摘　要

面向空间数据集成的 XQuery 查询执行引擎系统实现了对分布式空间数据集成应用和共享的支持，把网络环境中的多个空间数据库以信息集成的方法联系起来，实现空间数据共享。由于数据源的异构性和分布存储特性，以及空间数据查询的复杂性，如何提高查询效率，优化系统查询性能，使用户层在最短的时间内得到正确的查询结果成为目前亟待解决的问题。

本文研究面向查询优化的 XQuery 查询引擎系统，采用延迟求值的求值策略实现对执行引擎的优化。本文从两个方面实现延迟求值策略。第一，执行引擎通过对中间语言表达式的求值完成具体的查询操作，因此采用暂存表达式计算信息的方法将对表达式的求值向后延迟；第二，考虑到序列是 XQuery 查询引擎系统数据模型中最主要的数据结构，因此通过重新定义序列结构并重写与序列操作相关的操作函数，从而达到延迟对序列的计算的目的。

本文通过对上述两个方面的优化，初步实现了加快查询速度、提高查询效率的目的，为系统的进一步优化与完善打下了基础。

关键词 查询优化；延迟求值；XQuery

Abstract

The XQuery for geo-referenced data integration which implements sustain for applying and sharing of distributed spatial data source contracts several spatial data integration system in the environment of Internet via information integration implements the sharing of spatial data. Because of heterogeneous data source and its distributed characteristic, and the complicacy of query, how to make users get the correct result in the shortest time by increasing query efficiency and optimizing the query system becomes an urgent problem needed to be solved.

This paper studies the XQuery system which implements the query's optimization via the strategy of lazy evaluation. The paper carries out this lazy evaluation strategy in two ways. In the first place, save the information of expression so that it can be evaluated later because the query is operated based on interim language. In the second place, define a new structure for sequence and rewrite the functions on it considering sequence is the primary data structure in XQuery system's data model.

This paper intends to optimize the XQuery system and make query faster and more efficient, which lays the foundation for further optimization and perfection.

Keywords: query optimize; lazy evaluation; XQuery

目　录

第 1 章 绪论

1.1 课题简介

1.1.1 课题背景

地理信息系统（Geographical Information System，GIS）是一种获取、处理、管理和分析地理空间数据的重要工具。从技术和应用的角度，GIS 是解决空间问题的工具、方法和技术；从科学的角度 GIS 是在地理学、地图学、测量学和计算机科学等学科基础上发展起来的一门学科，具有独立的学科体系；从功能上，GIS 具有空间数据的获取、存储、显示、编辑、处理、分析、输出和应用等功能；从系统学的角度，GIS 具有一定结构和功能，是一个完整的系统[1]。自地理信息系统的第一个商业化产品诞生 30 多年以来，GIS 被广泛的应用于资源调查、环境评估、灾害预测、国土管理、城市规划、邮电通讯、交通运输、军事公安、水利电力、公共设施管理、农林牧业、统计、商业金融等几乎所有领域[2]。

随着计算机网络技术的发展和数据库技术的发展，地理信息系统得到了更加迅速的发展，越来越多的空间数据存储在网络上。为了实现数据共享，可以使更多的人更充分地使用已有的数据资源，减少资料收集、数据采集等重复劳动，已经建立了能够支持分布式空间数据集成应用和共享的框架，把网络环境中的多个空间数据库以信息集成的方法联系起来。考虑到数据源的异构性和分布存储特性，以及空间数据查询的复杂性，如何提高查询效率，优化系统查询性能，使用户层在最短的时间内得到正确的查询结果就成为一个迫切需要解决的问题。

1.1.2 课题研究任务

目前，课题组已经研究并实现了面向空间数据集成的 XQuery 查询引擎系统 GeoQuery。本课题的研究任务是面向查询优化的 XQuery 查询引擎系统，即在原有系统（以下简称“原系统”）的基础上对查询执行引擎模块其进行扩展，重新分析查询语法树，实现支持延迟求值和积极求值的查询计划执行机制，使执行引擎能够针对不同求值表达式的分析结果判断并应用不同的求值策略，以避免不必要的求值步骤。在简化求值过程的同时，节省时间资源和空间资源，从而达到加快查询速度，提高查询效率的效果，最终使系统得到优化。

