конструкторов из предиката происходит при помощи PE-правила, PT-правила и PA-правила, а также правил 3.32,3.33,3.35,3.36. Избавление от конструкторов, вычисление которых не требуется для получения результата запроса, осуществляется, главным образом, за счет применения правил 3.28,3.29,3.30, 3.31,3.32,3.33.

Докажем это формально. Проведем это доказательство при помощи определения свойств нормальной формы. Для этого сформулируем и докажем теорему о важном свойстве нормальной формы. Опираясь на следствия этой теоремы, покажем, что СПС-Б реализует упомянутые виды оптимизации.

Теорема 2 (о свойстве нормальной формы) *В выражении, находящемся в нормальной форме и не содержащем ошибок несовместимости типов, к результатам конструктора element могут применяться только мета-операции и конструктор element, причем только в позициях, отмеченных знаком "?"* $\mathrm{return}(e, \lambda(x|?))$, $\mathrm{seq}(?, \ldots, ?)$, $\mathrm{if}(e, ?, ?)$, $\mathrm{ts}(e, \lambda(x|\mathrm{cases}(\mathrm{case}(e, ?), \ldots, \mathrm{case}(e, ?), \mathrm{def}(e))))$, $\mathrm{element}(e, ?)$.

Доказательство 2 Доказательство теоремы можно свести к следующей последовательности шагов:

- Шаг 1. Покажем, что в нормальной форме при вычислении значений, которые связываются с переменными, конструкторы не вычисляются. Это необходимо, поскольку к значениям, представленным переменными, может применяться, например,

операция child, и, следовательно, если при вычислении такого значения используются результаты вычисления конструкторов, то условие теоремы не будет выполнено.

- Шаг 2. Выберем из LR-операций те, которые могут применяться в нормальной форме непосредственно к конструкторам. Обозначим полученное множество C_1. Отметим, что благодаря предыдущему шагу доказательства в C_1 не имеет смысла включать операции, которые применяются непосредственно к значениям, представленным переменными.

- Шаг 3. Построим множество C_c всех операций, которые могут применяться к результатам конструкторов. Для этого построим последовательность $\{C_i\}$. В качестве C_1 возьмем множество, построенное на прошлом шаге. Далее построим последовательность рекурентно по следующим правилам. Допустим, что C_{i-1} построено. Для каждой операции $c \in C_{i-1}$ выберем подмножество LR-операций, которые могут применяться к c в нормальной форме с учетом того, что операция c уже применяется к результатам вычисления конструктора. Пусть C'_i является объединением всех таких множеств. Если $C'_i - C_{i-1} = \emptyset$ (т.е. прироста операций не было), то в качестве C_i возьмем C'_i, и построение последовательности прекращаем, поскольку C_i является искомым множеством C_c. В противном случае в качестве C_i возьмем $C'_i \cup C_{i-1}$ и продолжим процесс построения последовательности $\{C_i\}$. Очевидно, что, благодаря конечности множества LR-операций, описанный процесс построения последовательности является конечным и приведет нас к искомому множеству.

- Шаг 4. Совпадение множества C_c со множеством операций, перечисленных в условиях теоремы, доказывает теорему.

Для удобства понимания следующей части доказательства приведем перечень всех LR-операций: return, seq, if, ts, element, attribute, accessor (name, node-kind), child, descendant, some, every, empty, aggregate (cont,avg,min,max,sum), fo, doc.

Проделаем перечисленные выше шаги.

Шаг 1. Чтобы показать, что при вычислении значения переменных не вычисляются конструкторы, достаточно доказать, что в выражениях,

определяющих значения переменных, не содержатся конструкторы. Связывание переменных со значениями происходит только в операциях return, ts, some, every. При этом значение переменной определяет первый операнд. Верно то утверждение, что первый операнд операций return, ts, some, every может быть представлен только операциями child, descendant. Оставшиеся операции, к которым относятся return, seq, if, ts, element, attribute, accessor, some, every, empty, aggregate, fo, doc, var, const, не могут представлять первый операнд по причине наличия правил 3.42, 3.43, 3.44, 3.45, 3.46, 3.47, 3.48. В свою очередь, операнды операций child и descendant могут быть выражены только через эти же операции или операции doc и var, благодаря наличию правил 3.28, 3.29, 3.30, 3.31, 3.32, 3.33, исключающих другие возможности, и благодаря тому, что операции some, every, empty, aggregate, fo возвращают атомарные значения. Таким образом, в выражении, определяющем значение переменной, не может содержаться конструктор, что доказывает утверждение шага один.

Шаг 2. Для построения C_1 необходимо найти операции, операнды которых могут быть представлены конструктором element[9]. Получаем $C_1 = \{\text{return}(e, \lambda(x|?)), \text{seq}(?, \ldots, ?), \text{if}(e, ?, ?), \text{ts}(e, \lambda(x|\text{cases}(\text{case}(e, ?), \ldots, \text{case}(e, ?), \text{def}(e)))), \text{element}(e, ?)\}$. Возможности появления в C_1 других операций исключается наличием правил 3.32,3.35,3.46.

Шаг 3. Для осуществления этого шага сначала для каждой операции $c \in C_1$ построим множество LR-операций, которые могут применяться к c в нормальной форме с учетом того, что c применяется к результатам вычисления конструктора element. Для каждой c мы получим одно и тоже множество $\{\text{return}(e, \lambda(x|?)), \text{seq}(?, \ldots, ?), \text{if}(?, ?, ?), \text{ts}(e, \lambda(x|\text{cases}(\text{case}(e, ?), \ldots, \text{case}(e, ?), \text{def}(e)))), \text{element}(?, ?)\}$. Для операции return это множество не будет содержать других операций благодаря наличию правил 3.42,3.39,3.28,3.34, PE-правилу, PT-правилу и PA-правилу. Для операции seq это множество не будет содержать других операций благодаря наличию правил 3.29,3.38,3.43,3.45,PE-правилу, PT-правилу и PA-правилу. Для операции if это множество не будет содержать других операций благодаря наличию правил 3.30,3.37,3.41,3.44,3.50, PE-правилу, PT-правилу и PA-правилу. Для операции ts это множество не будет содержать других операций благодаря наличию правил 3.31,3.40,3.45, PE-правилу, PT-правилу и PA-правилу. Для операции element это множество не будет содержать

[9]Переменные можно не рассматривать по результатам предыдущего шага.

