Российская академия наук
Сибирское отделение
Институт систем информатики
имени А. П. Ершова

В. В. Кальченко

ОБЗОР АЛГЕБР ДЛЯ XML-БАЗ ДАННЫХ

Препринт
127

Новосибирск 2005

В последние годы все большее распространение получают системы XML-баз данных. Наиболее перспективным языком запросов для них считается XQuery. Предлагаемая работа содержит обзор различных алгебр для XML-баз данных и дается оценка их применимости для поддержки XQuery.

© Институт систем информатики им А. П. Ершова СО РАН, 2005

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Vitaliy V. Kalchenko

REVIEW OF ALGEBRAS FOR XML DATABASE SYSTEMS

Preprint
127

Novosibirsk 2005

XML database systems have become very popular last years. XQuery is regarded to be the most perspective query language for these systems. This paper contains a review of different algebras for XML databases. Estimation of their applicability for XQuery support is given.

© A. P. Ershov Institute of Informatics Systems, 2005

ВВЕДЕНИЕ

В последнее время требования к базам данных возросли, и реляционные базы данных уже не могут решить все задачи без значительного усложнения внутренней структуры данных. В качестве альтернативы рассматриваются XML-базы данных, но до сих пор не существует всеми признанного стандарта для языка запросов, такого как SQL для реляционных баз. Наиболее перспективным в этом смысле языком является XQuery [1]. Разработка алгебры для XQuery — первоочередная задача при создании XML-базы данных. Предлагаемая работа содержит обзор различных алгебр для XML-баз данных и дается оценка их применимости для поддержки XQuery. Некоторые из этих алгебр уже успешно используются в существующих проектах. Один из таких проектов — база данных TIMBER [2] — рассмотрен более подробно, так как на момент написания статьи это был один из самых удачных с коммерческой точки зрения проектов XML-баз данных.

Работа организована следующим образом: в первой части даны основные понятия языка XQuery и структура XML-документов, вторая часть содержит обзор алгебр, разработанных при построении системы TIMBER, в третьей части описаны алгебры, основанные на реляционных алгебрах, в четвертой части рассмотрены алгебры, изначально ориентированные на XML, и пятая часть — заключительная.

Работа выполнена при частичной финансовой поддержке РФФИ, грант № 04-01-00272.

1. ОСНОВНЫЕ ПОНЯТИЯ

XML-документ [20] состоит из *информационных единиц* с определенным *документным порядком* (порядок, в котором они следуют в текстовой версии XML-документа). Информационные единицы хранятся в базе данных в виде *узлов*. XML-документ часто представляют в виде упорядоченного дерева. Корнем дерева является документный узел, каждый элемент может иметь дочерние элементы и атрибуты. В модели данных XQuery [13] каждый узел имеет идентификатор и состояние, данные о котором можно получить с помощью *аксессоров* (например, *node_kind*, *node_name*, *base_uri*, *type_name*, *type_value*, `attributes`, *parent*, `children` и т.д.). Каждый узел принадлежит типу Node, который является объединени-

ем типов Document, Element, Attribute, Text, Namespace, ProcInstruction и Comment. Все узлы линейно упорядочены таким образом, что если какой-то узел одного документа предшествует узлу другого документа, то все узлы первого документа предшествуют данному узлу второго документа. Структура XML-документа часто задается XML-схемой [21]. В схеме определяются типы элементов и атрибутов, отношения “дочки—матери” между элементами, пользовательские типы данных и т.д.

Язык XQuery используется для написания запросов к XML-базам данных. Обычно запрос выглядит следующим образом:

for $\$x_1$ in s_1, $\$x_2$ in $s_2(\$x_1)$, ..., $\$x_m$ in $s_m(\$x_1,\ldots,\$x_{m-1})$
let $\$y_1 := e_1(\$x_1,\ldots,\$x_m)$, $\$y_2 := e_2(\$x_1,\ldots,\$x_m,\$y_1)$,..., $\$y_n := e_n(\$x_1,\ldots,\$x_m,\$y_1,\ldots,\$y_n)$
where $p(\$x_1,\ldots,\$x_m,\$y_1,\ldots,\$y_n)$
order by $e(\$x_1,\ldots,\$x_m,\$y_1,\ldots,\$y_n)$
return $f(\$x_1,\ldots,\$x_m,\$y_1,\ldots,\$y_n)$,

где s_i — последовательности, $\$x_i$, $\$y_i$ — переменные, а p, e и f — выражения. Операция `for` определяет область, откуда берутся значения, `let` позволяет вычислить промежуточные значения, необходимые для остальных операций, `where` задает условие на аргументы, `order` задает порядок полученных в результате вычисления значений и `return` — структуру возвращаемого XML-фрагмента. Такой запрос обычно называют выражением FLWOR. В более ранних версиях XQuery не было операции `order`, и запрос назывался FLWR. В выражениях XQuery широко используются *путевые выражения* [18], которые позволяют задать путь в дереве XML-документа. Путевое выражение обычно содержит в себе последовательность узлов, следующих в порядке вложенности, указывающую на то, как достигнуть последнего узла последовательности, начиная с первого.

При обработке запросов к базе данных выражение на языке запроса обычно транслируется в выражение соответствующей алгебры и затем производится вычисление полученного выражения. Алгебра используется по двум основным причинам. Во-первых, алгебра задает семантику языка запросов. Во-вторых, алгебра обычно поддерживает набор правил оптимизации полученного в результате трансляции выражения. При разработке алгебр для XML-данных возникают три ключевые проблемы: вычисление путевых выражений, вычисление и выполнение вложенных запросов и сохранение документного порядка при вычислениях.

Алгебру для XQuery удобно рассматривать как многоосновную алгебру [22]. Многоосновная сигнатура Σ определяется как пара (T,F), где T — множество основ, а F — множество символов операций, каждому из кото-

рых сопоставлен профиль. Символ операции — это символ или имя, а профиль — это элемент множества T или $t_1,\ldots,t_n \rightarrow t_{n+1}$, где t_i — элементы из T.

Многоосновная алгебра $\mathtt{A}$ сигнатуры Σ содержит в себе:

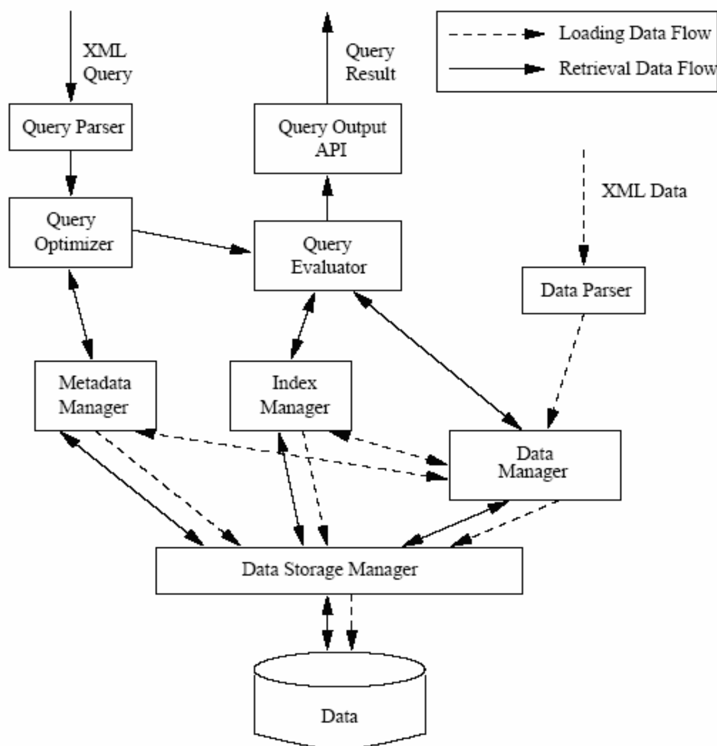
- множество $\mathtt{A}_t$ для каждого элемента $t \in T$,
- элемент $c^{\mathtt{A}} \in \mathtt{A}_t$ для каждой операции c с профилем t,
- функцию $f^{\mathtt{A}} : \mathtt{A}_{t1} \times \ldots \times \mathtt{A}_{tn} \rightarrow \mathtt{A}_{tn+1}$ для каждого символа операции f с профилем $t_1,\ldots,t_n \rightarrow t_{n+1}$.

Набор множеств алгебры $\mathtt{A}$ называется *носителем алгебры* и обозначается $|\mathtt{A}|$. Алгебра сигнатуры Σ называется Σ-алгеброй. Множество T состоит из множества имен типов, а множество F — из операций, определенных на этих типах. В алгебре определены понятия *множество* и *последовательность* (упорядоченное множество или мультимножество). Термин *коллекция* в дальнейшем употребляется там, где уместно использование как множества, так и последовательности.

2. АЛГЕБРЫ, РАЗРАБОТАННЫЕ ДЛЯ СИСТЕМЫ TIMBER

2.1. Общая архитектура базы данных TIMBER

Один из популярных подходов к хранению XML-данных — это их преобразование в реляционные данные. Но при этом, из-за гибкости XML возникает огромное количество таблиц. Авторы системы Timber [2], реализованной в Университете Мичигана, поставили перед собой задачу разработать базу данных, в которой XML-данные хранятся непосредственно в виде деревьев. В то же время было необходимо сохранить все преимущества реляционных баз данных, как, например, обработка множества элементов за одну операцию и др. Система основана на алгебре деревьев TAX [3] и хранит XML-данные в естественном виде (т.е. не преобразуя их в реляционные таблицы, см. разд. 2.2). Для обработки запросов в ней реализованы новые методы доступа и методы оптимизации.



База данных Timber построена на основе Shore (популярной машины баз данных, ответственной за управление диском, буферизацию и параллельное выполнение операций чтения-записи). XML-данные, индексы и метаданные также хранятся в Shore и обрабатываются через интерфейсы, предоставляемые процессорами Data Manager, Index Manager и Metadata Manager (см. рис. выше). Data Parser преобразует XML-документ в дерево, каждый узел которого потом преобразуется процессором Data Manager во внутреннее представление в базе данных. Для извлечения данных Query Evaluator предоставляет набор навигационных и сканирующих интерфейсов, при этом извлекается один узел за раз. Эти интерфейсы также могут использоваться процессорами Index Manager и Metadata Manager. Атрибуты узла (как XML-атрибуты, так и специальные атрибуты, заданные базой данных) хранятся в отдельном узле, присоединенном к исходному в качестве дочки, содержи-

мое элемента также хранится в отдельном узле. Комментарии и команды обработки игнорируются. С каждым узлом связывается пара атрибутов (здесь и далее *атрибут* — это некоторое значение в базе данных, а *XML-атрибут* — атрибут элемента в XML-документе) *start* и *end*, так чтобы этот интервал был точно вложен в интервал предка. Для указания уровня вложенности узла используется атрибут *level*. С помощью атрибутов *start* и *end* определяется связь предок—потомок, с помощью дополнительного атрибута *level* можно определить связь мать—дочь. В качестве индекса узла при этом используется тройка *start*, *end* и *level*.

Запросы на языке XQuery преобразуются в алгебраические операции с помощью Query Parser. Алгебра TAX и ее физическая интерпретация подробно описаны ниже. Query Optimizer оптимизирует полученное дерево операций по набору правил и метаданных (например, схемы) и осуществляет перевод логических операций в физические. Получившийся план вычислений выполняется с помощью Query Evaluator. Конкретные данные извлекаются из базы только тогда, когда уже невозможна работа с одними индексами. Осуществляется это с помощью специальной операции физической алгебры.

2.2. TAX: Алгебра деревьев для XML

При разработке алгебры авторы ставили перед собой целью решение следующих задач: 1) реализация всего многообразия операций с деревьями с помощью достаточно простой алгебры, 2) учет значительного разнообразия схожих объектов (например, объект “книга”, который может иметь разное количество авторов).

Алгебра работает не с множеством узлов дерева, а с множеством деревьев. Это избавляет от ряда сложностей, возникающих при представлении узла в виде фиксированного набора атрибутов. В XML узлы одного типа могут содержать разные атрибуты, поэтому информацию о структуре документа (отношения “предок—потомок” и т.д.) достаточно сложно получить при использовании реляционных таблиц. Но при выборе множества деревьев в качестве модели данных возникают и такие сложности, как неоднородность схожих объектов. Авторы решили эту проблему, введя *шаблоны деревьев* (pattern trees).

Таким образом, TAX оперирует множеством деревьев *C*, где дерево *c*, принадлежащее *C*, построено на основе XML-документа (или фрагмента документа) и удовлетворяет следующим условиям.

1. Каждому XML-элементу соответствует узел дерева, при этом содержимое узла — это XML-атрибуты данного элемента (пары «имя, значение»), а

дети узла — вложенные элементы. TAX не требует, чтобы XML-атрибут содержал лишь единственное значение, хотя этого требует спецификация XML.