扩展的查询执行引擎要求能够实现多源 XML 数据查询，查询采用 Java 语言开发，所开发的引擎将集成到原系统中，实现基于 XQuery 语言的多数据源查询。

1.1.3 研究目的和意义

面向空间数据集成的 XQuery 查询执行引擎消除了各种存储在不同数据源中的空间数据之间的差异，支持空间数据资源的共享与互操作。因为空间查询的复杂性和屏蔽不

同数据源、为用户提供统一访问接口的需求，所以查询系统的执行效率会受到影响，查询速度缓慢。

本课题的研究目的是为面向空间数据集成的查询语言提供高效率的执行引擎系统，支持基于 Internet 的地理信息应用系统的开发；提高用户查询的速度与效率，减少因数据源的异构性和空间查询的复杂性引起的查询速度慢、效率低的不足，使用户层在最短的时间内得到正确的查询结果。

随着地理信息系统在各个领域的应用，一方面，越来越多的数据存储在网络上，数据源的存储位置不同，每一个数据都可以分布在世界任意一台计算机上；存储格式各异，如存储于 XML 文档或 Oracle 数据库中等等。另一方面，访问这些数据的用户的数量也在急速的增长，他们希望通过数据共享减少资料收集、数据采集等重复劳动和相应费用，而把精力重点集中在开发应用程序及系统集成上。因此，能够高效快速的访问各种不同的软硬件平台，查询不同的专业词汇和数据标准、不同的数据组织方法和数据格式的查询执行引擎成为地理信息系统继续发展的关键技术之一，这也是本课题研究与开发的意义所在。

1.2 Mediator 结构概述

1.2.1 基于 Mediator 的空间数据集成方法

在空间数据处理方面，存在多种处理办法，使用基于 Mediator 的系统处理数据就是方法之一。基于 Mediator 的系统采用三层体系结构，包括用户层、Mediator 层和数据层。其中用户层又称应用层，由系统的客户程序组成，向下层发起空间查询；Mediator 层位于中间处理查询；数据层即为用 Wrapper 包装的各种数据源，是系统的最底层。

由此可见，Mediator 层位于三层结构的中间，它接受上层的查询语句，将查询经过处理后访问下层数据库中的数据，再将访问结果重新组织返回给用户层，成为整个查询系统的核心。

1.2.2 Mediator 查询处理流程

采用 Mediator 结构的查询处理流程分为六个步骤，即语法分析、查询分解、规范化、静态类型检查、翻译为 FXQL（有关 FXQL 的内容，参见第 2 章）和执行引擎。图 1 为查询在 Mediator 层的处理流程，描述了对用户发出的查询的处理流程。