других операций благодаря наличию правил 3.32,3.35,3.46, PE-правилу, PT-правилу и PA-правилу. Тогда получаем, что $C_2' = \{\text{return}(e, \lambda(x|?)),\ \text{seq}(?, \ldots, ?),\ \text{if}(e, ?, ?),\ \text{ts}(e, \lambda(x|\text{cases}(\text{case}(e, ?), \ldots, \text{case}(e, ?), \text{def}(e)))),\ \mathit{element}(e, ?)\}$. Поскольку $C_2' - C_1 = \emptyset$, то $= C_2 = C_2'$, и искомое множество найдено.

Шаг 4. Множество C_c, содержащее все операции, которые могут применяться к результатам конструкторов, совпадает с множеством, указанным в формулировке теоремы.

Теорема доказана.

Мета-операции и конструктор XML-элементов обладают тем важным свойством, что, принимая на вход x-последовательность в качестве значения одного из параметров, отмеченных в формулировке теоремы, эти операции никоим образом не анализируют эту x-последовательность и возвращают ее в некоторой структуре данных (например, как подпоследовательность новой x-последовательности или содержание XML-элемента). Поэтому можно сформулировать следующее следствие теоремы: результат вычисления любого конструктора в запросе, который находится в нормальной форме, появится в результате этого запроса. Последнее утверждение приводит к двум дополнительным следствиям:

- Поскольку результаты вычисления операций в предикате не может появиться в результате запроса, можно сделать вывод, что при вычислении предиката не используются результаты вычисления конструкторов. Другими словами, приведенные правила перезаписи достигают цели опускания предикатов ниже конструкторов, т.е. порядок вычисления запроса таков, что сначала применяются предикаты, которые выражены в терминах исходных структур, а затем производится трансформация, под которой понимается построение новых XML-элементов и XML-атрибутов.

- Вычисление выражение, находящегося в нормальной форме, не приводит к построению XML-элементов, которые не требуются для получения результата выражения. Это означает отсутсвие в нормальной форме конструкторов, вычисление которых не требуется для получения результата запроса.

3.6.2 Привила для И-операций

В предыдущем разделе были рассмотрены методы оптимизации выражений, состоящих из базовых операций логического представления. Расширим множество рассматриваемых операций И-операциями. К таким операциям относятся: union, intersect, except, ddo. Определение синтаксиса и семантики этих операций можно найти в 3.2. Очевидно, что расширение множества операций делает недостаточным набор правил СПС-Б для того, чтобы опускать предикаты ниже конструкторов и избавляться от конструкторов, в вычислении которых нет необходимости. Будем стремиться построить на базе СПС-Б такую систему СПС-И (Система Перезаписи Структуры выражений, содержащих И-операции), которая бы реализовывала два таких вида логической оптимизации при наличии в выражении И-операций.

Перед тем, как приступить к построению СПС-И, выделим важный класс выражений, результат вычисления которых состоит только из узлов с уникальными идентификаторами, отличными от уникальных идентификаторов всех узлов, участвующих в запросе. Выделение такого класса выражений поможет лаконично сформулировать правила перезаписи, которые составляют СПС-И. Для этого, во-первых, введем вспомогательную тождественную на x-последовательности операцию uid (сокращение от Unique IDentity), которую будем использовать для того, чтобы отметить, что все узлы (причем, на любом уровне вложенности), составляющие результат этой операции, имеют идентификаторы, не равные друг другу и не равные ни одному идентификатору, который может встретиться при вычислении запроса. Теперь модифицируем правило 3.32, применяя к результатам преобразований операцию uid. Такая модификация правила имеет смысл, поскольку конструктор производит глубокое копирование с генерацией новых идентификаторов. Во-вторых, выделим подмножество LR-операций и выделим знаком "?" некоторые параметры мета-операций, конструкторов (которые в этом контексте интересны тем, что в результате вычисления строятся узлы с уникальными в рамках запроса идентификаторами), операций, возвращающих атомарные значения, и множественных операций (которые в данном контексте рассматриваются как мета-операции): return$(e, \lambda(x|?))$, seq$(?, \ldots, ?)$ (в том числе ()), if$(e, ?, ?)$, ts$(e, \text{cases}(\text{case}(e, ?), \ldots, \text{case}(e, ?), def(?)))$, element$(e, e)$, attribute$(e, e)$,

accessor(e), some(e, e), every(e, e), empty(e), aggregate(e), fo(e, e), doc(e), значение атомарного типа, union($?, \ldots ?$), intersect($?, ?$), except($?, ?$), node $-$ equal(e, e), uid(e).

Определение 17 *Будем называть выражение* uid-генератором*, если это выражение является композицией в выделенных параметрах указанных выше операций.*

При таком введении uid-генератора можно утверждать, что его результат обладает тем же свойством уникальности идентификаторов всех составляющих его узлов, что и результат операции uid.

Перейдем теперь к построению СПС-И для операций union, intersect, except, ddo. Начнем с операции ddo. Частое появление этой операции в представлении запросов объясняется тем, что она используется для реализации семантики XPath-выражений, поскольку требуется, чтобы любое XPath-выражение возвращало x-последовательность узлов без дубликатов и в порядке появления узлов в документе. Будем стремиться максимально избавиться от этой операции в представлении, что будет приводить представление к виду, при котором возможна реализация двух необходимых видов логической оптимизации при помощи правил СПС-Б. Это также позволит уменьшить время выполнения запроса за счет избавления от такой дорогостоящей операции как ddo.