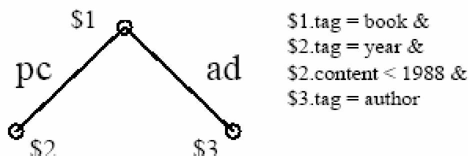
2. Каждый узел содержит дополнительные атрибуты. Атрибут *tag* указывает тип элемента. Узел может содержать атрибут *content*, тип которого может быть одним из простых типов (string, int, …). Предполагается, что каждый узел содержит специальный атрибут *pedigree*, указывающий на происхождение элемента и необходимый для определения того, являются ли два узла копиями одного и того же узла (содержимое этого атрибута зависит от реализации, например, это может быть ID документа плюс смещение в документе).

Как было отмечено выше, база данных представляет собой лес деревьев. Операции TAX используют одно или несколько множеств деревьев из этого леса в качестве аргументов и возвращают множество деревьев в качестве результата.

Шаблон дерева — это пара $P = (T, F)$, где $T = (V, E)$ — дерево, V — множество узлов, E — множество ребер, F — формула, состоящая из предикатов, применяемых к узлам.

- Каждый узел в V помечен числом (*$1*,*$2…*).
- Каждое ребро помечено как *pc* (parent—child) или *ad* (ancestor—descendant).
- Допустимые предикаты для узла *$i*, атрибута *attr* и значения *val*:
  $i.attr θ val, где *θ* — один из =, >, < и т.д.,
  $i._ θ = val, любой из атрибутов содержит такое значение,
  $i.attr = _, элемент содержит атрибут,
  $i.attr = *$j.attr′*; сравнение атрибутов узлов *$i* и *$j*.
- Дополнительно допустимы предикаты следующего вида:
  $i.attr + *$j.attr′* = 0,
  $i.attr = *$j.attr′* *”terrorism”, *attr* и *attr′* – строковые атрибуты,
  * — конкатенация строк, “terrorism” — строковая константа,
  $i.index = *first*, *$i* — первый ребенок родителя,
  $i before $j, узел *$i* находится до узла *$*.

Пример шаблона:



С помощью шаблона дерева можно выделять из заданного множества деревьев подмножество схожих деревьев, удовлетворяющих этому шаблону (это называется *вложением шаблона дерева во множество*). Формально, вложение шаблона дерева $P=(T, F)$ во множество деревьев C — это такое тотальное отображение $h:P\rightarrow C$ узлов из T в узлы из C, что:

1) h сохраняет структуру T, т.е. для ребра (u, v), помеченного в T как *ad* (*pc*), ребро $(h(u), h(v))$ тоже помечено как *ad* (*pc*);

2) полученные в результате отображения деревья удовлетворяют формуле F.

Таким образом, получается множество деревьев, схожих указанным в шаблоне фрагментом.

Для упорядочивания множества деревьев в TAX используется функция TVF (Tree Value Function), которая сопоставляет дереву некоторое значение из упорядоченного множества (например, действительное число).

В TAX определены следующие операции.

1. `Selection`: аналог операции SELECT в реляционной алгебре. Аргументы: C — множество деревьев, P — шаблон дерева, SL — список узлов из P, для которых будут возвращены все потомки (а не только сам узел). Результат: множество деревьев, полученное с помощью вложения шаблона с дальнейшим присоединением потомков узлов из SL.
2. `Projection`: исключение всех лишних узлов. Как мы знаем, в реляционных базах данных SELECT выбирает строки, а проекция выбирает столбцы, таким образом являясь ортогональной к SELECT операцией. В случае XML-данных такой связи между операциями нет. Аргументы: C — множество деревьев, P — шаблон дерева, PL — список узлов, которые будут возвращены. Результат: множество деревьев, полученное с помощью вложения шаблона с дальнейшим удалением узлов $h(v)$, для всех $v \in PL$.
3. `Product`: прямое произведение двух множеств. Аргументы: C, D — множества деревьев. Результат: множество всевозможных деревьев следующего вида: корневой узел — специальный узел с именем “tax_prod_root”, левым потомком которого является корневой узел дерева из C, правым — корневой узел дерева из D.
4. `Join`: соединение деревьев, взятых из разных множеств, по общему атрибуту. Аргументы: $C1$, $C2$ — множества деревьев, $P1$, $P2$ — шаблоны деревьев, $o1$, $o2$ — операции `Selection` или `Projection`, *newTag* — тип корневого узла в результирующих деревьях, формула вида *$i.attr1 = $j.attr2*, где *$i* (*$j*) принадлежит $P1$ ($P2$). Результат:

деревья с корневым узлом типа *newTag*, левым поддеревом *w1* и правым *w2*, где *w1* (*w2*) является результатом применения операции *o1* (*o2*) на множестве *C1* (*C2*) с использованием шаблона *P1* (*P2*) (*h*(*$i*) = *w1*, *h*(*$j*) = *w2*), и *w1.attr1* = *w2.attr2*.

5. `LeftOuterJoin` (`RightOuterJoin`): то же, что и `Join`, но дополнительно содержит деревья без правого потомка (т.е. если для *w1* не нашлось *w2* такого, что *w1.attr1* = *w2.attr2*, то все равно будет построено дерево в возвращаемом множестве так же, как и в операции `Join`, но без правого потомка).
6. `Grouping`: разделение множества деревьев на множества по заданному правилу (например, группировка книг по авторам). Аргументы: *C* — множество деревьев; *P* — шаблон дерева; функция группировки, указывающая на базис группировки (обычно это список узлов из *P* и/или атрибутов этих узлов, на основе значений которых происходит разбиение множества *C*); *VTF* — функция упорядочивания деревьев. Результат: функция группировки разбивает полученное с помощью вложения шаблона *P* множество деревьев *W* на множества *Wi* (так, чтобы в каждом *Wi* содержимое узлов деревьев, т.е. значений атрибутов, упомянутых в списке функции группировки, было одним и тем же). Операция `Grouping` выполняет также функции операции ORDERBY в SQL. Для этого задается пустой базис, и все деревья попадают в одно возвращаемое отсортированное дерево.
7. `Distinct`: удаление элементов, одинаковых по значению.
8. `Aggregation`: получение некоторого значения на основе данного множества значений. Простые функции агрегации — это *max*, *min*, *sum*, *count* и т.д. Аргументы: *C* — множество деревьев; *P* — шаблон дерева; *f* — функция агрегации; *update specification*, параметр, указывающий, куда нужно поместить полученное с помощью *f* значение (оно помещается в какое-либо место деревьев, получившихся после вложения *P* во множество). Результат: множество деревьев, содержащих в себе значение, полученное с помощью функции агрегации.
9. `Renaming`: получение изоморфного множества, в котором у некоторых узлов изменены атрибуты *tag* или переименованы некоторые атрибуты.
10. `Reordering`: перегруппировка элементов на основе заданной *TVF*, *P* (шаблона дерева) и *RL* — списка параметров в *P*, на основе которых производится перегруппировка.

11. `Copy-and-Paste`: реструктуризация деревьев. Аргументы: *C* — множество деревьев, *P* — шаблон дерева, *CL* — список узлов из *P*, которые будут перемещены и *update specification* — выражение вида *AfterLastChild*(*$i*), *Before*(*$i*) и т.д., т.е. то место, куда будет осуществлена вставка. Результат: модифицированное множество.
12. `Value Updates`: задание новых значений атрибутам.
13. `Node deletion`: удаление узлов.
14. `Node Insertion`: вставка новых узлов (задается позиция вставки, как в операции `Copy-and-Paste`).

Все операции TAX независимы, т.е. нельзя выразить какую-либо из них через другие.

Авторы доказывают, что любую реляционную базу данных *D* можно представить в виде множества множеств деревьев (с помощью кодирующей схемы *Rep*) так, что для любого реляционного выражения *Q* существует в TAX аналогичное выражение *Q′*, так что *Q′*(*Rep*(*D*)) = *Rep*(*Q*(*D*)).

В статье также рассматривается перевод в указанные операции FLWR-выражений XQuery следующего вида:

1) выражение не содержит рекурсии и вызовов функций;
2) переменным в `LET` присваиваются только значения, полученные с помощью функций агрегации;
3) все регулярные путевые выражения (path expressions) содержат только константы или пробельные символы и могут использовать символы '/' и '//' (символы, служащие для задания пути в XML-документе, см. [13]).

Стоит заметить, что при указанных выше ограничениях не сохраняется семантика запроса, и результат вычисления может быть другим.

2.3. Физическая алгебра для XML

Алгебру, подобную изложенной выше, принято называть *логической*. Такую алгебру не всегда удобно реализовывать напрямую. Сложная модель данных является причиной того, что даже простые запросы могут потребовать значительных вычислений. Поэтому часто вводят понятие *физической алгебры*, которая отличается от логической более простой моделью данных. Например, в логической реляционной алгебре декартово произведение — базовая операция, а обычное соединение — уже нет. В физической реляционной алгебре обычное соединение — базовая операция, а декартово произведение — специальный случай соединения. Операция сортировки — важный элемент физической алгебры — отсутствует в логической. В реля-

ционной алгебре физическая и логическая алгебры манипулируют одними и теми же объектами, в базах данных XML логическая алгебра работает с деревьями, в то время как физическая — с множеством узлов.

В TIMBER выражение XQuery сначала переводится в выражение логической алгебры, которое, в свою очередь, переводится в выражение физической алгебры [4]. Последнее преобразование осуществляет Query Optimizer.

Фундаментальная операция логической алгебры TAX — вложение шаблона дерева во множество деревьев. Это часто встречающаяся операция, поэтому необходимо минимизировать стоимость ее выполнения. К счастью, многие операции в выражении часто используют одно и то же дерево (или очень похожие), что позволяет оптимизировать процесс обработки выражения. Для этого промежуточные результаты (полученные после вложения шаблона дерева) снабжаются ссылками (node reference), указывающими на шаблон дерева и позицию в нем. Дальнейшие операции могут указывать на отдельные узлы во входной последовательности посредством этих ссылок, обычно уточняя содержание узла или его атрибута. Порядок в последовательностях не обязательно задается сортировкой по какому-либо значению атрибута или элемента, например, вполне можно упорядочить деревья по возрастанию некоторой хэш-функции. Операции, которые не осуществляют сортировку, упорядочивают выходную последовательность так же, как и входную.

В физической алгебре нет преимуществ в использовании операций `rename`, `reorder`, `copy-and-paste`, которые работают с множествами деревьев. В алгебре используется одна операция `Procedural Construction`, которая обрабатывает одно дерево за один раз. Эта операция сделана так, чтобы покрывать почти все возможности операции `CONSTRUCT` в XQuery (последняя может содержать вложенные FLWR операторы, в этом случае она будет преобразована в комбинацию операций физической алгебры).

Набор операций физической алгебры не минимален, например, удаление повторяющихся элементов — частный случай группирования. Авторы не видят причины сохранять физическую алгебру минимальной, если это вредит производительности (ведь некоторые специальные операции могут быть реализованы более эффективно). Главная причина использования физической алгебры — возможность строить более эффективные с точки зрения вычислений алгебраические выражения.

Эффективная реализация нахождений структурных связей (мать-дочь или предок-потомок) между узлами при вложении шаблона дерева во множество

деревьев, определяет общую производительность системы. Экспериментальным путем авторы установили, что выгодно использовать все доступные индексы для независимого поиска как можно большего количества узлов по шаблону дерева. Структурная зависимость между найденными узлами проверяется в следующей фазе, после нахождения всех узлов и проверки предикатов. В данной работе также описан алгоритм Stack-Tree-Anc, осуществляющий вложение шаблона во множество указанным выше способом.

При использовании операции группировки данных по какому-либо признаку (например, XML-атрибуту) из-за специфики XML некоторые XML-элементы могут оказаться в нескольких группах (например, по одному XML-элементу “book” в каждой из групп, содержащей все книги одного автора, если у данной книги несколько авторов). Возможный вариант — копировать узел для каждой из групп, но это слишком громоздко. Центральная идея, реализованная в Timber, состоит в следующем. При вложении шаблона дерева во множество деревьев создаются всевозможные кортежи группируемых переменных. Если за группировкой следует функция агрегации, то копирования элементов можно избежать. Например, если необходимо посчитать количество книг у автора, то это можно сделать без физического копирования.