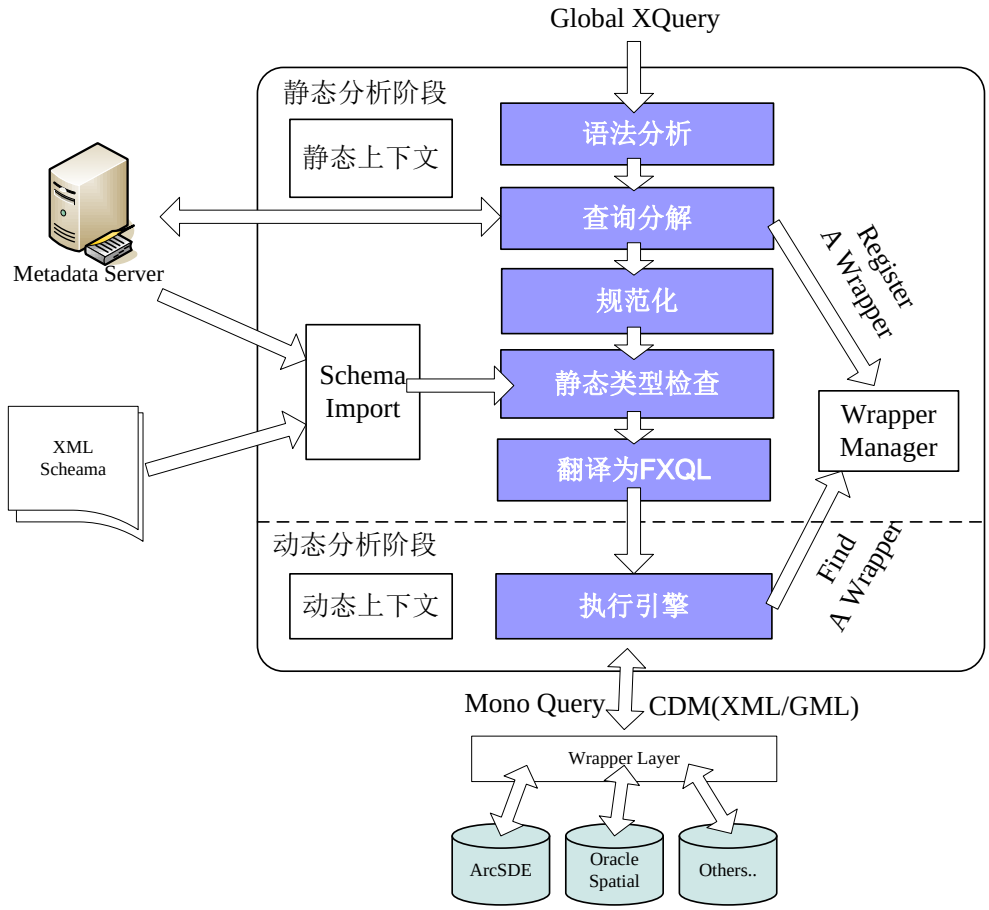


图 1 查询在 Mediator 层的处理流程

查询处理流程中各步骤完成的功能是：

- 语法分析：对输入的 XQuery 查询语言进行词法、语法分析，生成语法树，并将 XQuery 查询序言中的内容导入到 XQuery 的静态上下文中。
- 查询分解：根据各数据源的能力，将全局的针对多数据源的查询分解为多个单数据源子查询。
- 规范化：将 XQuery 标准文法的语句转化为 XQuery 核心文法的语句。
- 静态类型检查：对 XQuery 语法树进行类型检查和类型推导，为 XQuery 核心语法树上的每个表达式结点生成类型信息，包括表达式返回值类型，函数调用版本等。
- 翻译为 FXQL 中间语言：将 XQuery 核心语法树翻译成为 FXQL 语法树。
- 查询计划的执行：由执行引擎根据 FXQL 语言的语义对 FXQL 语法树进行解释执行，通过具体查询动作，完成对数据库的访问和查询，得到最终查询结果。

1.3 求值策略问题

1.3.1 原系统求值引擎分析

原系统的查询执行引擎采用积极求值策略完成查询。积极求值又称应用序求值或者

先行求值，是一种先求出运算对象，而后再应用运算符或函数的方法[3]。在原系统中，执行引擎处理 FXQL 语法树（如图 2 所示）时采用后序遍历的方法，即在对运算符确定的子表达式求值之前，先完成其各子树的求值。

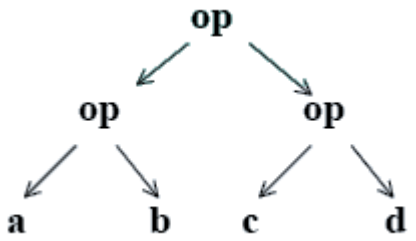


图 2 语法树示意图

积极求值策略遍历所有子树的要求使在计算过程中某些不会被使用到的运算对象也需要计算出结果，这样就产生了不必要的求值过程。在大规模的查询中，这种无需计算的运算对象大量存在，因此影响了求值速度。为避免这样的运算，提高求值效率，采用并实现一种新的求值策略成为本课题研究的关键所在。

1.3.2 延迟求值策略简介

延迟求值，也称正则序求值或者懒求值。这种求值策略的思路是仅对表达式中需要参加计算的运算对象求值，其余运算对象处于未求值状态[3]。这种求值策略可以避免积极求值过程带来的大量的不必要的运算过程，从而达到提高执行效率，优化查询引擎的效果。因此，如何设计并实现一个同时支持积极求值和延迟求值两种策略的查询执行引擎，成为本课题的研究方向。

第 2 章 函数式 XML 查询语言——FXQL

数据库的查询计划是由语法树表示的，所有的具体查询操作均依据语法树执行。XQuery 是基于 XML 的查询语言，而 XQuery 的查询语法树由其中间语言 FXQL 提供的。它是查询计划的描述手段，用于构成定义简单且易于计算机执行的查询计划。最终查询的执行效率在很大程度上取决于 FXQL 设计的好坏与否。

2.1 XML 语言

计算机网络技术的发展使人们可以随时随地查询网络上的数据，这需要为信息交换提供一套能够独立于任何特定语言和应用程序，并且易于扩展的组织原则。XML 作为一种主流的组织原则，已经被广泛的接受和使用。

可扩展标记语言（eXtensible Markup Language，XML）通过标记用于定义数据本身的结构和数据类型，提供一种描述结构化数据的方法，它提供了一种标记信息中各个部分并且能够描述各部分之间关系的中性符号。由于 XML 对存储技术没有特殊需求，因此逐渐成为数据存储系统中的公共交换格式。相对于其它格式，使用 XML 存储信息具有若干优越性：

- XML 格式是基于文本的，具有平台无关性，不仅易读，而且便于记录和调试；
- 信息的内容与信息的表示是分开的，能够满足各种不同的应用需求；
- 具有自描述性，可以自定义数据集的结构信息；