Чтобы определить правила, позволяющие избавиться от операции ddo в тех случаях, когда это возможно, рассмотрим выражения двух видов: ddo(op(e)) и op(ddo(e)), где op - любая LR-операция.

Рассмотрим выражение вида ddo(op(e)), где op - любая LR-операция. Для таких выражений верны следующие правила, которые позволяют избавиться от применения операции ddo:

- ddo(e) $= e$, если e есть композиция операций child, descendent, var, doc
- ddo(e) $= e$, если e есть uid-генератор
- ddo(child(e, text())) $=$ child(e, text())
- ddo(e) $= e$, если ddo(e) находится в такой позиции запроса, где на основе результата этой операции будет вычисляться действительное логическое значение (например, if(ddo(e), $e1, e2$) $=$ if($e, e1, e2$)). Это действительно верно, потому что ни одно из значений (), false,

"", числовое значение равное нулю, в результате применения к которым правила вычисления действительного логического значения вырабатывается false, не может быть получено в результате применения ddo к e, если e не равно одному из этих значений.

Рассмотрим теперь выражения вида op(ddo(e)), где op - любая LR-операция. Для таких выражений верны следующие правила, которые позволяют избавиться от применения операции ddo:

- some(ddo($e1$), $e2$) = some($e1$, $e2$)
- every(ddo($e1$), $e2$) = every($e1$, $e2$)
- union(..., ddo(e), ...) = union(..., e, ...)
- intersect(ddo($e1$), ddo($e2$)) = intersect($e1$, $e2$)
- except(ddo($e1$), ddo($e2$)) = except($e1$, $e2$)
- ddo(ddo(e)) = e

Замечание 5 *Предпоследние три правила верны, потому что порядок результата операций union, intersect и except не определен, и они удаляют дубликаты.*

Понятно, что приведенные правила во многих случаях позволяют избавиться от операции ddo, но это возможно не всегда, что может препятствовать избавлению от конструкторов, вычисление которых не требуется для получения результата запроса, и опусканию предиката.

Рассмотрим теперь операции union, intersect, except. Они по своей природе близки к операции seq, поскольку, как и seq, строят по некоторым правилам выходную x-последовательность из элементов входных x-последовательностей без модификации этих элементов. Немного упрощая ситуацию, можно сказать, что правила из СПС-Б, в которых фигурирует операция seq, позволяют применить к каждому аргументу seq операцию, применяемую к результатам seq, то есть произвести следующее преобразование $\mathrm{op}(\mathrm{seq}(e_1, \ldots, e_n)) \Rightarrow \mathrm{seq}(\mathrm{op}(e_1), \ldots, \mathrm{op}(e_n))$. Нетрудно видеть, что преобразование такого вида является крайне необходимым для того, чтобы опустить предикат и избавиться от конструкторов, в вычислении которых нет необходимости. Поэтому подобное преобразование необходимо и при замене операции

seq на операции union, intersect, except. К сожалению, в общем случае это невозможно. Рассмотрим, например, выражение вида $\text{return}(\text{union}(e_1, ..., e_n), \lambda(x|\text{if}(p, \text{element}(\ldots), ())))$. Если бы вместо операции union присутствовала операция seq, то можно было бы применить правило 3.43 и тем самым опустить предикат p ниже операции seq. К сожалению, это невозможно для операции union, поскольку эта операция удаляет дубликаты при объединении входных x-последовательностей, и выполнение такого преобразования привело бы к выражению, в котором, возможно, содержится большее число узлов. Это действительно так, поскольку конструктор XML-элементов применялся бы ко всем элементам из x-последовательностей, полученных в результате вычисления выражений $e_1, \ldots, e_n$, которые удовлетворяют условию p, включая и дубликаты. Подобные преобразования возможны в том случае, если все аргументы операций union, intersect, except выражены через uid-генераторы, потому что в этом случае любую из операций можно заменить на seq, не изменив при этом значение выражения.

Таким образом, рассмотрение И-операций показывает, что их появление в выражении существенно препятствует его оптимизации, но для многих выражений оптимизация все-таки возможна.

3.6.3 Правила для поддержки запросов, обращающихся к динамическому контексту

В языке XQuery предполагается, что предикаты XPath выражений вычисляются в так называемом динамическом контексте, к данным которого можно получить доступ из предиката (о динамическом контексте смотри страницу 74). Контекст состоит из нескольких компонентов, доступ к которым возможен через вызовы предопределенных функций. Некоторые компоненты контекста являются *глобальными* для всего запроса (то есть их значение одно и тоже в любой позиции запроса), а часть — нет (будем называть такие компоненты *локальными*). Поэтому значение вызова функции, возвращающей глобальный компонент контекста, является одним и тем же при нахождении вызова в любой позиции запроса, в то время как значение вызова функции, возвращающей локальную компоненту, определяется позицией вызова в запросе. Поскольку при перезаписи запроса позиция вызова функции может измениться, возможен переход к неэквивалентному запросу в результате перезаписи. В этом

разделе предлагается подход к перезаписи выражений, содержащих вызовы функций для доступа к локальным компонентам контекста.

Предлагаемый подход основан на введении операций генерации компонентов контекста (context-generator или сокращенно con-gen) и связывания их результатов с переменными. При этом вызовы функций доступа к компонентам контекста заменяются на соответствующие переменные. Тем сам достигается привязка использования компонентов контекста к выражению, которое определяет контекст. Перейдем к определению операций генерации контекста.

$\mathrm{con-gen-1}(e, m)$, принимая в качестве значения параметра e x-последовательность атомов (e_i), возвращает x-последовательность уний вида $[e_i, m + i]$. Другими словами, по каждому объекту входной x-последовательности строится уния, содержащая этот объект и его порядковый номер в x-последовательности, увеличенный на значение параметра m.