Для шаблона дерева, состоящего из нескольких узлов, существует множество планов вычисления. Query Optimizer должен перебрать все из них (или самые выгодные) и выбрать тот, который наиболее оптимален для данного запроса. Выбор порядка нахождения структурных зависимостей наиболее важен для оптимизации, так как это основная операция при вычислении, существенно влияющая на производительность. В реляционной базе данных почти всегда выгодно сначала обработать `Select`, но в данном случае это не так. Структурные зависимости могут быть обработаны с помощью одних индексов, без доступа к реальным данным. Следовательно, при оптимизации порядка соединений должна быть предусмотрена возможность выполнения операций `Select` в порядке, отличном от порядка, в котором они обрабатываются первыми. Алгоритм, предложенный авторами, состоит в следующем. Они определяют дерево *Status*, которое равно в начале исходному шаблону дерева, с присоединенными узлами для каждой операции `Select`. Операция `move`, которая представляет собой операцию определения структурной связи, сращивает два узла этого дерева, удаляя ребро между ними и добавляя к атрибуту “cost” стоимость, которая вычисляется в зависимости от мощности узла и размера полученного соединения. Завершается алгоритм, когда в *Status* остается один узел. Для уменьшения затрат, авторы доказывают теорему, что для любого узла можно со-

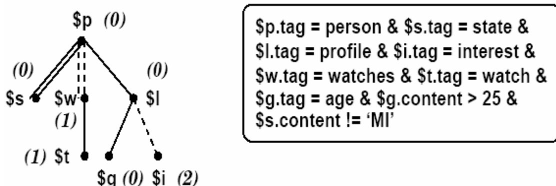
ставить план вычислений, чтобы результат был отсортирован по этому узлу и дополнительной сортировки не будет требоваться.

При обновлении базы обновлять метки *start* и *end* иногда довольно проблематично. В Timber используются вещественные значения для этих меток. Только при большом поступлении новых данных происходит полная перенумерация узлов, что является довольно ресурсоемкой операцией. Изначально данные в базе хранятся в порядке возрастания *start*. При этом используется эвристический алгоритм размещения данных, схожий с алгоритмом, используемым в реляционных базах данных.

2.4. Алгебра с обобщенными шаблонами деревьев

В работе [5] мы видим дальнейшее развитие алгебры TAX. Авторы пишут о том, что гораздо более эффективно можно обрабатывать выражения XQuery, используя *обобщенные шаблоны дерева* (generalized pattern tree, GPT) вместо обычных шаблонов. Для одного выражения XQuery строится один GTP, что позволяет сократить объем вычислений по нахождению деревьев, соответствующих шаблону. Используя эту идею, авторы разработали эффективные планы обработки запросов на языке XQuery.

GTP представляет собой некую абстракцию того, что нужно сделать для вычисления выражения, и дает понять, как сделать эту работу, используя минимальное количество проходов по входным данным. GTP содержит сплошные и прерывистые ребра. Сплошные ребра представляют собой отношения *parent-child*, *ancestor-descendant*, как и в обычном шаблоне дерева. Прерывистые ребра обозначают связи, которые не обязательно должны присутствовать в исходном дереве. При этом обработка дерева происходит следующим образом: сначала находятся все узлы, связанные сплошными ребрами, а затем находятся узлы, связанные прерывистыми ребрами, если они существуют. Узлы в GTP, в отличие от шаблона дерева, имеют не только номер, но и отдельное имя. При этом номер указывает на номер группы. Группа узлов — это максимальное множество узлов, соединенных сплошными ребрами. Пример:



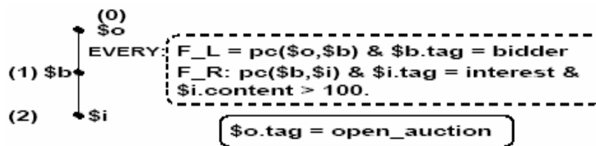
В результате вложения GTP во множество деревьев мы получим множество деревьев, которые частично или полностью содержат в себе узлы, указанные в GTP. Но в формуле F (см. определение шаблона дерева) могут содержаться выражения, использующие те узлы, что отсутствуют в полученном дереве (отмечены в GPT как факультативные). Для решения этой проблемы авторы ввели третье булевское значение, обозначенное $\perp$, со следующими аксиомами: $(\perp\wedge\perp) = (\neg\perp) = (\perp\vee\perp) = \perp$; $(\perp\wedge 1) = (\perp\vee 1) = 1$; $(\perp\wedge 0) = (\perp\vee 0) = 0$. Таким образом, если выражение использует узел, не присутствующий в дереве, то его значение — $\perp$. Для возможности использования квантификаторов XQuery вводится понятие универсального GTP. Универсальное GTP может содержать сплошные ребра, помеченные идентификатором EVERY и удовлетворяющие следующим условиям.

1. Дочерний узел, соединенный с матерью ребром, помеченным как EVERY, достижим из корневого узла дерева по сплошным ребрам.
2. GTP содержит пару формул F_R и F_L для ребра, помеченного как EVERY, которые являются комбинацией предикатов, применяемых к узлам.
3. Узлы, упоминаемые в F_L, должны быть в разных группах.

   Например, для следующего запроса:

   ```
   for $o in document("auction.xml")//open_auction
   where every $b in $o/bidder satisfies $b/increase >100
   return <result>{$o}</result>
   ```

   будет построено GTP, изображенное на рисунке ниже.



При этом должно выполнятся условие $\forall \$b\colon (F_L \rightarrow \exists \$i : F_R)$.

При вложенных запросах генерируется несколько деревьев (по одному для каждого запроса) со следующей нумерацией групп: самый внешний запрос генерирует группы (0), (1), (2)…, следующий по вложенности — группы (x,y), например (1,0), (1,1), следующее — (x,y,z) и т.д. При этом x указывает на номер группы в первом дереве (т.е. группа (x,y) будет дочерней для группы (x), а группа (x,y,z) — дочерней для (x,y)). Проверка на соответствие дерева дочерним группам производится только после установленного соответствия родительской группе (т.е. группа (1,0) проверяется,

только если найдено соответствие для группы (1)). Более подробно с нумерацией можно ознакомиться в рассматриваемой работе, где авторы приводят подробный пример.

Рассмотрев все идеи, приведенные здесь, мы получим алгоритм для преобразования большинства выражений XQuery без функций в GTP, но в данной работе авторы приводят, для простоты, только алгоритм преобразования следующего FLWR-блока:

FLWR ::= *ForClause LetClause WhereClause ReturnClause*.

ForClause ::= `for` $\$x_{v1}$ `in` E_1, ..., $\$x_{vn}$ `in` E_n.

LetClause ::= `let` $\$y_{v1}$:= E_1, ..., $\$y_{vn}$:= E_n.

WhereClause ::= `where` $p(E_1, \ldots, E_n)$.

ReturnClause ::= `return` $\{E_1\}\ldots\{E_n\}$.

E_i ::= *FLWR* | *XPATH*.

p — выражение, *XPATH* — путевое выражение.

Главная мотивация использования GTP — это простота построения эффективной реализации. Это можно сделать, избежав повторяющихся проверок с помощью сходных шаблонов деревьев и откладыванием материализации узлов, пока это возможно.

Далее авторы описывают физическую алгебру для GTP со следующими операциями (*s* — последовательность деревьев):

1. `Index Scan`: $IS_p(s)$: ищет в *s* узлы, удовлетворяющие предикату *p*, использующему некоторый индекс.
2. `Filter`: $F_p(s)$: производит последовательность деревьев, удовлетворяющих в *s* предикату-фильтру *p*.
3. `Sort`: $S_b(s)$: сортирует последовательность деревьев *s*, используя сортировочный базис *b* (имя атрибута или элемента, на основе которого необходимо провести сортировку).
4. `Value Join`: $J_p(s_1,s_2)$: соединяет две последовательности s_1 и s_2 на основе значения какого-либо поля, используя предикат *p*, в котором есть вложенные циклы или сортирующее соединение (`sort-merge join`). Порядок получившейся последовательности совпадает с порядком s_1.
5. `Structural Join`: $SJ_r(s_1,s_2)$: соединяет последовательности s_1 и s_2, используя структурную зависимость *r* между парами (*ac* или *pc*).
6. `Group By`: $G_b(s)$: осуществляет группировку деревьев из *s* по базису *b*. Исходная последовательность должна быть отсортирована на основе базиса группировки.

7. `Merge`: $M(s_1,\ldots,s_n)$: осуществляет группировку деревьев из последовательностей $s_1,\ldots,s_n$. Предполагается одинаковое количество деревьев в каждой последовательности.

В работе также описывается алгоритм преобразования GTP в физический план вычислений.

При выполнении запроса довольно сложно эффективно обработать XML-документ из-за его факультативных и повторяющихся элементов и нерегулярной структуры. При отсутствии схемы документа приходится учитывать всевозможные варианты для каждого элемента. Если известна структура документа (схема или DTD), то можно обрабатывать его более эффективно, упростив при этом GTP на логическом уровне путем удаления узлов с одной стороны и удаления дополнительных операторов (сортировка, удаление дубликатов…) в сгенерированном физическом плане вычислений с другой стороны. Логическая оптимизация достигается за счет удаления невозвращаемых узлов из GTP; например, последовательность $a \rightarrow b \rightarrow c$ преобразуется в последовательность $a \rightarrow c$, если b не содержит других потомков и не участвует в вычислениях предикатов. Логическая оптимизация также достигается за счет удаления узлов с одинаковыми именами, если в схеме четко указано, что у данного родителя не может быть больше одного потомка с таким именем. Физическая оптимизация достигается за счет удаления ненужных операций “сортировка”, “группировка” и “удаление одинаковых элементов”. Подробные примеры описаны в работе. Также приводится алгоритм упрощения GTP на основе данных о связях узлов из схемы. После проведенных экспериментов авторы выяснили, что вычисления с помощью GTP во всех тестах показывают лучшие результаты, иногда превосходящие результаты полученные с помощью других алгебр на несколько порядков. GTP с оптимизацией с помощью схемы в некоторых частных случаях тоже дает большой прирост производительности, в остальных же не очень отличается от обычного алгоритма.

2.5. Алгебра с логическими классами

Гибкость структуры XML-документов усложняет разработку алгебр, которые обычно оперируют множеством схожих элементов. Даже если XML-документ согласован со схемой, все равно допускается множество вариаций одного и того же элемента. Часто эта проблема решается путем преобразования запроса в шаблон дерева. По шаблонам, которые задают структуру деревьев, находятся однообразные деревья, с которыми в дальнейшем ведется работа. Но при таком подходе имеется множество недос-

татков, в результате которых замедляется обработка запроса. Семантика XQuery часто требует, чтобы узлы разбивались на группы на основе структурной зависимости. Это влечет за собой потерю оригинальной структуры XML-документа, которую неплохо было бы сохранить. В запросах на языке XQuery довольно часто возникает ситуация, когда значение какого-либо атрибута извлекается из базы более одного раза, причем этого невозможно избежать, если мы хотим работать с однородными объектами. При нахождении структурного соответствия мы “забываем” о уже раннее найденных зависимостях, что тоже не очень хорошо сказывается на скорости работы. В общем, повторяющийся процесс нахождения соответствий шаблонам деревьев разбивает исходные деревья на маленькие части, которые потом обрабатываются операциями группировки и проекции. В результате многие операции работают с очень похожими шаблонами дерева, и количество таких мелких операций сильно замедляет работу в целом.

В работе [6] авторы описывают алгебру TLC, которая позволяет работать с неоднородными наборами деревьев. Для этого они расширили понятие ребра, позволяя таким образом получать разные деревья в результате вложения шаблона во множество деревьев (а не однородные, как при использовании обычных шаблонов). Для любого ребра $e = (u,v)$ задается спецификация согласования $mSpec_e$, которая может содержать следующие значения:

“-”: дочкой узла *u* может быть один и только один узел *v*;

“?”: дочкой узла *u* может быть один или ни одного узла;

“+”: дочками узла *u* могут быть один и более узлов *v*;

“*”: дочками узла *u* может быть любое количество узлов *v*.

Таким образом, мы получаем понятие *аннотированного шаблона дерева* (Annotated Pattern Tree, APT) как шаблона дерева, для каждого ребра которого задана спецификация согласования, а для каждого узла *v* — предикат P_v (см. определение шаблона дерева). При таком подходе возникает проблема, когда у нас имеется несколько узлов с одним именем и неизвестно, с каким из них работать в предикате. Для ее решения авторы вводят понятие *логического класса*. Для дерева $h(Q)$ (h — функция вложения шаблона во множество деревьев, описанная в разд. 2.2), удовлетворяющего шаблону Q, логический класс *LC* для узла *v* определяется следующим образом: $LC(v) = \{h(v)\colon h(v) \in V_{h(Q)}\}$. Следом вводится понятие *сужения логического класса LCR*, определяемого как $LCR(h(Q)) = (V,E)$, такое, что $V = \{h(v)|v$ — узел в $Q\}$, $(h(u),h(v))$ — ребро, если (u,v) — ребро в Q. Сужение логического класса изоморфно *Q*. *Метка логического класса* (*LCL*) — это номер, уникальный в пределах одного дерева, присваиваемый каждому логическому

классу. Таким образом, для разных по структуре деревьев сужения будут однообразными, и уже с ними могут работать операции алгебры.