由于具有上述优点，XML 被大量应用于异构系统间的数据交换、数据集成、数据共享、将同一数据以不同形式表现出来等方面。XML 作为一种文档标记语言，被接受、应用和发展起来。目前，XML 不仅仅是一个技术规范，而且已经形成了一整套技术规范体系，在电子商务、金融行业等多个领域得到了广泛的应用。

2.2 XQuery 语言

随着信息存储和交换量的增加，以及对 XML 的广泛使用，如何智能的查询 XML 数据源变得越来越重要。XQuery 是 XML Query 的缩写形式，是一种基于 XML 的功能强大的数据查询语言，是用于查询各种类型的 XML 数据源，能够从 XML 文档中选择并提取出复杂的模式，进而把查询结果组织成新的 XML 文档。XML 的特点之一是可以描述不同的数据源，因此 XQuery 语言具备对不同数据源进行检索和解释的能力。

XQuery 语言具有以下特点：

1. XQurey 是无层次的、无序的、支持异构的查询语言。
2. XQuery 将查询表示为表达式，各种 XQuery 表达式可以嵌套使用，执行查询的过程即为对表达式求值的过程。
3. XQuery 可以把查询结果按照用户层的需求组织成新的 XML 文档，这使其非常

适合承担结构重构的任务。

4. XQuery 是一种典型的函数式语言。在该语言中，表达式能够被任意的组合，并允许嵌套，因此可以查询数据源中未知位置的数据。
5. XQuery 是一种强类型语言。XQuery 的类型系统基于 XML Schema，可以从一个或几个 XML Schema 导入类型；表达式的操作数必须和操作符要求相配，函数参数必须符合指定类型；支持静态类型分析。

2.3 FXQL 语言简介

XQuery 语言作为用户输入的原始查询，书写格式自由，没有严格的位置要求，不容易从用户输入的原始查询中分析出用户的查询意图，因此不能作为查询计划的描述语言，因此需要另一种 XML 查询语言——FXQL 语言——作为查询执行的中间语言。FXQL 的设计目的就是在语义不变的基础上通过把 XQuery 核心表达式转变成 FXQL 表达式，然后交由执行引擎进行处理，进而完成真正的执行动作，精确的实现用户的查询意图。

2.3.1 FXQL 的特点

FXQL 语言的特点是它是一种函数式语言，即以函数调用作为基本的程序结构。整个程序就是一个由各种函数嵌套调用构成的表达式，每个函数从调用它的函数接受值并将新值传递回去，构成嵌套。与命令式语言相比，函数式语言没有赋值语句，变量只能在声明时赋值。因此，FXQL 语言中的变量只起到了值替换的作用，而没有保存程序状态的作用，从而为开发和管理潜在的计算并行性提供了有利的支持。

2.3.2 FXQL 的文法

FXQL 语言的文法如下所示：

(1) r ➔ v(r, … , r)
(2) r ➔ r where v = r, … , v = r,
f(v, … , v) = r, … , f(v, … , v) = r
(3) r ➔ if r then r else r
(4) r ➔ v
(5) r ➔ f
(6) r ➔ c
(7) r ➔ type

产生式(1)为函数调用表达式，包括原语函数调用和自定义函数调用两部分。其中 v 为调用函数名，括号中的若干个 r 为函数调用时传递的参数，它们可以是 FXQL 文法中的任意产生式。

产生式(2)为 where 表达式。其中关键字 where 之前的部分 r 称为主体表达式，where

之后的部分是一个或多个绑定表达式。绑定表达式包括变量绑定表达式和函数绑定表达式两类，其中“v = r”表示变量绑定，等号左面的 v 表示绑定变量名，右面的 r 为被绑定的变量表达式，即把对表达式 r 的求值结果赋给变量 v；“f(v, … v) = r”表示函数绑定，f 为被绑定的函数名，括号中的若干个 v 为函数 f 的参数，等号右面的 r 表示函数体，即把函数体及函数的若干参数作为一个整体绑定到函数名 f 上。

产生式(3)为条件表达式，用于执行筛选。其中关键字 if 后面的 r 表示判断条件；关键字 then 后面的 r 称为 then 子句，表示判断条件成立时执行的表达式；关键字 else 后面的 r 称为 else 子句，表示判断条件不成立时执行的表达式。

产生式(4)为变量引用表达式，v 表示被引用的变量。

产生式(5)为绑定函数引用表达式，f 表示被引用的函数名。

产生式(6)为常数表达式，c 表示常数。常数的类型要符合 XML Schema 的定义，包括各种基本数据类型，如 xs:string、xs:integer 等，还包括 SingleQName 类型，用来表示合法的名字。