$\mathrm{con-gen-2}(e, m)$, принимая в качестве значения параметра e x-последовательность атомов (e_i), возвращает x-последовательность уний вида $[e_i, m + i, \mathrm{count}(e)]$. Другими словами, по каждому объекту входной x-последовательности строится уния, содержащая этот объект; его порядковый номер в x-последовательности, увеличенный на значение параметра m; и размер всей входной x-последовательности.

Определим правила представления XPath-выражений, содержащих в предикатах функции обращения к контексту, через операции генерации контекста (с использованием операций con-gen-1 и con-gen-2). Применение этих правил к XPath-выражению e будем обозначать $[e]_{\mathrm{lr}}$. Для этого достаточно определить только правило для XPath-выражения из двух шагов с одним предикатом вида $e_1/a[e_2]$, поскольку более сложные XPath-выражения могут быть представлены в логическом представлении при помощи рекурсивного применения этого правила. Выделим два случая: (1) e_2 не содержит вызов функции last(); (2) e_2 содержит вызов функции last().

В первом случае правило будем следующим:

$$[e_1/a[e_2]]_{\mathrm{lr}} = \mathrm{return}(\mathrm{con-gen-1}(\mathrm{child}(e_1, [a]_{\mathrm{lr}}), m), \lambda(e, i|\mathrm{if}(e_2', e, ()))), \text{ где } e_2' = [e_2]_{\mathrm{lr}} \{\text{текущий item} \mapsto e, position() \mapsto i\}_{context} \quad (3.51)$$

Во втором случае:

$$[e_1/a[e_2]]_{\text{lr}} = \text{return}(e_1, \lambda(v|$$
$$\text{return}(\text{con} - \text{gen} - 2(\text{child}(e_1, [a]_{\text{lr}}), m),$$
$$\lambda(e, i, c|\text{if}(e_2', e, ()))))), \text{ где } e_2' = [e_2]_{\text{lr}},$$
$$\{\text{текущий item} \mapsto e, position() \mapsto i, last() \mapsto c\}_{context} \quad (3.52)$$

При расширении поддерживаемого подмножества языка XQuery возможностью работы с контекстом (за счет введения операций генерации контекста) мы стремились к тому, чтобы при оптимизации запросов из расширенного подмножества реализовывались все виды оптимизации, реализации которых удалось добиться для базового подмножества (как для наиболее оптимизируемого). Для этого необходимо сохранить все полезные свойства нормальной формы. Рассмотрим, на какие правила может повлиять возможность появления генераторов контекста в логическом представлении запроса. Чтобы сузить множество правил, подлежащих рассмотрению, заметим, что введенные операции могут появляться в результатах трансляции XPath-выражений в логическое представление только в позиции ? в выражении вида return(?,f). Следовательно, необходимо рассматривать только такие правила, которые анализируют первый аргумент return. Оказывается, что для части таких правил можно ввести новые правила, которые будут осуществлять преобразования, позволяющие добиться тех же результатов при наличии операции генерации контекста. Ниже приводится перечень таких правил. С целью сократить число перечисляемых правил заметим, что правило, верное для операции gen-con-2, верно и для gen-con-1 с учетом отличия арности уний, возвращаемых в x-последовательности.

$$\text{return}(\text{gen} - \text{con} - 2(e_1, m), \lambda(x, i, c|e_2))\{\text{var?}(e_1)\} \Rightarrow$$
$$e_2\{x \mapsto e_1, i \mapsto 1, c \mapsto 1\}_{\text{context}} \quad (3.53)$$

$$\text{return}(\text{gen} - \text{con} - 2((), m), f) \Rightarrow () \quad (3.54)$$

$$\begin{aligned}&\text{return}(\text{gen}-\text{con}-1(\text{seq}(e_1,\ldots,e_n),m),f)\Rightarrow\\&\text{seq}(\text{return}(\text{gen}-\text{con}-1(e_1,m),f),\ldots,\text{return}(\text{gen}-\text{con}-1(e_n,m),f))\end{aligned}\tag{3.55}$$

Отметим, что последнее правило верно только для gen-con-1 и неверно для gen-con-2.

$$\begin{aligned}&\text{return}(\text{gen}-\text{con}-1(\text{return}(e,\lambda(x_1|e_1))),\lambda(x_2|e_2))\Rightarrow\\&\begin{cases}\text{return}(e,\lambda(v|\text{return}(\text{gen}-\text{con}-1(e_1',m),\lambda(x_2|e_2)))),\\ x_1\subseteq\text{context},\ v=\text{gen}-\text{var}(\text{context}),\ e_1'=e_1\{x_1\mapsto v\}_{\text{context}}\\ \text{return}(e,\lambda(x_1|\text{return}(\text{gen}-\text{con}-1(e_1,m),\lambda(x_2|e_2))))\end{cases}\end{aligned}\tag{3.56}$$

Отметим, что последнее правило верно только для gen-con-1 и неверно для gen-con-2.