Каждая операция алгебры работает с одним или несколькими множествами деревьев и возвращает множество деревьев. Так как логический класс содержит в себе несколько узлов, операции работают с множествами узлов. Некоторые операции могут работать с произвольным количеством узлов, другие же — только с одним.

1. `Filter` F[LCL_f, p, m](s). Аргументы: множество деревьев *s*, предикат *p*, состояние *m*, описанное ниже, и метка LCL_f. Результат: те деревья из *s*, у которых LC_f (логический класс для узла *f*) удовлетворяет предикату *p*. Параметр *m* указывает, как работать с набором узлов LC_f. По умолчанию *m* — это квантор всеобщности (Every(E)). Могут употребляться также квантор существования *ALO* (*At Least One*) и квантор *EX* — один и только один. Возможны и другие варианты — например, первый элемент и т.д. (на основе порядка узлов в логическом классе).
2. `Join` J[apt,p](s_l,s_r). Аргументы: два множества деревьев s_l и s_r, аннотированный шаблон дерева *apt* и предикат *p*. Параметр *p* задает необходимые связи значений между узлами, используя метки *LCL* для описания узлов из множеств. Для каждого *LC*, упоминающегося в *p*, необходимо наличие только одного элемента. Параметр *apt* задает структуру возвращаемого дерева, используя в качестве корня специальный узел. Левая часть должна быть соединена ребром “-”, для правой — ограничений нет. Результат: дерево со структурой, заданной в *apt*, являющееся соединением деревьев из s_l и s_r.
3. `Select` S[apt](s). Аргументы: множество деревьев *s* и аннотированный шаблон дерева *apt*. Результат: *LCR* для деревьев из *s*, удовлетворяющих *apt*.
4. `Project` P[nl](s). Аргумент: *nl* — список меток, указывающих на логические классы, которые должны быть сохранены в возвращаемом множестве деревьев. Результат: множество деревьев, содержащих только те логические классы, которые указаны в *nl*.
5. `Duplicate-Elimination` DE[nl,ci](s). Аргумент: *nl* — список логических классов, на основе которого происходит удаление повторяющихся деревьев. Все логические классы, упоминающиеся в *nl*, должны содержать один элемент. Параметр *ci* указывает, на базе чего выполнять удаление: на базе содержимого узла или его номера. Результат: множество деревьев.
6. `Aggregate-Function` AF[fname,LCL_a,newLCL](s). Аргументы: *fname* — имя функции (count, max…), метка *LCLa*, указываю-

щая на узлы, на которых будет выполнятся функция, и метка *newLCL*, указывающая на новый узел, содержащий результат.

7. `Construct` C[c](s). Аргумент: *c* — аннотированный шаблон дерева, по которому создаются возвращаемые деревья.

Далее авторы приводят алгоритм, по которому упрощенное FLWOR-выражение языка XQuery преобразуется в выражение алгебры.

Многие операции используют почти одинаковые шаблоны деревьев, поэтому результаты вложения шаблонов с метками сохраняются между операциями и могут быть использованы повторно. Часто одни и те же узлы встречаются в выражении с разными предикатами и аннотированными ребрами, что приводит к многократному извлечению значений атрибутов из базы данных. Чтобы этого избежать, авторы вводят операцию `Flatten`: FL[LCL_p,LCL_c](s) для множества деревьев *s*, в которой класс *P* должен содержать один узел, а класс *C* должен быть дочкой *P*. В результате для каждого $c \in C$ образуется новое дерево, которое содержит только один узел *c* у родителя *P*. В работе приведен пример, как за счет этой операции исключается повторное обращение к базе данных. Иногда требуется оператор, обратный `Flatten`, но такой оператор не может быть введен из-за того, что может быть необходимо включить все дочерние узлы, а не только те, что удовлетворят предикату. Для разрешения такой ситуации авторы вводят понятие *затененных узлов* (shadowed nodes) и две новых операции `Shadow` и `Illuminate`. В операции `Shadow` SH[P,C](s) класс *P* должен содержать один узел, а класс *C* должен быть дочкой *P*. В результате для каждого $c \in C$ образуется новое дерево, которое содержит только один узел *c* у родителя *P*, а остальные узлы из *C* помечены как *shadowed* (такие узлы не доступны для операций). В результате операции `Illuminate` IL[LCL_i](s) все классы, на которые указывает LCL_i, становятся активными (не *shadowed*).

В реализации авторы вводят дополнительные физические операции:

`Nest-Structural-Join` NSJ[LC_l,r,LC_r](s_l,s_r): s_l, s_r — множества деревьев, *r* — предикат, указывающий на структурную связь между LC_l и LC_r. Параметр LC_l должен содержать один узел, а LC_r должен быть корнем дерева. В результате получается следующее множество: для каждого $T_l \in s_l$ и всех $T_r m \ldots T_r n \in s_r$, для которых выполняется [$r(LC_l, LC_r\ I)$ = True, $i \in m..n$], генерируется дерево, соединяющее T_l и $T_r m \ldots T_r n$, как это указано в *r* (*ac* или *pc*). Таким образом, эта операция работает как обычная операция `Join` с включенной в нее `GroupBy`.

Операция `Nest-Value-Join` NVJ[LC_l,p,LC_r](s_l,s_r) определяется аналогично.

Далее в статье приводятся результаты сравнительного тестирования TLC, GTP, TAX и алгоритма, реализующего рекурсивный обход деревьев

(NAV). В результате авторы выяснили, что TLC опережает GTP во всех тестах на 30%-300%. В одном из тестов (запрос со вложенным в LET запросом, с 12-ю возвращаемыми аргументами и большим количеством возвращаемых деревьев) GTP не справился за 10 минут, в то время как NAV (самый емкий алгоритм) справился за 89,24 секунды, а TLC — за 74,3 секунды. Оптимизация за счет операций `Flatted`, `Shadow` и `Illuminate` иногда позволяет в два раза ускорить скорость вычисления с помощью TLC.

3. АЛГЕБРЫ, ОСНОВАННЫЕ НА РЕЛЯЦИОННЫХ АЛГЕБРАХ

3.1. Алгебра Xstasy

В работе Carlo Sartiani и Antonio Albano [9] рассматривается алгебра, которая является основой базы данных Xtasy [10], и определена как расширение объектно-ориентированных и слабоструктурированных алгебр [11,12].

Алгебра Xstasy использует модель данных, схожую с W3C XML Query Data Model [13]. XML-документ представлен в виде неупорядоченного леса деревьев с помеченными узлами. Порядок при этом задается с помощью специальной функции *pos*. Каждый узел имеет уникальный идентификатор — *oid*, который может быть получен с помощью функции `oid`. Алгебра Xstasy построена на основе алгебры, описанной в статье [12], и заимствует оттуда идею хранения промежуточных данных в структурах, схожих со структурами в реляционных алгебрах. Благодаря этому в ней присутствуют *граничные* операции, которые являются промежуточным звеном между физическими XML-данными и остальными операциями. В алгебре определены две *граничные* операции — `path` и `return`, которые соответственно преобразуют XML-данные в структуры внутреннего формата и наоборот. В дополнение к этим операциям в алгебре определены все стандартные операции: `Selection`, `Projection`, `TupJoin`, `Join`, `DJoin`, `Map`, `Sort`, `TupSort` и `GroupBy`.

Алгебраические операции алгебры Xstasy работают со структурами *Env*, являющимися множествами кортежей, каждый из которых описывает множество связей между переменными. Работа с множествами кортежей вместо деревьев позволяет использовать алгоритмы оптимизации, определенные в реляционных алгебрах. Порядок кортежей важен только в операции

сортировки, все остальные операции работают с несортированными структурами *Enu*. Пример:

$b : o1	$a : o3
$b : o1	$a : o4
$b : o1	$a : o5
...	…

В данном случае, структура представляет собой набор кортежей, каждый из которых содержит два поля. Первое поле ($b : …) содержит *oid* элемента “book”, а второе — *oid* элемента “author”, являющегося дочкой первого элемента. Для книги с тремя авторами структура содержит три кортежа. В общем случае структура *Enu* определена следующим образом:

$Enu\ \ [\ tuple[label_1[t_{11}], \ldots, labe_{ln}[t_{1n}]],$
$\quad \ldots,$
$\quad tuple[label_1[t_{k1}], \ldots, labeln[t_{kn}]]];$

где $label_i$ — имя переменной, t_{ji} — соответствующее значение.

Операция `path` использует аргумент *input filter* для указания на данные, над которыми необходимо работать. *Input filter* определен следующим образом:

$F ::= F_1, \ldots, F_n$ — конъюнктивный фильтр
$\quad | F_1 \vee \ldots \vee F_n$ — дизъюнктивный фильтр
$\quad | (op, var, binder)\ label\ [F]$ — простой фильтр
$\quad | \varnothing$ — пустой фильтр,

где $op \in \{/,//,_\}$ (символ ‘/’ означает pc связь, ‘//’ — ac связь)
$\quad var \in \text{String} \cup \{_\}$
$\quad binder \in \{_, \text{in}, =\}$

При обработке простого фильтра выполняются четыре шага: а) применяется навигационный оператор *op*, б) выбираются элементы или атрибуты, имеющие метку *label*, в) переменная *var* связывается указанным в *binder* способом, г) продолжается вычисление с использованием вложенного фильтра *F*. Таким образом, в качестве аргумента операция *path* использует XML-дерево и возвращает структуру *Enu*.

Операция `return` использует в качестве аргументов структуру *Enu* и *output filter* и возвращает новое XML-дерево. *Output filter* определен следующим образом:

$OF ::= OF_1, . . . , OF_n$
$\quad | \; label[OF]$
$\quad | \; @label[val]$
$\quad | \; val$
$val ::= v_B \mid var \mid vvar$

Этот фильтр может быть как конструктором элемента (*label*[*OF*]), в результате выполнения которого мы получим элемент с именем *label* и содержимым, заданным фильтром *OF*, так и конструктором атрибута (@*label*[*val*] — атрибут с именем *label* и значением *val*), или комбинацией фильтров. Информация, содержащаяся в структуре *Enu*, может быть сохранена двумя способами: копированием узла (*vvar*) или копированием ссылки (*var*). Для элементов, которые были скопированы, создаются новые *oid*.

Рассмотрим остальные операции алгебры.

1. `Selection`. Аргументы: структура *e*, предикат *P*. Результат: новая структура *Enu*, в которой отсутствуют кортежи, не удовлетворяющие условию *P*.
2. `TupJoin`. Аргументы: две независимые структуры e_1 и e_2 и предикат *P*. Предикат *P* вычисляется для каждой из пар кортежей $(t_1,t_2) \in e_1 \times e_2$. Результат: все пары, удовлетворяющие условию *P*.
3. `DJoin`. Аргументы: две структуры e_1 и e_2 (они могут зависеть друг от друга, в отличие от `TupJoin`) и предикат *P*. Результат: пары кортежей, такие же, как и в `TupJoin`.
4. `Sort`. Аргументы: структура *e* и сортирующий предикат *P*, имеющий вид $tuple \times tuple \rightarrow boolean$. Результат: отсортированная структура *Enu*.
5. `GroupBy`. Аргументы: *e* — структура *Enu*, $A \subseteq Att(e)$ ($Att(e)$ — множество меток в *e*), $y \notin Att(e)$, *f* и f_1 — функции, θ — знак операции. Результат: пары, состоящие из элементов *A* и множеств $G = f(\{x \mid x \in e, f_1(x) \, \theta \, f_1(y)\}$.

При оптимизации выражений алгебры используются три класса алгебраических эквивалентностей. Первый класс — классические эквивалентности, унаследованные от реляционных и объектно-ориентированных алгебр, такие, как коммутативность операций `Join` и др. Второй класс — правила разложения путевых выражений, которые позволяют разбивать сложные фильтры на несколько более простых. Третий класс — правила для преобразования вложенных запросов в обычные, без вложений. В работе авторы рассматривают некоторые из этих правил на примерах.

К сожалению, авторы не приводят никаких данных относительно скорости вычислений запросов при использовании алгебры Xtasy. При сравнении

с другими алгебрами (в частности, с TAX), приводятся лишь их некоторые недостатки, которые уже были перечислены в обзорах этих алгебр. Большим недостатком данной алгебры является то, что лишь очень небольшая часть запросов XQuery может быть переведена в выражения алгебры.

3.2. Алгебра XAT

В работе [16] описывается алгебра (XML Algebra Tree, или XAT) для XQuery, которая является ядром базы данных Rainbow. XAT разрабатывалась для обработки как XML-данных, так и реляционных данных.

Модель данных XAT базируется на так называемых XAT-таблицах. Элементом таблицы может быть:

а) атомарное значение,

б) узел (XML-элемент, XML-атрибут, XML-документ),

в) множество, состоящее из узлов и атомарных значений,

г) последовательность (упорядоченное множество).