产生式(7)为类型描述表达式，用于描述在 FXQL 表达式中出现的特定类型。

2.3.3 FXQL 的原语

XQuery 提供的核心表达式种类远远多于 FXQL 文法的产生式，为了满足 XQuery 的诸多查询功能，FXQL 针对每一种功能设计并实现了相应的原语，因此原语函数体现了 XQuery 语言最基本的、核心的子集。

在原系统的查询执行引擎中，根据功能将原语函数分为如下几类：

- 基本操作：完成过滤、筛选等基本查询操作；
- 数据源操作：完成从指定的数据源获取所需数据或从指定路径获取 XML 文档的操作；
- 序列操作：完成与序列相关的操作，如序列连接、求子序列、求序列项数、判断序列是否为空、求序列倒序等；
- 逻辑操作：完成查询中的逻辑运算，包括：与运算、或运算和非运算；
- 轴操作：完成查询中关于轴的操作，如取结点的指定类型的祖先、祖先-自身、后继、后继-兄弟、前驱、前驱-兄弟等；
- 构造操作：完成各种结点的构造，包括 7 种基本结点的构造和向结点中添加内容等操作；
- 算术操作：完成查询中的各基本数据类型的算术运算；
- 比较操作：完成查询中的值比较（单个值的比较）和一般比较（任意长度的操作数序列的比较）[6]；
- 结点操作：完成查询中的结点比较（比较两个结点本身或文档序）[6]；
- 集合操作：完成对集合的运算，包括求集合的交集、并集和差集；
- 字符串操作：完成关于字符串的操作，如求序列的字符串值、求子串等；
- 元组操作：完成关于元组的操作，如取元组中的序列、向元组中添加序列等；

第 3 章 延迟求值策略的分析与设计

3.1 基于执行机制的延迟求值策略

3.1.1 FXQL 表达式的分析

执行引擎是通过对 FXQL 表达式的求值完成具体的查询操作的，基于执行机制的延迟求值即为对 FXQL 表达式的延迟求值。下面给出对各表达式延迟求值策略的分析。

一、函数调用表达式

函数调用表达式由一个函数名和若干参数组成。不论对哪种具体的函数进行求值，函数调用表达式的积极求值策略均分为两个步骤：首先求出参数列表中各参数的值；第二步求函数体的值并返回求值结果。不同的是，在执行自定义函数求值时则需要将求出的参数值存入子环境，然后在子环境中求函数体的值。

在实际的查询中，对参数的计算有可能为系统带来很大的开销，而函数的某一个具体执行却不一定用到所有的参数。尤其在大规模的查询中，如果一味将所有参数的值都计算出来再计算函数体，将出现大量的不必要的计算过程，造成时间上和空间上的浪费，降低查询的效率。

延迟求值的求值策略解决了这个问题。执行函数调用表达式时，首先保存各个参数的表达式和该参数的求值环境，然后计算函数体。在计算函数体的过程中，随着计算过程的逐步深入将发现一些必须参数值（如条件表达式的比较条件），此时再根据事先保存的表达式和求值环境计算出该参数的结果，并将结果应用于对函数体的计算。可以发现，虽然保存了每个参数的计算信息，但是在函数体中未被使用的参数将不会被计算，由此节省了多余的计算，提高了查询的效率。

二、where 表达式

where 表达式由主体表达式和若干绑定表达式组成。where 表达式的积极求值策略为先对绑定表达式求值，再求主体表达式的值，即：第一步，对于函数绑定“f(v, … ,v) = r”，将函数体及函数参数作为整体绑定到函数名上；对于变量绑定 “v = r”，求出各绑定变量的值。第二步，求主体表达式的值。

在对主体表达式求值的过程中，存在只使用部分绑定表达式的求值结果的情况。因此，将所有绑定表达式的值求出的策略会对造成计算时间上的浪费。

根据上述分析，得到 where 表达式的延迟求值策略：如果是函数绑定，则将参数列表和函数体作为一个整体与函数名绑定；如果是变量绑定，则保存变量的求值表达式和求值环境。在完成各个绑定表达式信息的保存后，求主体表达式的值。

三、条件表达式

条件表达式包含判断条件、then 子句和 else 子句。条件表达式积极求值的策略是：首先求判断条件的值，如果求值结果为真，则求 then 子句的值，否则求 else 子句的值。