$$\begin{aligned}&\text{return}(\text{gen}-\text{con}-2(\text{element}(e_1,e_2),m),\lambda(x,i,c|e_3))\Rightarrow\\&\qquad e_3\{x\mapsto\text{element}(e_1,e_2),i\mapsto 1,c\mapsto 1\}_{\text{context}}\end{aligned}\tag{3.57}$$

$$\begin{aligned}&\text{return}(\text{gen}-\text{con}-2(\text{attribute}(e_1,e_2),m),\lambda(x,i,c|e_3))\Rightarrow\\&\qquad e_3\{x\mapsto\text{attribute}(e_1,e_2),i\mapsto 1,c\mapsto 1\}_{\text{context}}\end{aligned}\tag{3.58}$$

$$\begin{aligned}&\text{return}(\text{gen}-\text{con}-2(\text{if}(c,e_1,e_2),m),\lambda(x,i,l|e_3))\Rightarrow\\&\qquad\text{if}(c,\text{return}(\text{gen}-\text{con}-2(e_1,m),\lambda(x,i,l|e_3)),\\&\qquad\qquad\text{return}(\text{gen}-\text{con}-2(e_2,m),\lambda(x,i,l|e_3)))\end{aligned}\tag{3.59}$$

$$\text{return}(\text{gen}-\text{con}-1(\text{ts}(e,\lambda(x_1|\text{cases}(\text{case}(t_1,e_1),\ldots,\\ \text{case}(t_n,e_n),\text{def}(e_{n+1})))),m),\lambda(x_2,i|e_3))\Rightarrow$$

$$\begin{cases} \text{ts}(e,\lambda(v|\text{cases}(\text{case}(t_1,\text{return}(\text{gen}-\text{con}-1(e'_1,m),\lambda(x_2,i|e_3))),\ldots,\\ \text{case}(t_n,\text{return}(\text{gen}-\text{con}-1(e'_n,m),\lambda(x_2,i|e_3))),\\ \text{def}(\text{return}(\text{gen}-\text{con}-1(e'_{n+1},m),\lambda(x_2,i|e_3))))),\\ x_1\subseteq\text{context},\ v=\text{gen}-\text{var}(\text{context}),\ e'_i=e_i\{x_1\mapsto v\}_{\text{context}}\\ \text{ts}(e,\lambda(x_1|\text{cases}(\text{case}(t_1,\text{return}(\text{gen}-\text{con}-1(e_1,m),\lambda(x_2,i|e_3))),\ldots,\\ \text{case}(t_n,\text{return}(\text{gen}-\text{con}-1(e_n,m),\lambda(x_2,i|e_3))),\\ \text{def}(\text{return}(e_{n+1},\lambda(x_2,i|e_3)))))) \end{cases} \tag{3.60}$$

Легко видеть, что эти правила, по сути, повторяют (с учетом возможности наличия con-gen-1 или con-gen-2) правила 3.49, 3.43, 3.42, 3.46, 3.47, 3.44, 3.45. Но в двух случаях, когда необходимо применить правила 3.55 и 3.56 при замене gen-con-1 на gen-con-2, невозможно добиться тех же свойств нормальной формы. Очевидно, что понижение уровня оптимизируемости запроса, использующего данные из контекста, обусловлено самим языком, а не неудачностью предложенного способа поддержки таких запросов. Действительно, в двух вышеупомянутых случаях нельзя "опустить верхний return ниже return и seq", если для вычисления второго аргумента "верхнего" return необходимо вычислить количество элементов в результате "нижнего" return или seq. Если "верхний" return, например, содержит предикат, то в этих случаях предикат не будет опущен. Одно из возможных усовершенствований основано на том, что по анализу второго аргумента нижнего return или, соответственно, аргументов операции seq, в некоторых случаях можно точно определить количество элементов, получаемых в результате вычисления этих выражений. Например, если упомянутые выражения представляют собой конструкторы элементов или применение операции seq к конструкторам элементов, то при любом связывании переменных, анализируя выражения, можно вычислить количество элементов в значении таких выражений. Тем не менее, описанное улучшение не позволяет решить задачу в общем случае.

3.7 Повышение уровня декларативности представления запроса

При помощи правил, описываемых в этом разделе, осуществляется переход от итераторов (таких как return и select) к операциям соединения, описанным в разделе 3.3. По результатам такого перехода реализуются несколько видов логической оптимизации. Перечислим каждое из правил, сопровождая их необходимыми комментариями и в заключении раздела покажем, какие виды логической оптимизации могут быть реализованы при помощи этих правил.

Введем обозначения, которые будут использоваться при определении правил. $fv(x)$ -возвращает все свободные переменные в выражении x. $conj(x)$ - возвращает множество конъюнктов после приведения логического выражения x к конъюнктивной нормальной форме. Будем использовать запись $\lambda(v|e)$, когда имена и число формальных параметров не являются существенными. В этом случае под v понимается некоторый список из одной или более переменных. Если необходимо указать имена формальных параметров и их число, будем использовать запись вида $\lambda((x_1, x_2)|e)$, в которой имена переменных перечисляются в круглых скобках через запятую.

$$\text{some}(e, f) \Rightarrow \text{exist}(\text{select}(e, f)) \tag{3.61}$$

Правило 3.61 позволяет перейти к формулировке запроса, более удобной для определения других правил.

$$\text{if}(c, e_1, e_2)\{e_1, e_2\text{– логические выражения}\}$$

$$\Rightarrow$$

$$\begin{cases} \text{and}(\text{not}(c), e_2), & e_1 = false \\ \text{and}(c, e_1), & e_2 = false \\ \text{and}(\text{or}(c, e_2), \text{and}(\text{or}(e_1, e_2), \text{or}(e_1, \text{not}(c)))) & \end{cases} \tag{3.62}$$

Правило 3.62 позволяет перейти к представлению логических выражений в конъюнктивной нормальной форме.

$$\text{return}(e_1, \lambda(v|\text{if}(c, e_2, e_3)))$$

$$\Rightarrow$$

$$\begin{cases} \text{return}(\text{select}(e_1, \lambda(v|c)), \lambda(v|e_2)) & \text{fv}(c) \subseteq v, e_3 = () \\ \text{return}(\text{select}(e_1, \lambda(v, \text{not}(c))), \lambda(v|e_3)) & \text{fv}(c) \subseteq v, e_2 = () \end{cases} \quad (3.63)$$

Правило 3.63 позволяет перейти к формулировке запроса, более удобной для определения других правил.