Имя каждого столбца XAT-таблицы может быть как связанной переменной из запроса пользователя (например, *$v*), так и специально созданной для него переменной col_n. Каждый столбец col_n содержит значения одного из типов, определенных в XML Query Algebra [14]. Таким образом, XAT-таблица — это расширение реляционной таблицы путем добавления типов XML-данных и возможности использования коллекций в качестве элементов таблицы.

Коллекция обладает следующими свойствами:

а) одноэлементная коллекция может рассматриваться как входящий в него элемент и наоборот,

б) коллекция не может быть вложена в другую коллекцию,

в) коллекция может содержать разнотипные данные, тип элемента коллекции при этом определяется как супертип всех элементов.

Примеры XAT-таблиц:

title	price
TCP/IP Illustrated	69.95
Data on the Web	39.95

prices
{<price>65.95</price>,<price>69.96</price>}
{<price>34.95</price>,<price>39.95</price>}

Для обозначения коллекций используется запись {...,...}.

Элементы XAT-таблицы сравниваются по значениям. В случае множеств и последовательностей элементы сравниваются по стандартным математическим правилам.

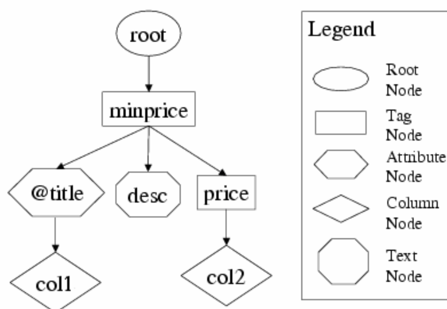
У алгебры XAT два основных предназначения. Первое — это явное задание семантики XQuery. Второе — использование алгебры в качестве базы для системы Rainbow. Алгебра может использоваться для доступа как к XML, так и к реляционным данным. Каждая операция алгебры использует в качестве аргументов одну или несколько XAT-таблиц и возвращает XAT-таблицу (в дальнейшем просто — таблица). Существует три типа операций: XML-операции, SQL-операции и специальные операции. Операции могут работать в двух режимах — чувствительном к порядку или нет.

Операция `Tagger`, работающая в качестве конструктора узлов, использует шаблоны следующего вида:

Pattern ::= *RootNode*, *PatternTree*;
PatternTree ::= (*TagNode* | *AttributeNode*), *PatternTree*;
PatternTree ::= *PatternTree*, *PaternTree*;
PatternTree ::= (*ColumnNode*, *TextNode*).

- *RootNode*: корень шаблона.
- *TagNode*: тэг элемента, содержит имя элемента.
- *AttributeNode*: тэг атрибута, содержит имя атрибута.
- *ColumnNode*: имя колонки, из которой берется значение для атрибута или элемента.
- TextNode: константа.

Пример:
шаблон <minprise@title = [*col1*]>desc<price>[*col2*]</price></minprice> проиллюстрирован ниже.



XML-операции

1. `Expose` $\varepsilon_{col}(s)$. Аргументы: *s* — таблица, *col* — имя колонки. Результат: XML-данные указанной колонки в текстовом формате (т.е. возвращает не таблицу, а строку).
2. `Tagger` $T_p^{col}(s)$. Аргументы: *s* — таблица, *col* — имя колонки, *p* — шаблон. Результат: новая колонка в таблице *s*, в которой содержаться узлы, созданные на основе шаблона *p* и кортежей из входной таблицы.
3. `Navigate` $\phi(\Phi)^{col'}_{col,path}(s)$. Существует два варианта операции — это `navigate unnest` (ϕ) и `navigate collection` (Φ). Первая из них разбивает отношения дочки-матери, создавая для каждой дочки копию матери, вторая же работает с последовательностью дочерних элементов и одной матерью. Аргументы: *col* — имя колонки, *path* — путевое выражение, s — таблица. Результат: новая колонка *col′*, построенная следующим образом: для каждого кортежа в *s* берется значение колонки *col*, и затем по пути *path* в этом значении извлекаются дочерние элементы. Новые элементы помещаются в колонку *col′*. Если в значениях колонки *col* не существует пути *path*, то в *col′* будет пустое множество.
4. `Aggregate` $Agg_{col}(s)$. Аргументы: *s* — таблица, *col* — имя колонки. Результат: сгруппированные в одну последовательность значения колонки *col*, с сохранением дубликатов. Для всех остальных колонок также группируются значения, но с удалением дубликатов. Таким образом, в результате получается таблица с одной строкой.
5. `Composer` C(s). Аргументы: *s* — таблица. Результат: XML-документ, созданный по таблице специального вида. Таблица состоит из следующих колонок: *id*[1..*n*],*type*,*att*[1..*m*],*value*. Колонки *id* определяют идентификаторы элементов, при этом номер колонки *id* указывает на вложенность элемента. Порядок сточек в таблице определяет порядок элементов в документе. Пример таблицы и соответствующего документа:

id1	id2	id3	type	value
1	null	null	prices	null
1	2	null	book	null
1	2	3	title	TCP/IP Illustrated
1	2	4	source	www.amazon.com
1	2	5	price	65.95
1	6	null	book	null
1	6	7	title	TCP/IP Illustrated
1	6	8	source	www.bn.com
1	6	9	price	69.95
1	10	null	book	null
1	10	11	title	Data on the Web
1	10	12	source	www.amazon.com
1	10	13	price	34.95
1	14	null	book	null
1	14	15	title	Data on the Web
1	14	16	source	www.bn.com
1	14	17	price	39.95

```
<prices>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.amazon.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.bn.com</source>
    <price>69.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.amazon.com</source>
    <price>34.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.bn.com</source>
    <price>39.95</price>
  </book>
</prices>
```

6. Операции над множествами: `Union`($\cup^{col}_{col1,col2}(s)$), `Intersect`($\cap$), `Difference` ($-$). Работают с множествами значений из колонок одной строки. Результат помещается в новую колонку и содержит объединение/пересечение/разность множеств.

Специальные операции XAT

1. Оператор SQL: $SQL_{stmt}^{col[1..n]}$. Аргументы: *stmt* — запрос SQL, *col*[*1..n*] — имена колонок. Результат: таблица, построенная на основе SQL-запроса *stmt*. Результаты запроса помещаются в колонки $col_1, .., col_n$.
2. Применение функции: $F^{col}_{param[1..n]}(s?)$. Аргументы: *s* — таблица (необязательный параметр), *col* — имя колонки, *F* — функция, *param*[*1..n*] — аргументы функции. Результат: новая колонка с результатами вычисления функции *F* на элементах входной таблицы или новая таблица, если входная таблица не задана. Примеры функций: *max*, *min*, *sum*, *date*, *time*, *concat*, *contains*, *uppercase*...
3. Создание таблицы: $S_{desc}^{col[1..n]}$. Аргументы: *desc* — строка, *col*[*1..n*] — имена колонок. Результат: таблица, построенная на основе аргумента *desc*, который может быть именем XML-документа, url-ссылкой, описанием таблицы в базе данных и т.д. Если в *desc* задана реляционная таблица, результирующая таблица будет ее копией; если задан XML-документ, то документный узел будет помещен в одну клетку таблицы.
4. Смена имени: $\rho_{col1,col2}(s)$. Аргументы: *s* — таблица, *col1*, *col2* — имена колонок. Результат: таблица, в которой колонка *col1* переименована в колонку *col2*.
5. Итерация: $FOR_{col}(s,sq)$. Используется для представления операции FOR в XQuery. Аргументы: *s* — таблица, *col* — имя колонки, *sq* — дерево операций. Результат: для каждого из значений колонки *col* выполняется *sq* (дерево операций).
6. Условие: $IF_c(sq1,sq2)$. Аргументы: *c* — выражение типа Boolean, *sq1* и *sq2* — деревья операций. Результат: *sq1*, если *c* = true, иначе *sq2*.
7. Объединение: M(s[1..n]). Аргументы: *s*[*1..n*] — таблицы. Результат: объединение нескольких таблиц с одинаковым количеством строк путем конкатенации колонок. Все таблицы должны быть упорядочены.

В алгебре определены стандартные SQL-операции: `Project`, `Select`, `Cartesian Product`, `Join`, `Distinct`, `GroupBy`, `OrderBy`, `Union`, `Difference`, `Intersaction` и `Outer Union`.

Обработка запроса XQuery происходит следующим образом.

1. Анализ. На этом шаге запрос XQuery проверяется внешним синтаксическим анализатором на лексические и синтаксические ошибки. После этого создается дерево операций XAT.
2. Нормализация. Этот шаг используется для нормализации операций дерева запроса. Например, применяются правила трансформации для таких операций, как `Navigator`, `Tagger`, `Select`.
3. Семантический анализ. На этом шаге удаляются выражения, которые не влияют на ход вычислений, например *if a > 4* and *a < 2*.
4. Упрощение. Задачи этого шага: найти избыточные ограничения, удалить простые подвыражения и преобразовать запрос в более простой семантический эквивалент.

В работе авторы подробно рассматривают два основных преобразования при получении дерева операций XAT из запроса XQuery — это преобразование путевых выражений и преобразования FLWR-циклов. К сожалению, отсутствуют какие-либо данные по времени исполнения тех или иных запросов в сравнении с реализациями других алгебр.

4. АЛГЕБРЫ, ИЗНАЧАЛЬНО ОРИЕНТИРОВАННЫЕ НА XML-ДАННЫЕ

4.1. Алгебра, использующая контексты

Авторы работы [7] поставили перед собой задачу, разработать алгебру, которая сможет работать с языком XQuery, но в тоже время простую — чтобы ее можно было легко реализовать и оптимизировать в дальнейшем.

Отличительной особенностью этой алгебры является структура *context* (далее — контекст), предназначенная для облегчения доступа к XML-элементам. Одним из компонентов этой структуры является позиция в XML-документе, благодаря которой контекст, используемый в операции, указывает путь в документе, по которому находятся аргументы операции. Несмотря на то что авторы не ставят основной задачей определение алгоритма вычисления запросов, они все же рассматривают некоторые особенности, которые необходимо учитывать при использовании контекста. Одна из них — необходимость передачи большого количества данных между операциями в дереве вычислений (дерево состоит из операций алгебры и строится на основе запроса XQuery). Например, может понадобиться копирование материнского узла в качестве контекста для каждого дочернего узла после операции, использующей в качестве аргумента данный материн-

ский узел. Эта проблема решается путем использования одного контекста для всех дочерних узлов вместо его копирования. Таким образом, для каждого дочернего узла будет сделана лишь ссылка на контекст. Другая особенность — это возможность альтернативных реализаций контекста в физической алгебре, в то время как в логической алгебре контекст определен однозначно. Это необходимо для эффективной оптимизации процесса вычисления запроса.

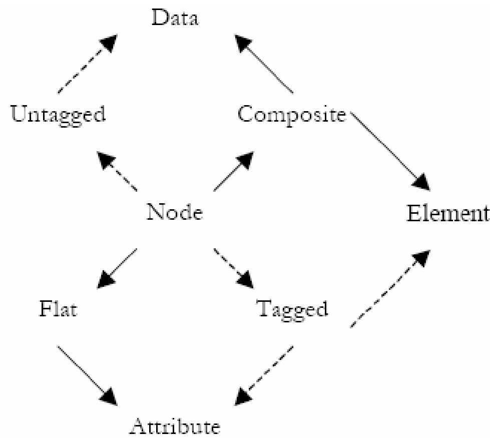
При использовании контекста, по мнению авторов, выражения становятся более простыми и прозрачными в понимании, что делает более легкими разработку и реализацию процессоров, эффективно вычисляющих и оптимизирующих запросы.

Алгебра, представленная авторами, манипулирует следующими данными.

1. XML-документы, описанные в стандартной спецификации WC3 [7]. Документ хранится в виде ориентированного дерева (XML-дерево), состоящего из узлов (элементы и атрибуты) и ребер (ребро элемента, атрибута, матери). Каждое ребро имеет имя, указывающее имя элемента или атрибута, следующего за этим ребром.

2. Путевые выражения, состоящие из последовательности имен ребер, упорядоченных таким образом, чтобы представлять собой путь в дереве до определенного узла. Путевое выражение не всегда уникально, так как некоторые узлы могут быть достигнуты несколькими способами (например, при использовании ссылок в XML-документе).