执行条件是判断执行 then 子句或 else 子句的依据，因此必须求出结果；then 子句和 else 子句则是表达式的嵌套，其求值策略需要在下一层的计算中体现。因此，条件表达式的延迟求值策略与其积极求值策略是一致的。

四、变量引用表达式、绑定函数引用表达式和常数表达式

变量引用表达式的积极求值策略是：从环境中取变量的值作为求值结果返回。

绑定函数引用表达式的积极求值策略是:从绑定函数引用表达式中取出绑定函数的名称作为求值结果返回。

常数表达式积极求值的策略是：从常数表达式中取常数值作为求值结果返回。

这三种表达式只执行了取值操作，没有经过具体的计算，因此其延迟求值策略与积极求值策略是一致的。

3.1.2 基于表达式的闭包

基于执行机制的延迟求值策略需要暂存表达式及其求值环境，以备在需要时根据存储的表达式和环境求出结果，代入计算。将表达式及其求值环境作为一个整体保存在某一结构中的过程称为绑定，保存的结构称为闭包。

闭包除保存表达式和求值环境外，还需要保存其他相关信息。在查询中，一个表达式的求值结果可以在多处使用，因此创建的闭包也会在多处被引用。当其中一个引用因需要操作数的具体数值而将闭包中存储的表达式计算出结果后，为避免其他引用再次使用该值而引起的重复计算，需要在闭包中保存表达式的求值结果，相应的，还需要保存对该结果是否已经被求出的标记值。

对闭包的操作主要有创建和计算两种。正如前面提到的，在计算中为了实现延迟而创建闭包，那么在发现某个被存入闭包中表达式的值确实需要时，则应该执行对闭包求值，即将闭包中存储的表达式在其求值环境下计算出结果。

闭包是为了替代原表达式的求值结果而建立，并可以作为操作数参与其他的表达式的计算，因此闭包是一种特殊的数值。从数据模型的角度，闭包的上层应该是表示原子值的 Value 接口。

图 3 为基于表达式的闭包的结构，对闭包的说明如下：

- ◆ 类名： Closure
- ◆ 属性：
  - isEvalued:boolean　标记表达式是否已经计算出结果
  - env:Enviorment　记录表达式的求值环境
  - value:Value　标记表达式的计算结果
  - expr:MExpr　记录待求值的表达式
- ◆ 方法：
  - Closure()　构造一个闭包
  - acEval()　对存储的表达式在其计算环境中求值
  - accept()　接受访问者对闭包的访问

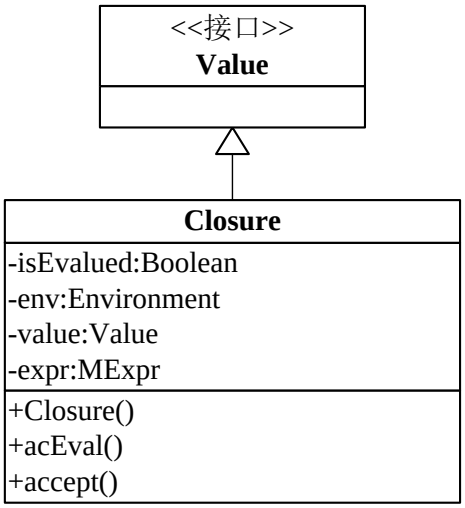


图 3 基于表达式的闭包结构

3.2 基于数据模型的延迟求值策略

3.2.1 FXQL 数据模型

原有面向空间数据集成的 XQuery 查询引擎系统的数据模型与 W3C 的 XQuery 数据模型基本一致。图 4 为原系统数据模型的类体系结构。数据模型的最顶层是接口 Item，表示序列中的项，Item 的下一层是接口 Value 和 Node，分别表示原子值和结点。Value 还可以表示序列，查询的求值结果均用 Value 来表示。有 3 个类实现了 Value 接口，它们是：BaseValue、ArrayTuple 和 RelationTuple，其中 BaseValue 及其子类实现了在查询中出现的所有基本类型，包括简单数据类型和一般数据类型；ArrayTuple 用于在排序是对数据进行分组；RelationTuple 用于分组从数据源获取的数据。

面向空间数据集成的 XQuery 查询引擎系统主要的数据结构为序列。如图 4 所示，类 BaseValue 除 8 个表示简单数据类型（如布尔类型、整型等）的子类外，另外还包括一个表示一般数据类型的 GenericValue 类，它拥有 4 个子类，分别是：SingleItem、ArraySequence、DistinctSequence 和 MultiSequence。其中，SingleItem 表示序列的项，其余 3 个子类用于表示不同组织结构的序列。