$$\text{return}(e_1, \lambda(v_1|\text{return}(e_2, \lambda(v_2| \ldots \text{return}(e_n, \lambda(v_n|\text{if}(p, e_{n+1}, ()))))))))$$

$$\Rightarrow$$

$$\text{return}(\text{xjoin}(\lambda(\cup v_i|p_m), \ldots$$

$$\text{xjoin}(\lambda(v_1 \cup v_2|p_3), \text{select}(e_1, \lambda(v_1|p_1)), \text{select}(e_2, \lambda(v_2|p_2))) \ldots, e_n), \lambda(v|e_{n+1})),$$

где $\cup_{i=1}^{m} p_i = conj(p)$, select применяется только к тем $e_i (i = 1 \ldots n)$

для которых $\exists p_j \in conj(p) \wedge fv(p_j) \subseteq v_i$ (3.64)

В результате применения правила 3.64 получается представление, которое в ряде случаев является более оптимальным, чем исходное. Во-первых, переход от вложенных итераторов return к операциям xjoin, повышает уровень декларативности запроса. Во-вторых, разбиение предиката p на несколько предикатов и распределение последних между операциями select и xjoin способствует опусканию предикатов. В третьих, в исходной формулировке запроса выражения e_i $(i = 2 \ldots n)$ вычисляются многократно по причине наличия внешних операций return, хотя в этом может не быть необходимости, поскольку эти выражения могут не зависеть от переменных, определенных во внешних операциях return. Применение правила позволяет избежать многократных вычислений.

$$\text{select}(e_1, \lambda(v_1|\text{and}(p_1, \text{exist}(\text{select}(e_2, \lambda(v_2|p_2))))))$$
$$\{v_1 \cap fv(e_2) = \emptyset \wedge <p_2 \text{ не содержит подзапроса}>\}$$
$$\Rightarrow$$
$$\text{xproject}(\text{select}(\text{xjoin}(\text{select}(e_1, \lambda(v_1|c_1)), \text{select}(e_2, \lambda(v_2|c_2)),$$
$$\lambda(v_1 \cup v_2|c_3)), \lambda(v_1 \cup v_2|c_4)), (1 \ldots count(v_1 \cup v_2)))$$

где

$$\text{conj}(c_1) = \{x \in \text{conj}(p_2) \cup y \in \text{conj}(p_1)|\text{fv}(x) \subseteq v_1\}$$
$$\text{conj}(c_2) = \{x \in \text{conj}(p_2)|\text{fv}(x) \subseteq v_2\}$$
$$\text{conj}(c_3) = \{x \in \text{conj}(p_2)|\text{fv}(x) \subseteq (v_1 \cup v_2) \wedge \text{fv}(x) \cap v_1 \neq \emptyset \wedge \text{fv}(x) \cap v_2 \neq \emptyset\}$$
$$\text{conj}(c_4) = \{x \in \text{conj}(p_1)|\text{fv}(x) - v_1 \neq \emptyset\} \cup \{x \in \text{conj}(p_2)|\text{fv}(x) - (v_1 \cup v_2) \neq \emptyset\} \quad (3.65)$$

Правило 3.65 выражает запрос с подзапросами, вложенными в предикат операции select, через операции соединения и проекции. При этом полученная в результате перезаписи формулировка обладает рядом преимуществ, аналогичных тем, которые обеспечивает применение правила 3.64: повышается уровень декларативности формулировки запроса, опускается предикат (благодаря появлению select над e_1), избегаются многократные вычисления e_2. При применении правила проверяется, что предикат p_2 не содержит вложенного подзапроса. Рассмотрим случай, когда предикат p_2 содержит вложенный подзапрос, и проследим, к применению какой последовательности правил это приведет. В этом случае правило 3.65 будет применяться к самому вложенному подзапросу. В результате внешняя операция самого вложенного подзапроса (select) заменяется на xproject(select(xjoin(...),<предикат, который зависит от переменных, определенных во внешней операции select>)). Поскольку к результату select в исходном запросе применялась операция exist (так как операция select была частью вложенного подзапроса), появляется формулировка exist(xproject(select(xjoin(...),<предикат, который зависит от переменных, определенных во внешнем запросе>))), к которой применимо правило 3.66, удаляющее операцию xproject. Применение правила 3.66 приводит к формулировке, к которой вновь применимо правило 3.65. Таким образом, последовательное применение правил приводит к перезаписи подзапросов, вложенных в предикат, к формулировке с последовательным применением операций соединения.

В результате последнего применения правила 3.65, получаем выражение вида xproject(xjoin(...)) (между операциями xproject и xjoin отсутствует select, поскольку уже переписан последний вложенный подзапрос), к которому применимо правило 3.67. Применение этого правила завершает процесс перезаписи вложенных в предикат подзапросов с произвольной глубиной вложенности.

$$\text{exist}(\text{xproject}(e, ProjectSpec)) \Rightarrow \text{exist}(e) \tag{3.66}$$

Смотри комментарий к правилу 3.65.

$$\text{xproject}(\text{xjoin}(e_1, e_2, p), ProjectSpec) \quad \{ProjectSpec = (1 \ldots <\text{арность уний в } e_1>)\} \quad \Rightarrow \quad \text{semixjoin}(e_1, e_2, p) \tag{3.67}$$

Смотри комментарий к правилу 3.65.

Обсуждение правил построенной системы перезаписи позволяет сделать вывод, что эта система реализует следующие виды логической оптимизации: повысить декларативность представления запроса, опустить предикаты ниже итераций. Кроме того, применение правил построенной системы перезаписи позволяет избавиться от повторных вычислений выражений во вложенных итераторах.