В алгебре используются два основных типа данных. Первый — это *Text*, который соответствует текстовому представлению XML-данных (например, содержанию элементов или атрибутов). Этот тип данных используется при сравнении значений XML-данных. Второй тип — это *node* (*узел*). Данный тип включает все структуры, встречающиеся в XML-документе, т. е. элемент (тип *element*), атрибут (тип *attribute*) и открытый текст (тип *data*). Тип *node* может быть уточнен следующим образом: *tagged node* — это узел, к которому ведет именованное ребро и *untagged node* — узел, к которому ведет безымянное ребро; *flat node* — узел без дочерних узлов (например, атрибут), *composite node* — узел с дочерними узлами. Для сравнения порядков узлов и вычисления предикатов и путевых выражений используются два дополнительных примитивных типа — *integer* и *boolean*. Иерархия типов изображена на следующем рисунке:



Сплошными и штриховыми стрелками изображены возможные альтернативы.

В алгебре определены следующие конструкторы:

1) конструктор для типа *data*, использующий в качестве аргумента параметр типа *text* и возвращающий в результате узел типа *data*;
2) конструктор для типа *attribute*, использующий в качестве аргументов два параметра типа *text* и возвращающий узел типа *attribute* с тегом, задаваемым первым параметром, и значением, задаваемым вторым параметром;
3) конструктор для типа *element*, использующий в качестве аргументов параметр типа *text*, содержащий в себе тег элемента, *m* параметров типа *attribute* и *n* параметров типа *composite node*, которые представляют собой содержание элемента (дочерние элементы и *data* узлы).

На узлах определяются пять функций:

1) *id*() — уникальное число для каждого узла, представляющее собой порядковый номер узла в XML-дереве;
2) *tag*() — имя ребра, указывающего на данный узел (только для *tagged node*);
3) *value*() — текстовое представление узла, используемое при сравнении значений узлов;
4) *content*() — все дочерние узлы данного узла;
5) *parent*() — материнский узел для данного узла.

В алгебре используются два предиката: *boolean* и *text-in-context*. Предикаты первого типа используются для сравнения значений, полученных с

помощью функций, описанных выше. Посредством предикатов второго типа задаются структурные связи между узлами, например, *contains*, *contained in*, *directly contains* и т.д.

Контекст — главная особенность рассматриваемой алгебры — указывает на позицию в XML-дереве. Контекст описывает то, каким образом предыдущие операции достигли данных узлов. Аргументы для каждой операции описываются контекстом и выражением *forward*, в котором указаны элементы, над которыми необходимо произвести вычисления. Таким образом, используется запись [*context*]*forward*, обозначающая все элементы, достижимые по пути *forward*, относительная позиция которых в XML-дереве указана в контексте. Символ $\perp$ обозначает нулевой контекст, ниже которого находятся все остальные узлы.

Все операции алгебры используют последовательность последовательностей в качестве аргументов, и в результате тоже получают последовательность последовательностей. Для удобства последовательность последовательностей названа *внешней*, а те последовательности, которые в ней содержатся — *внутренними*. Каждая внутренняя последовательность состоит из пар (*context*, *navigation*), функция *evalPE*(x,P) возвращает последовательность узлов для путевого выражения $P = [B]A$, если x принадлежит контексту B в этом путевом выражении. При определении операций используются следующие обозначения:

x,y,z, ... — узлы в XML-дереве;

a,b,c, ... — узлы контекста;

A,B,C,... — внешние последовательности;

K,L,M,... — внутренние последовательности;

$a \leftarrow A$ — а принимает все возможные значения из A;

P — путевое выражение;

y/x — узел x — дочка узла y;

$p(x)$ — предикат, имеющий значение *true* или *false* при вычислении на узле x.

В алгебре определены следующие операции.

1. α — производит переименование узлов на уровне тегов. Формальное определение:

$$\alpha(old,new)C = \{K \mid K \leftarrow C, (c, x) \leftarrow K, (x/y \wedge tag(y) = old \wedge rename(y,new))\}.$$

2. μ — расширяет внутренние последовательности, которые являются аргументами операции, добавляя в них узлы, достижимые с помощью путевого выражения. Если путевому выражению удовлетворяют несколько узлов, то в результат будет добавлено несколько последовательностей.

$$\mu(P)C = \{L \mid K \leftarrow C, (c, y) \leftarrow K, (x \in evalPE(y,P) \wedge L = K \cup \{(y,x)\})\}.$$

3. σ — выбирает во внутренних последовательностях элементы, удовлетворяющие заданному предикату:

$$\sigma(p)C = \{K \mid K \leftarrow C, (c,y) \leftarrow K, (p(y))\}.$$

4. π — удаляет из последовательностей ненужные узлы на основе списка путевых выражений:

$$\pi\,(P_1,P_2,\ldots)C = \{L \mid L \leftarrow \{(y,x) \mid K \leftarrow C, (c,y) \leftarrow K, (x \in evalPE(y,P_i))\}\}.$$

5. ⊗ — соединяет две внутренние последовательности из двух внешних последовательностей, если они удовлетворяют предикату соединения:

$$\otimes(p)(C,D) = \{M \mid K \leftarrow C, L \leftarrow D, (c,x) \leftarrow K, (d,y) \leftarrow L, (p(x,y) \wedge M = K \cup L)\}.$$

6. ε — добавляет к каждой внутренней последовательности результат, полученный путем применения к этой последовательности функции f:

$$\varepsilon(f)C = \{L \mid K \leftarrow C, (d,y) \leftarrow K, (L = f(y) \cup K)\}.$$

7. ρ — разбивает входную внешнюю последовательность на две внешних последовательности, причем первая из них одноэлементная и состоит из первой внутренней последовательности аргумента, затем применяет определенное алгебраическое выражение к одноэлементной внешней последовательности и рекурсивно разбивает вторую получившуюся последовательность:

$$\rho(E)C = \rho(E)\{K\} \cup \rho(E)A, \text{ где } C = [K \mid A].$$

При этом рекурсия прерывается, как только E возвращает пустую последовательность.

8. γ — группирует несколько внутренних последовательностей в одну на основе значений функции *f*. Для формального определения этой операции авторы вводят операцию *p*:

$$p(f)C = \{K \mid L \leftarrow C, M \leftarrow C, (a,x) \leftarrow L, (b,y) \leftarrow M, f(x) = f(y), K = L \cup M - \{(b,y)\}\}.$$

тогда операция group определяется следующим образом:

$$\gamma(f)C = \rho(p(f))C.$$

В дополнение к перечисленным выше операциям, для того, чтобы алгебра была полной, авторы включили в нее операции над множествами `union`, `intersection`, `difference` и `cartesian product`. Все эти операции работают с идентификаторами узлов, а не с их значениями. Операции-конструкторы (см. выше) для каждого типа узла добавляют но-

вый узел к входной внутренней последовательности. При этом новый узел имеет пустой контекст.

Далее авторы приводят пример преобразования выражения XQuery в разработанную ими алгебру. На момент написания работы оптимизатор для алгебры находился в разработке.

Основные отличия описанной выше алгебры от алгебры TAX следующие. В алгебре TAX шаблоны деревьев работают как конструкторы, копирующие только те XML-данные, которые удовлетворяют шаблону, в данной же алгебре копирования не происходит, вместо этого создаются контексты. Второе главное отличие — авторы не используют шаблоны деревьев для определения необходимой структуры XML-данных, эти задачи выполняет операция `unnest` и функции и предикаты алгебры. Главный недостаток алгебры — выбор операций, не ориентированных на XQuery, из-за чего процесс преобразования запроса XQuery в выражение алгебры не очевиден и иногда требует значительных вычислений.

4.2. Алгебра XAL

В работе [8] авторы описывают алгебру XAL, отличительными особенностями которой являются простая модель данных и четко определенные логические операции. Операции алгебры делятся на три категории: операции извлечения данных; мета-операции, контролирующие процесс вычисления выражений, и операции-конструкторы, позволяющие строить новые XML-данные на основе результата вычислений.

Каждый XML-документ может быть представлен в виде ориентированного корневого графа с определенным частичным порядком на ребрах. При этом узлы и ребра графа обладают некоторыми свойствами. Формально модель данных, используемая в алгебре, определяется следующим образом:

$G = (V,E,O,root)$, $root \in V$ — дерево XML-документа;

$V = V_{element} \cup V_{int} \cup V_{string} \cup V_{\mathrm{ID}} \cup V_{\mathrm{IDREF}} \cup \ldots$ — множество узлов;

$E = E_A \cup E_E \cup E_R \cup E_D$ — множество ребер;

$O = \bigcup_{p \in V} ((v_1,v_2,\ldots,v_n) | v_i \in V, parent(v_i)=p)$ — множество, задающее порядок на ребрах.

Множество V состоит из элементных узлов и узлов, содержащих текстовые данные. Тип узла может быть одним из двух: *element* и *simple* (все остальные типы). Узел *root* содержит одну дочку — корневой узел дерева XML-документа. Каждый узел имеет два первичных свойства и несколько вторичных. Вторичные свойства могут быть извлечены из свойств ребер.

Первичные свойства	для узла типа element	для узла типа simple
value	идентификатор узла	значение узла
type	element	тип значения узла
Вторичные свойства	результат	
name	имя узла	
parent	материнский узел	
parentedge	ребро, ведущее от материнского узла	
childelements	выходящие E-ребра (ребра, ведущие к узлам типа element)	
attributes	выходящие A-ребра (ребра, ведущие к узлам типа attribute)	
references	выходящие R-ребра (ребра, ведущие к узлам типа reference)	
Data	выходящие D-ребра (ребра, ведущие к узлам типа data)	

Для ребер также определены первичные и вторичные свойства. Первичные свойства — это *name* (имя), *type* (*A*,*E*,*R* или *D*), *parent* (материнский узел), *child* (дочерний узел). Вторичные свойства определены только для ребер типа *E* и *D*. К ним относятся: *next* (следующее по порядку ребро матери), *previous* (предыдущее ребро матери). Имя ребра — это имя соответствующего элемента/атрибута или строка “data”. Порядок ребер типа *E* и *D* задается множеством *O*, элементами которого являются последовательности дочерних узлов для каждого из узлов, принадлежащих *V*.

Как было сказано выше, операции алгебры делятся на три группы. Каждая унарная операция преобразует аргументы в коллекцию узлов и использует неявную операцию *map* для вычисления результатов, т.е. для каждого *x* из коллекции вычисляется функция *f* (функция над узлами), и результат помещается в выходную коллекцию.

Операции поиска данных

1. `Projection`[type,name](e). Аргументы: *type* — тип, *name* — строка, *e* — коллекция узлов. Результат: коллекция узлов, следующих за ребрами с именем *name* (*name* может быть регулярным выражением) и типом *type*.
2. `Selection`[condition](e). Аргументы: *condition* — выражение, *e* — коллекция узлов. Результат: узлы, удовлетворяющие условию

condition. Условие может содержать операцию projection (например, Selection[projection[*A*,*name*] = “Dali”](e)).

3. Unorder(e). Аргумент: *e* — коллекция узлов. Результат: неотсортированная коллекция.
4. Distinct(e). Аргумент: *e* — коллекция узлов. Результат: коллекция, преобразованная во множество.
5. Sort[value_expression(e)](e). Аргументы: *value_expression* — выражение, заданное на узлах коллекции, *e* — коллекция узлов. Результат: коллекция, отсортированная на основе значений, полученных с помощью *value_expression*.
6. Join[condition](x,y). Аргументы: *x*, *y* — коллекции, *condition* — выражение. Результат: декартово соединение двух коллекций, которое определяется двумя вложенными циклами: внешний цикл по первой коллекции и внутренний — по второй. В каждой итерации проверяется условие *condition* и, если узлы удовлетворяют этому условию, создается виртуальный узел, правой дочкой которого является узел из первой коллекции, левой — из второй. Результат операции — коллекция всех этих виртуальных узлов.
7. CartesianProduct(x,y). Аргументы: *x*, *y* — коллекции. Результат: декартово произведение двух коллекций. Частный случай Join с *condition* = *true*.
8. Union(x,y). Аргументы: *x*, *y* — коллекции. Результат: конкатенация последовательностей или объединение множеств.
9. Difference(x,y). Аргументы: *x*, *y* — коллекции. Результат: все узлы, содержащиеся в первой коллекции, но не содержащиеся во второй.
10. Intersection(x,y). Аргументы: *x*, *y* — коллекции. Результат: те узлы, которые присутствуют в обеих коллекциях.

Мета-операции

1. Map[f](e) = union($f(e_1)$,$f(e_2)$,$f(e_3)$,...,$f(e_n)$). Аргументы: *e* — коллекция, *f* — функция. Операция применяет функцию *f* к каждому элементу во входной последовательности.
2. KleeneStar[f](e) = e + f(e) + f(f(e)) + ... Аргументы: *e* — коллекция, *f* — функция. Операция применяет функцию *f* на коллекции до тех пор, пока не будет получен результат, совпадающий с результатом на предыдущей итерации.

Операции-конструкторы

1. `CreateVertex`[type](value). Аргументы: *type* — тип, *value* — строка. Операция создает узел заданного типа со значением *value*.
2. `CreateEdge`[type,name,parent](child). Аргументы: *type* — тип, *name* — строка, *parent*, *child* — узлы. Операция создает ребро между узлами *parent* и *child*.