设序列 seq = (a_1, a_2, …. , a_n)。在访问 seq 的某项 a_i (i = 1,2, … ,n)时，执行的操作为：首先判断 seq 的第一项 a_1 是否为待访问的项。如果是，按指定的操作完成对该项的访问；如果不是，则判断下一项 a_2 是否为待访问的项。如果是，执行访问；否则继续对下一项进行判断，直至在序列 seq 中找到该项为止，否则返回序列 seq 中不存在待访问的项的消息。由上述分析可知，对序列第 a_i 项的访问只需要遍历序列的前 i 项，而从第 i+1 项开始的其他项不需要遍历。

根据分析结果，可以考虑这样一种序列：序列中的项既可以是真正的值，也可以是保存了能够计算该项值的表达式的闭包。这样的设计可以在构造序列时不必将序列每一项的值都计算出来，而是用闭包代替。在访问序列时只计算需要遍历各项的闭包，由此减少了计算，达到优化效果。

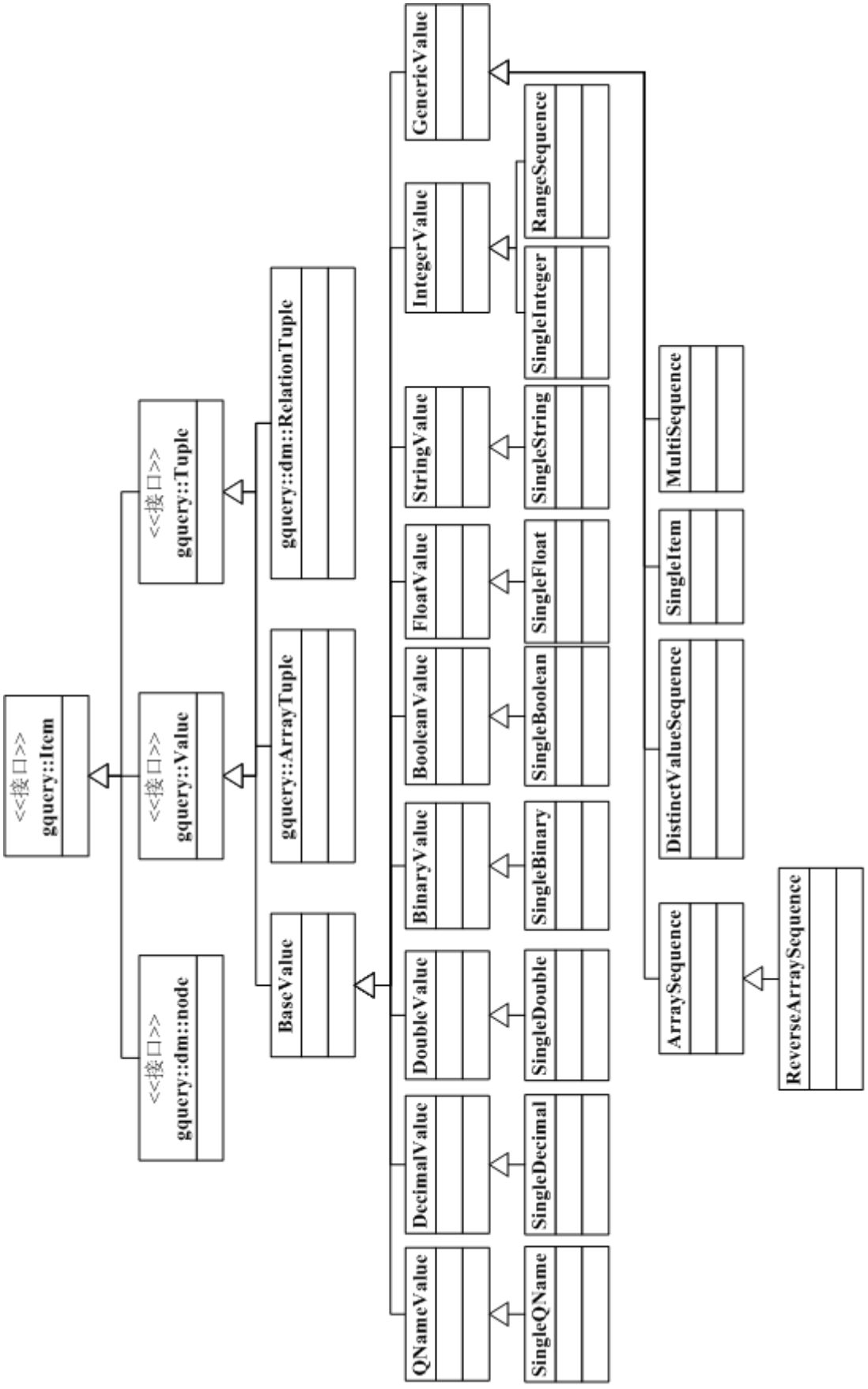


图 4 原系统数据类型的体系结构图

3.2.2 基于数据模型的序列结构

基于数据模型的延迟求值策略需要采用新的数据结构表示序列（在没有特殊说明的情况下均表示含有闭包的序列）。在 FXQL 语言中，序列可以作为表达式的操作数参加运算，因此其上层应该为表示原子值的 Value 接口。任一序列均由头序列和尾序列两部分组成，即序列是由两个子序列连接构成的。这样的结构既可以与原系统中的序列连接原语 concatenate(Value,Value)结合，完成序列构造，又可以通过使用递归方便的访问序列。