3.8 Результаты использования логического оптимизатора

Все описанные в этой работе методы логической оптимизации, кроме правил для И-операций и правил для поддержки запросов, обращающихся к динамическому контексту, были полностью реализованы. Эта реализация была использована в качестве модуля логической оптимизации в исследовательском прототипе системы виртуальной интеграции BizQuery, которая разрабатывалась в Институте системного программирования РАН. Этот прототип, включая упомянутый модуль оптимизации, был взят за основу разрабатываемой в момент написания работы промышленной версии системы. Отсутствие реализации правил

для И-операций и правил для поддержки запросов, обращающихся к динамическому контексту, объясняется, главным образом, тем, что поддержка этих операции в системах виртуальной интеграции существенно затруднена по ряду причин теоретического и реализационного характера. Обсуждение этих причин выходит за рамки данной работы. Логический оптимизатор был реализован на языке Scheme. Размер реализации составляет около 6500 строк кода.

Для проверки на практике эффективности разработанных методов логической оптимизации было проведено большое число экспериментов с использованием их реализации. В настоящем разделе приводятся результаты только небольшого числа наиболее показательных экспериментов. Все эксперименты проводились на компьютере с процессором Intel Pentium 4 с частотой 1.5 ГГц и оперативной памятью объемом 512 МБ. Использовалась XML база данных размером около 10 МБ. В таблице 3.1 приведены результаты экспериментов. Указывается время выполнения запроса с включенным и отключенным оптимизатором и время работы оптимизатора. При оптимизации первых трех запросов осуществляется следующий вид оптимизации: избавиться от конструкторов, вычисление которых не требуется для получения результата запроса. Различия в отношении времени с включенным оптимизатором ко времени с выключенным оптимизатором для разных запросов этой группы объясняются разным количеством конструкторов, устраненных из запроса в результате оптимизации. При оптимизации запросов с четвертого по шестой осуществляются следующие виды оптимизации: опустить предикат ниже конструкторов и опустить предикат ниже вложенных итераторов. Различия в отношении времени с включенным оптимизатором ко времени с выключенным оптимизатором для разных запросов этой группы объясняются различной селективностью опускаемых предикатов. Чем выше селективность опускаемого предиката, тем меньше обсуждаемое отношение (т.е. оптимизация менее действенна). Кроме того, все запросы содержат вызовы рекурсивных функций, без избавления от которых было бы невозможно осуществить перечисленные виды оптимизации. Поэтому при оптимизации всех запросов осуществляется вид оптимизации "произвести открытую вставку тел XQuery-функций" при этом задействуется вид оптимизации "вычислить подвыражения в статике, когда это возможно". Проведенные эксперименты не демонстрируют эффективность вида

оптимизации "повысить уровень декларативности запроса", поскольку этот вид оптимизации используется, главным образом, для качественного улучшения запроса, которое требуется для последующего этапа физической оптимизации.

Номер запроса	Вр. с оптим.	Вр. без оптим.	Вр. оптим.
1	10.30	90.20	0.04
2	3.04	75.01	0.05
3	1.40	84.71	0.09
4	2.48	63.18	0.04
5	1.60	67.42	0.09
6	50.01	70.01	0.05

Таблица 3.1: Результаты экспериментов с логическим оптимизатором XQuery-запросов

Результаты проведенных экспериментов позволяют сделать следующие выводы:

- Разработанные методы логической оптимизации позволяют на несколько порядков снизить общее время выполнения XQuery-запросов.
- Время выполнения оптимизации незначительно по сравнению с временем выполнения запроса.

3.9 Выводы

Основным выводом из произведенного автором формального обоснования предложенных в работе методов логической оптимизации XQuery-запросов является то, что уровень оптимизируемости запросов различен в зависимости от рассматриваемого подмножества языка. Так, наиболее мощное и важное базовое подмножество языка характеризуется наиболее высоким уровнем оптимизируемости. Для этого подмножества возможно в полной мере осуществление всех выбранных в работе видов логической оптимизации за исключением открытой вставки тел XQuery-функций, определяемых пользователем. Реализация этого вида оптимизации возможна только для подмножества XQuery-функций, которое, тем не менее, имеет наибольшее значение для практики. Расширение базового подмножества языка XQuery операциями поддержки XQuery-контекста и операциями, основанными на наличии уникальных идентификаторов XML-узлов, приводит к снижению уровня оптимизируемости запросов,

хотя в некоторых случаях удается произвести оптимизацию путем применения приведенных в работе специальных приемов.

Заключение

В диссертационной работе получены следующие результаты:

1. Разработан язык UQL, позволяющий формулировать запросы к данным в терминах диаграмм классов UML.
2. Разработан и реализован метод поддержки языка UQL через трансляцию UQL-запросов в запросы на языке XQuery.
3. Разработаны, формально обоснованы и реализованы методы логической оптимизации XQuery-запросов.

Литература

[1] CODASYL DBTG Report, April 1971.

[2] Codd E.F. "A Relational Model of Data for Large Shared Data Banks." Comm. of the ACM, 1970, v. 13, no. 6, pp. 377-387.

[3] Chen P.P. "The Entity-Relationship Model. Toward to Unified View of Data." ACM Trans. on Database Syst., v. 1, no. 1, 1976, pp. 9-36.

[4] D. M. Campbell, D. W. Embley, B. D. Czejdo. "A Relationally Complete Query Language for an Entity-Relationship Model." International Conference on Conceptual Modeling (ER), p. 90-97, 1985.

[5] K. Subieta, M. Missala. "Semantics of Query Languages for the Entity-Relationship Model." International Conference on Conceptual Modeling (ER), p. 197-216, 1986.

[6] J.M. Smith, D.C.P. Smith. "Database Abstractions: Aggregation and Generalization." ACM Trans. on Database Syst., v. 2, no. 2, 1977, pp. 105-133.

[7] M. Hammer, D. McLeod. "Database Description with SDM: A semantic database model." ACM Trans, Database Syst., 19, 3, September 1987.

[8] D. Jagannathan et al. "SIM: A Database System Based on the Semantic Data Model." ACM SIGMOD Conf. 1988.

[9] R. Barker. "CASE*Method. Entity-Relationship Modelling." Addison-Wesley Publishing Co., 1990.