Далее в статье авторы приводят ряд правил, по которым происходит оптимизация выражений алгебры XAL и алгоритм, реализующий эту оптимизацию. Рассматривается пример преобразования выражения XQuery в выражение алгебры XAL с последующей оптимизаций. Операции XAL схожи с операциями в реляционных алгебрах. Алгоритм оптимизации для алгебры XAL построен на основе алгоритма оптимизации для реляционных алгебр. В дальнейшем авторы планировали добавить новые правила для оптимизации, учитывая такие особенности XML-данных, как древовидная структура и т.д.

4.3. Алгебра для XML-запросов

В работе [15] авторы рассматривают алгебру, способную по их утверждению охватить семантику многих языков запросов для XML и включающую в себя набор правил оптимизации, во многом схожий с правилами оптимизации для реляционных алгебр. Алгебра использует простую систему типов, которая очень схожа с системой типов в XML Schema. Для простоты, авторы используют всего три скалярных типа (*boolena*, *integer* и *string*) и не включили в модель атрибуты элементов.

Авторы утверждают, что любой тип, заданный XML-схемой, может быть преобразован в заданную ими систему типов, но не наоборот. Любой синтаксически корректный XML-документ будет соответствовать при этом типу *UrTree* = *UrScalar* | ~ [*UrTree**], где конструкция ~[*e**] означает тип элементного узла с любым именем.

Множество операций алгебры включает в себя основные операции, обычно используемые в языках запросов, такие как `if-then-else`, `case`, `where`, `for-in-do`, `let` и т.д. Такой подход делает алгебру похожей на обычный язык запросов, и выражение XQuery, переписанное на язык алгебры, практически не отличается от изначального выражения. Формальное описание синтаксиса используемых в алгебре выражений можно найти в статье. Формального описания семантики операции в работе не дано.

Рассмотрим в качестве примера, как можно записать на языке алгебры операцию `join`, соединяющую элементы *book* с элементами *review* на основе значения элемента *tittle* (*bib0* и *review0* — некоторые фрагменты XML-документов):

```
for b in bib0/book do
  for r in review0/book do
      where value(b/title) = value(r/title) do
          book [ b/title, b/author, r/review ].
```

Пример выражения, которое можно получить в результате:

```
book [
    title [ "Data on the Web" ],
    author [ "Abiteboul" ],
    author [ "Buneman" ],
    author [ "Suciu" ]
    review [ "A darn fine book." ]
],
book [
    title [ "XML Query" ],
    author [ "Fernandez" ],
    author [ "Suciu" ]
    review [ "This is great!" ]
]
```

Тип результирующего выражения:

```
book [
    title [ String ],
    author [ String ]+
    review [ String ]
]*
```

В алгебру также включены 5 встроенных функций агрегации — *avg*, *count*, *max*, *min* и *sum*, аргументами которых является последовательность элементов, а результатом — значение типа *integer*.

Авторы приводят в работе правила оптимизации, с помощью которых упрощаются выражения алгебры.

Алгебра имеет множество проблем, которые еще предстоит разрешить:

1) алгебра не использует множеств (только последовательности), что делает невозможным применение некоторых правил оптимизации, характерных для реляционных алгебр;
2) алгебра не поддерживает ссылочных значений, которые определены в XML Query Data Model [13];

3) возможна потеря данных при преобразовании внутренних типов к типам XML-схемы;
4) система типов мономорфна, т.е. нельзя определять полиморфные функции;
5) отсутствует явная поддержка внешних функций;
6) алгебра не гарантирует конечность вычислений в рекурсивных выражениях;
7) в алгебре фактически нет операций высокого уровня, манипулирующих коллекциями в целом.

4.4. Алгебра для XQuery

Описываемая в работе [17] алгебра охватывает практически все возможности XQuery. Алгебра использует достаточно простую модель данных, представленную W3C [13]. Это дало возможность определить набор простых операций (нецелесообразно при такой простой модели определять операции высокого порядка, которые используются в других алгебрах), что позволяет упростить перевод запроса XQuery на язык алгебры.

Модель данных описана посредством многоосновной алгебры, что дало авторам возможность формально определить все операции предложенной алгебры. Система типов модели данных включает в себя набор атомарных типов (*Boolean*, *Integer* и т.д.) и тип *untypedAtomic*, значениям которого не приписан более конкретный тип. Элементами коллекций могут быть только атомарные значения и узлы. Каждый узел принадлежит типу *Node*, который является объединением типов *Document*, *Element*, *Attribute*, *Text*, *Namespace*, *ProcInstruction* и *Comment*. Каждый узел и каждое атомарное значение принадлежат также типу *Item*, который, в свою очередь, является объединением типов *anyAtomicType* (объединение всех атомарных типов и *untypedAtomic*) и *Node*.

Предполагается, что для каждого атомарного типа существуют операции приведения значений этого типа к значениям других атомарных типов. Система типов также включает в себя конструкторы множеств, последовательностей и объединений $Set(t)$, $Seq(t)$, $Union(t_1,\ldots,t_n)$, где t, $t_1,\ldots,t_n$ — типы, и конструктор перечисления $Enumeration(I_1,\ldots I_n)$, где $I_1,\ldots I_n$ — идентификаторы.

В дополнение к стандартным типам авторы используют тип *запись*. Для типа запись $rec\ p_1{:}t_1, \ldots, p_n{:}t_n\ end$ заданы операция конструирования (создает новую запись, используя в качестве аргументов значения типа $t_1,..,t_n$) и операция проекции, возвращающая значение поля записи.

Следующие операции определены для каждого множества: “$\cup$” (объединение), “$\cap$” (пересечение), ”$\subset$” (включение), ”$\in$” (принадлежность) и *count* (количество элементов). Количество элементов последовательности обозначается $|s|$. К последовательностям и множествам, содержащим числовые значения, применимы операции *avg*, *sum*, *max* и *min*.

Схема базы данных определяет сигнатуру $\Sigma = (T,F)$. В дополнение к вышеперечисленным операциям F содержит в себе аксессоры узлов, определенные в [13], все имена функций и операций, определенных в [19], имена навигационных операций и некоторые специальные константы. В F также содержатся функции, упорядочивающие узлы в документном порядке и наоборот.

Используя операции из F, можно строить выражения, которые в дальнейшем могут быть вычислены, или интерпретированы. Выражения алгебры можно разделить на три группы:

- обычные алгебраические выражения, вычислимые в той же сигнатуре и алгебре, в которой заданы;
- выражения, написанные в одной сигнатуре, но интерпретированные в алгебре обогащенной сигнатуры (такой является, например, операция соединения двух отношений в реляционной алгебре);
- выражения, чьи интерпретации изменяют алгебру, и производят элементы новой алгебры (например, конструкторы новых узлов).

Для каждого узла определены навигационные функции в дополнение к аксессорам. Эти функции позволяют получать последовательности дочерних и материнских узлов. Точное определение навигационных функций можно найти в данной работе.

1. `child` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность, содержащую все дочерние узлы узла.
2. `descendant` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность всех потомков узла.
3. `descendant_or_self` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность, содержащую все потомки узла и сам узел.
4. `parent` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность, содержащую материнский узел.
5. `ancestor` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность предков узла.
6. `ancestor_of_self` : *Node* $\rightarrow$ *Seq*(*Node*). Возвращает последовательность, содержащую все предки узла и сам узел.

7. `following_sibling` : *Node* → *Seq*(*Node*). Возвращает последовательность сестер узла, следующих за ним в документном порядке.
8. `following` : *Node* → *Seq*(*Node*). Возвращает последовательность, содержащую все узлы, следующие за данными в документном порядке, за исключением потомков данного узла.
9. `preceding_sibling` : *Node* → *Seq*(*Node*). Возвращает последовательность всех сестер узла, предшествующих самому узлу в документном порядке.
10. `preceding` : *Node* → *Seq*(*Node*). Возвращает последовательность, содержащую все узлы, предшествующие данному узлу в документном порядке, за исключением предков данного узла.
11. `attribute` : *Node* → *Seq*(*Node*). Возвращает последовательность всех атрибутных узлов данного узла.

Перевод FLWOR выражения на язык алгебры выполняется по шагам. Для начала рассмотрим вспомогательные выражения, которые используются для поддержки некоторых выражений XQuery.

Частью выражений XQuery могут быть путевые выражения, которые позволяют задавать позицию в дереве. Эти выражения интерпретируются в алгебре с использованием навигационных функций. Пусть x, y — переменные, s_1 — последовательность узлов, t — атомарный тип или тип узла, s_2 — последовательность элементов типа t, тогда `path`(y : s_1)/s_2 — последовательность элементов типа t. Выражение s_1 и s_2 называются *левым шагом* и *правым шагом* соответственно. Левый шаг используется для выбора последовательности узлов, над которой будут произведены вычисления выражения правого шага. Если тип s_2 — атомарный, то результат вычисления путевого выражения будет конкатенацией результатов вычисления s_2 для каждого $y \in s_1$, если же тип s_2 — узел, то мы получим последовательность, состоящую из результатов вычисления s_2 для каждого y и отсортированную в документном порядке.

Еще одна особенность выражений XQuery — использование кванторов универсальности и существования. Семантики соответствующих выражений алгебры определены стандартно. Используемый синтаксис: `forall`(x_1 : s_1, ..., x_n : s_n)!b и `exists`(x_1 : s_1, ..., x_n : s_n)!b, где s_1, ..., s_n — такие последовательности, что каждая следующая может зависеть от предыдущих, b — булевское выражение.

XQuery содержит в себе набор выражений для создания последовательностей и управления ими. Поэтому в алгебре определены следующие опе-

рации для работы с последовательностями: `seq`(e_1, …, e_n) (где e_1, …, e_n — выражения), `range`(e_1, e_2) (в данном случае e_1 и e_2 возвращают целые числа, результат операции — последовательность чисел от e_1 до e_2), `except`(s_1,s_2), `union`(s_1,s_2), `intersect`(s_1,s_2). Если e_1 и e_2 — выражения типа *Seq*(*anyAtomicType*) и Θ — один из символов отношения "=", "!=", "<", ">", "<=", ">=", то s_1 Θ s_2 — выражение типа Boolean (с интерпретацией этих выражений можно ознакомиться в рассматриваемой работе).

Алгебра содержит в себе выражения, с помощью которых можно естественным образом записать выражение FLWOR. Рассмотрим их по порядку.

Для поддержки различных видов операций FOR и LET в алгебре определены три вспомогательных выражения.

1. <·*y* : *s*·> — преобразует последовательность *s* в последовательность записей типа *rec y:t end,* где *t* — тип элементов *s*. Это выражение необходимо для поддержки операции FOR, содержащей одну присваиваемую переменную.
2. <·*y*, *i* : *s*·> — преобразует последовательность *s* в последовательность записей *s′* = *rec*(*i* ,*s*[*i*]), где *I* = *1*,...,|*s*|. Это выражение необходимо для поддержки операции FOR, содержащей в себе одну присваиваемую переменную и одну позиционную переменную.
3. <·*y* = *e*·> — возвращает последовательность, состоящую из записи типа *rec y:t end*, где *t* — тип выражения *e*. Это выражение необходимо для поддержки операции LET.

Следующее выражение поддерживает любую комбинацию операций FOR и LET: s_1*s_2, где s_1 — выражение, возвращающее последовательность записей типа *rec* $x_{11} : t_{11}, ..., x_{1m} : t_{1m}$ *end*, а s_2 — выражение, возвращающее последовательность записей типа *rec* $x_{21} : t_{21}, ..., x_{2n} : t_{2n}$ *end*. Результат вычисления s_1*s_2 имеет тип *Seq*(*rec* $x_{11} : t_{11}, ..., x_{1m} : t_{1m}, x_{21} : t_{21}, ..., x_{2n} : t_{2n}$ *end*). Выражение s_1*s_2 вычисляется следующим образом: для каждого элемента из s_1 вычисляется выражение s_2, после чего результаты вычисления s_1 и s_2 заносятся в одну запись (сначала элементы из s_1, потом из s_2), которая добавляется к результирующей последовательности.

Пример: вычисление выражения <·*x* : (1, 2, 3)·>*<·*y* = (*x*+1, *x*+2)·> произведет следующий результат — (⟨1, (2,3)⟩, ⟨2, (3,4)⟩, ⟨3, (4,5)⟩). Если *books* — это последовательность книг, то выражение <·*x*: *books*·>*<·*y*:*x*.*child*::*element*(*authors*)·> будет последовательностью пар, где каждый "книжный" узел будет в паре с каждым "авторским" узлом этой

книги. Как мы видим, часть выражения XQuery, содержащая операции FOR и LET, естественным образом интерпретируется в алгебре.

Selection-выражения служат для выбора части последовательности на основе некоего критерия (операция WHERE). В отличие от реляционных алгебр, такое выражение в XML-алгебре может включать в себя проверку типа (kind test) в дополнение к проверке предикатом. Эти выражения сохраняют порядок в последовательности и не изменяют алгебру.

Пусть *s* — последовательность узлов, тогда проверка типа выглядит следующим образом:

1) *s* :: *element*() — последовательность всех элементов из *s*,
2) *s* :: *attribute* — последовательность атрибутов из *s* и т.д.,
3) *s* :: *element*(*n*,t) — последовательность элементов типа *t* с именем *n* (более подробно с видами kind test можно познакомится в данной работе).

Выражение проверки предикатом выглядит следующим образом: `select`(*y* : s) :: *p*, где *y* — переменная, принимающая значения из последовательности *s*, а *p* — предикат (выражение типа *Boolean*), зависящий от *y*. Для каждого *y* вычисляется предикат *p* и в возвращаемую последовательность добавляются те элементы, которые прошли проверку. Например, выражение `select`(*x*: *books*) :: *typed-value*(*x.attribute* :: *attribute*(*year*)) = 2000 определяет последовательность “книжных” узлов для книг, опубликованных в 2000 году.

Операция ORDER BY упорядочивает последовательность записей, полученных в результате выполнения предыдущих операций, основываясь на значениях, вычисленных для каждой записи. В представленной алгебре упорядочивающее выражение сортирует записи на основе одного или нескольких ключей порядка, которые являются пустыми или одноэлементными последовательностями. Синтаксис операций выглядит следующим образом:

`stable_order`(e_1,[a_1,b_1,c_1], …, e_m[a_m,b_m,c_m] : *s*) и
`order`(e_1,[a_1,b_1,c_1], …, e_m[a_m,b_m,c_m] : *s*),

где *s* — последовательность, e_1, ..., e_m — ключи сортировки, a_k и b_k — один из символов “↑” или “↓” (a_k указывает на восходящий или нисходящий порядок, b_k — на положение пустых последовательностей) и c_k — пустая строка, если t'_k не является типом string, и, возможно, не пустая иначе. В результате получаем последовательность, содержащую те же элементы, что и исходная, но упорядоченную так, как это задают *a*, *b* и *c*. Второе выражение

отличается от первого только сохранением относительного положения элементов с равными значениями ключей порядка.

Операция RETURN конструкции FLWOR представлена в алгебре в виде выражения *mapping*. Если s — выражение типа $Seq(rec\ x_1 : t_1, \ldots, x_n : t_n\ end)$ и e — типа $Seq(t)$, то $s \blacktriangleright e$ — выражение типа $Seq(t)$, называемое выражением *mapping*. Для каждого элемента последовательности s вычисляется выражение e и результат добавляется к возвращаемой последовательности. Порядок возвращаемой последовательности совпадает с порядком исходной.

Таким образом, мы можем увидеть, что конструкция FLWOR преобразуется в выражение алгебры достаточно просто, при этом тип записи используется только внутри этого выражения.

Отдельно в работе рассмотрены конструкторы узлов — набор выражений, копирующих узлы или создающих новые узлы. Интерпретация этих выражений меняет алгебру и создает элемент новой алгебры. Поэтому для указания результата выражения используется нотация $\langle A, v\rangle$, где A — Σ-алгебра, $v \in |A|$ — элемент алгебры. Множество всех пар $\langle A, v\rangle$, где v — значение типа t, обозначено $A_t(\Sigma)$. Функции *fst* и *snd*, применяемые к таким парам, возвращают первый и второй компонент соответственно. Предполагается, что сигнатура алгебры содержит перечислимый тип constructionMode = Enumeration{strip, preserve} и константу con_mode типа constructionMode, которая управляет значениями некоторых асессоров узлов. В алгебре определены следующие конструкторы.

1. `copy_node`(nd, end). Аргументы: *nd* — выражение типа *Node*, *end* — элементный узел. Результат: копия узла *nd* со всеми дочерними узлами, который присоединяется к узлу *end* в качестве дочки.
2. `copy_nodes`(s, end). Аргументы: *s* — выражение типа *Seq*(*Node*), *end* — элементный узел. Результат: копии узлов из *s* со всеми дочерними узлами, которые присоединяется к узлу *end* в качестве дочки.
3. `attribute_node`(n,e). Аргументы: *n* — *QName*, *e* — строка. Результат: новый атрибутный узел с именем *n* и значением *e*.
4. `text_node`(e). Аргумент: e — строка. Результат: текстовый узел со значением e.
5. `element_node`(n,atseq,e). Аргументы: *n* — *QName*, *e* — строка, *atseq* — последовательность атрибутных узлов. Результат: элементный узел с атрибутами из *atseq*, именем *n* и значением *e*.
6. `element_node`(n,atseq,elseq). Аргументы: *n* — *QName*, *elseq* — последовательность элементных и текстовых узлов, *atseq* — после-

довательность атрибутных узлов. Результат: элементный узел с атрибутами из *atseq*, именем *n* и дочерними элементными и текстовыми узлами из *elseq*.

7. `document_node`(elseq). Аргументы: *elseq* — последовательность элементных и текстовых узлов. Результат: документный узел с дочерними элементными и текстовыми узлами из *elseq*.

Формальные определения конструкторов можно найти в работе.

Представленная алгебра является набором простых выражений, с помощью которых строятся более сложные операции. Этот набор существенно отличается от операций реляционных алгебр, в основном, из-за более сложной структуры XML-документа. Только одна операция реляционной алгебры — `select` — немного похожа на описываемую в статье, но только той частью, где происходит проверка предиката. Операция `project` заменена путевыми выражениями, операция `join` — набором выражений, позволяющем создавать поток записей на основе нескольких, возможно, вложенных частей документа. Также определены выражения, позволяющие создавать новые узлы. Побочным эффектом этих выражений является создание новой алгебры, и, так как в XQuery возможны вложенные выражения, то любое выражение может обладать таким побочным эффектом. Еще одной особенностью алгебры является задание контекста первой операцией в сложном выражении для второй операции, что может использоваться при оптимизации вычислений запроса. К сожалению, авторы не рассматривают вопросов оптимизации выражений в работе, также отсутствует какая-либо реализация алгебры, и поэтому сложно что-либо сказать о скорости вычислений запросов XQuery и сравнить это с реализациями других алгебр.

ЗАКЛЮЧЕНИЕ

В данном обзоре рассмотрены работы по алгебрам для XQuery. Можно выделить два основных подхода при построении алгебры: первый заключается в расширении возможностей реляционных алгебр, путем добавления необходимых операций для работы с XML, например, работы [9,16]; при использовании второго подхода алгебра строится с нуля, как сделано в работах [3,17]. В обоих случаях есть как преимущества, так и недостатки. При построении алгебры на основе реляционной алгебры можно использовать уже имеющиеся правила оптимизации выражений, и сохраняется возможность работать не только с XML-данными, но и с данными в реляционном формате. Минусы этого подхода тоже очевидны — недостаточная гибкость

алгебры не позволяет использовать все преимущества XML без специальных ухищрений. При втором подходе обычно используется один из двух видов моделей данных: сложная модель, с деревьями в качестве информационных единиц [3,5], и простая модель, основанная на модели, предложенной WC3 [13].

В большинстве рассмотренных работ отсутствует формальное определение семантики операций, что делает построение реальной системы базы данных затруднительным. Во всех работах, кроме последней, рассмотрен лишь синтаксический перевод выражения XQuery на язык алгебры. Семантика запроса при таком переводе может не сохраниться, и в результате полученные значения могут сильно отличаться от необходимых. Некоторые алгебры (например, TAX и XAT) ориентированны на реальные системы баз данных, а не на формальную модель данных XQuery, что наложило существенный отпечаток на семантику операций. Так, например, алгебра TAX использует довольно сложную модель данных, что сподвигнуло авторов к разработке физической алгебры, на основе которой в дальнейшем были разработаны другие алгебры. В алгебре XAT отсутствуют операции, создающие или изменяющие существующие элементы. На момент написания работы алгебры для системы TIMBER и алгебра XAT использовались в коммерческих проектах, что подтверждает их практическую применимость, но при использовании этих алгебр для задания семантики XQuery необходимо привести модель данных к модели данных XQuery.

В некоторых работах задача оптимизации полученных алгебраических выражений ставится на первое место. Примером такой алгебры является алгебра, чьи операции работают в заданном контексте (разд. 4.1). Авторы утверждают, что понятие контекста позволит эффективно оптимизировать процесс вычисления запроса. Но при этом использованы операции, не ориентированные на работу с XQuery. Перевод запроса на язык алгебры сильно затруднен.

В других работах сделана попытка охватить множество языков запросов. Например, авторы алгебры, описанной в разд. 4.3, утверждают, что их алгебра способна охватить семантику многих языков запросов, но в качестве примера приводится лишь перевод выражений XQuery. К сожалению, алгебра имеет множество недостатков, на которые указывают сами авторы.

СПИСОК ЛИТЕРАТУРЫ

1. XQuery 1.0: An XML Query Language, W3C Candidate Recommendation, 3 November 2005. — available at http://www.w3.org/TR/xquery/.
2. Jagodish H. V. et al. TIMBER: A Native XML Database // The VLDB J. — 2002. — Vol. 11, Is. 4. — P. 274-291.
3. Jagodish H. V. et al. Tax: A Tree Algebra for XML // Proc. Intl. Workshop on Databases and Programming Languages, Marino, Italy, 2001. — P. 149-164.
4. Paparizos S. et al. A Physicial Algebra for XML. — available at http://www-personal.umich.edu/spapariz/publications.html
5. Chen Zh. et al. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of Xquery // Proc. VLDB Conf., Berlin, Germany, 2003.
6. Paparizos S.et al. Tree Logical Classes for Efficient Evaluation of XQuery // Proc. SIGMOD Conf., Jun. 2004, Paris, France.
7. Viglas S. D.et al. Putting XML Query Algebras into Context. — available at http://www.cs.wisc.edu/niagara.
8. Frasincar F., Houben G.-J., Pau C. Xal: an algebra for xml query optimization // Database Technologies, 2002.
9. Sartiani C., Albano A. Yet Another Query Algebra For XML Data // IDEAS, 2002. — P. 106-115.
10. Colazzo D., Manghi P., Sartiani C. Xtasy: A typed xml database management system. — 2001 — available at http://www.di.unipi.it/~sartiani/papers/mementomori.pdf.
11. Christophides V., Cluet S., Sim´eon J. Semistructured and Structured Integration Reconciled: YAT += Efficient Query Processing. — (Tech. rep. / INRIA, Verso database group).
12. Cluet S. and Moerkotte G. Classification and optimization of nested queries in object bases. — Karlsruhe, 1994. — (Tech. rep. / University of Karlsruhe).
13. XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Candidate Recommendation, 3 November 2005. — available at http://www.w3.org/TR/xpath-datamodel/.
14. The XML Query Algebra. W3C Working Draft 04 December 2000. — available at http://www.w3.org/TR/2000/WD-query-algebra-20001204/
15. Fernandez M., Simeon J., Wadler Ph. An Algebra for XML Query // Proc. FST TCS, Delhi, December 2000.
16. Zhang X. and Rundensteiner E. XAT: XML Algebra for Rainbow System. — (Tech. Rep. / Worcester Polytechnic Institute; WPI-CS-TR-02-24).
17. Novak L., Zamulin A. An XML-algebra for XQuery. — Novosibirsk, 2005. — (Prepr. / A. P. Ershov Institute of Informatics Systems; No 125).
18. XML Path Language (XPath) 2.0, W3C Candidate Recommendation, 3 November 2005. — available at http://www.w3.org/TR/xpath20/.

19. XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Candidate Recommendation, 3 November 2005. — available at http://www.w3.org/TR/xpath-functions/.
20. Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, 04 February 2004. — available at http://www.w3.org/TR/2004/REC-xml-20040204/.
21. XML Schema Part 1: Structures Second Edition, W3C Recommendation, 28 October 2004. — available at http://www.w3.org/TR/xmlschema-1/.
22. Ehrig H., Mahr B. Fundamentals ofAlgebraic Specifications 1, Equations and Initial Semantics // EATCS Monographs on Theor. Comput. Sci. — Berlin: Springer, 1985— Vol. 6.

В. В. Кальченко

ОБЗОР АЛГЕБР ДЛЯ XML-БАЗ ДАННЫХ

Препринт
127

Рукопись поступила в редакцию 27.10.05
Редактор З. В. Скок

Подписано в печать 13.12.05
Формат бумаги 60 × 84 1/16 Объем 2.9 уч.-изд.л., 3.1 п.л.
Тираж 60 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383) 330-72-02