[10] D.W. Shipman. "The Functional Data Model and the Data Language DAPLEX." ACM Trans. on Database Systems, Vol. 6, No. 1, March 1981, Pages 140-173.

[11] D.H. Fishman, D. Beech, H.P. Gate et al. "IRIS: An object-oriented database system." ACM Trans. Off. Inf. Syst. 5, 1, 1987.

[12] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. "Lore: A Database Management System for Semistructured Data." SIGMOD Record, 26(3), pp. 54-66, September 1997.

[13] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener. "The Lorel Query Language for Semistructured Data." International Journal on Digital Libraries 1(1), p. 68-88, 1997.

[14] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, D. Suciu. "XML-QL." The W3C Query Languages Workshop, 1998.

[15] J. Robie, J. Lapp, D. Schach. "XML Query Language (XQL)." The W3C Query Languages Workshop, 1998.

[16] "ANSI/X3/SPARC Study on Data Base Management Systems Interim Report." FDT Bulletin, 7 (2), 1975, pp. 1-140.

[17] M.M. Astrahan et al. "System R: Relational Approach to Database Management." ACM TODS 1, No.2, June 1976.

[18] M.W. Blasgen et al. "System R: An Architectural Overview." IBM Sys. J. 20, No.1, February 1981.

[19] M. Jarke, J. Koch. "Query Optimization in Database Systems." ACM Computing Surveys, Vol. 16, No. 2, June 1984.

[20] Y.E. Ioannidis. "Query Optimization." ACM Computing Surveys, Vol. 28, No. 1, 1996

[21] S. Chaudhuri. "An Overview of Query Optimization in Relational Systems." ACM PODS, 1998.

[22] G. Piatetsky-Shapiro, C. Connell. "Accurate Estimation of the Number of Tuples Satisfying a Condition." SIGMOD Conference 1984, pp. 256-276.

[23] W. Kim. "On Optimizing an SQL-like Nested Query." ACM TODS 7, No. 3, September 1982.

[24] U. Dayal. "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers." VLDB Conf., 1987.

[25] G.M. Lohman, K. Ono. "Measuring the Complexity of Join Enumeration in Query Optimization." VLDB Conf., 1990.

[26] A. Rosenthal, C. Galindo-Legaria. "Query Graphs, Implementing Trees, and Freely Reorderable Outerjoins." ACM SIGMOD Conf., 1990.

[27] S. Chaudhuri, K. Shim. "An Overview of Cost-based Optimization of Queries with Aggregates." IEEE DE Bulletin, September 1995.

[28] S.B. Yao. "Optimization of Query Evaluation Algorithm." ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, Pages 133-155.

[29] Terese. "Term Rewriting Systems." Cambridge University Press, 2002.

[30] D. D. Chamberlin, J. Robie, D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources." WebDB (Selected Papers) 2000, pp. 1-25.

[31] "XQuery 1.0: An XML Query Language." W3C Working Draft, 15 November 2002.

[32] "XML Path Language (XPath) 2.0" W3C Working Draft, 15 November 2002.

[33] "XQuery 1.0 and XPath 2.0 Formal Semantics." W3C Working Draft, 15 November 2002.

[34] "XQuery 1.0 and XPath 2.0 Functions and Operators." W3C Working Draft 15, November 2002.

[35] "XQuery 1.0 and XPath 2.0 Data Model." W3C Working Draft, 15 November 2002.

[36] "Extensible Markup Language (XML) 1.0 (Second Edition)." W3C Recommendation, 6 October 2000.

[37] "XML Schema Part 1: Structures." W3C Recommendation, 2 May 2001.

[38] "XML Schema Part 2: Datatypes." W3C Recommendation, 2 May 2001.

[39] H.V. Jagadish, L.V.S. Lakshmanan, D. Srivastava, K. Thompson. "TAX: A Tree Algebra for XML." DBPL, p. 149-164, 2001.

[40] N. J. Nunes, J. F. Cunha. "Towards Flexible Automatic Generation of User-Interfaces via UML and XMI." IDEAS, citeseer.nj.nec.com/nunes02towards.html, 2002.

[41] "Unified Modeling Language (UML)." Version 1.3, OMG Specification, 1999.

[42] M. Fernandez, J. Simeon, P. Wadler. "A semi-monad for semi-structured data." ICDT 2001, p. 263-300.

[43] Крачтен Ф. "Введение в Rational Unified Process." Вильямс, ISBN 5-8459-0239-8, 2002.

[44] P. Fankhauser. "XQuery Formal Semantics: State and Challenges.", SIGMOD Record 30(3): 14-19, 2001.

[45] B. Choi, M. Fernandez, J. Simeon. "The XQuery Formal Semantics: A Foundation for Implementation and Optimization.", www.cis.upenn.edu/ kkchoi/galax.pdf, 2002.

[46] I. Manolescu, D. Florescu, D. Kossmann. "Answering XML Queries on Heterogeneous Data Sources." VLDB 2001, pp. 241-250.

[47] Гринев М.Н. "Системы управления полуструктурированными данными" журнал "Открытые системы" 5-6, издательство "Открытые системы", 1999.

[48] Гринев М.Н. "XML-технологии: унифицированный доступ к разнородным данным" Сетевой журнал Data Communications, 6, 2001.

[49] M. Grinev, S. Kuznetsov. "An Integrated Approach to Semantic-Based Searching by Metadata over the Internet/Intranet" 5th East-European Conference on Advances in Databases and Information Systems (ADBIS), Professional Communications and Reports, Vol. 2, 2001.

[50] M. Grinev, S. Kuznetsov. "Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience." 6th East-European Conference on Advances in Databases and Information Systems (ADBIS), LNCS 2435, 2002, p. 340-345

[51] Гринев М.Н., Кузнецов С.Д. "UQL: язык запросов к интегрированным данным в терминах UML." Программирование, 2002, номер 4, с. 9-19.