图 5 为基于数据模型的序列结构，对其说明如下：

- 类名： SequenceList
- 属性： head:Value　　记录序列的头序列
  tail:Value　　记录序列的尾序列
- 方法： SequentList()　　构造一个序列
  accept()　　接受访问者对序列的访问

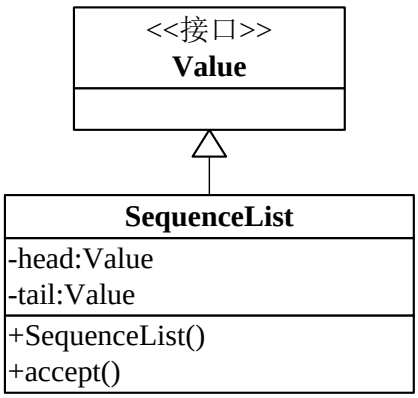


图 5 基于数据模型的序列结构

3.3 查询实例

本节分析一个具体的查询实例，用于比较积极求值和延迟求值两种策略的不同。假设用户发出如下 XQuery 查询请求：

```
let $b := (1,2)
return
        <res>{count($b)}</res>
```

查询定义变量 b 表示一个序列，然后调用 count 函数求序列 b 的项数。该查询经过查询分解转化成 XQuery 核心表达式后，最终被翻译为可以被执行引擎执行的 FXQL 语言：

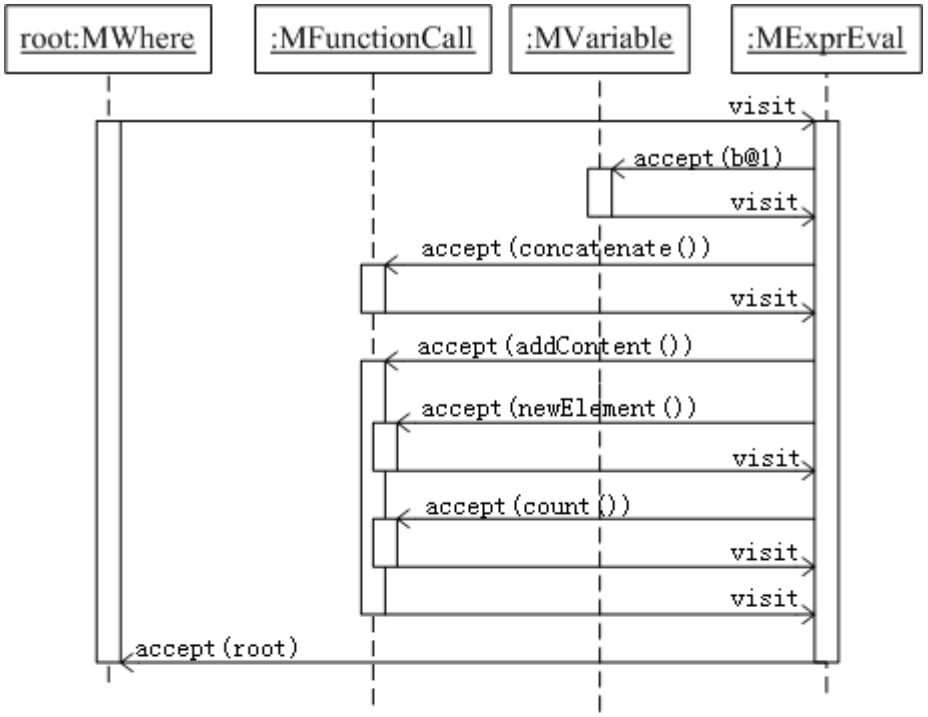
```
ADDCONTENT(
    NEWELEMENT( expanded-QName("", "res") ),
    COUNT( b@1 )
```

```
)
where
    b@1 =
            CONCATENATE( 1, 2 )
```

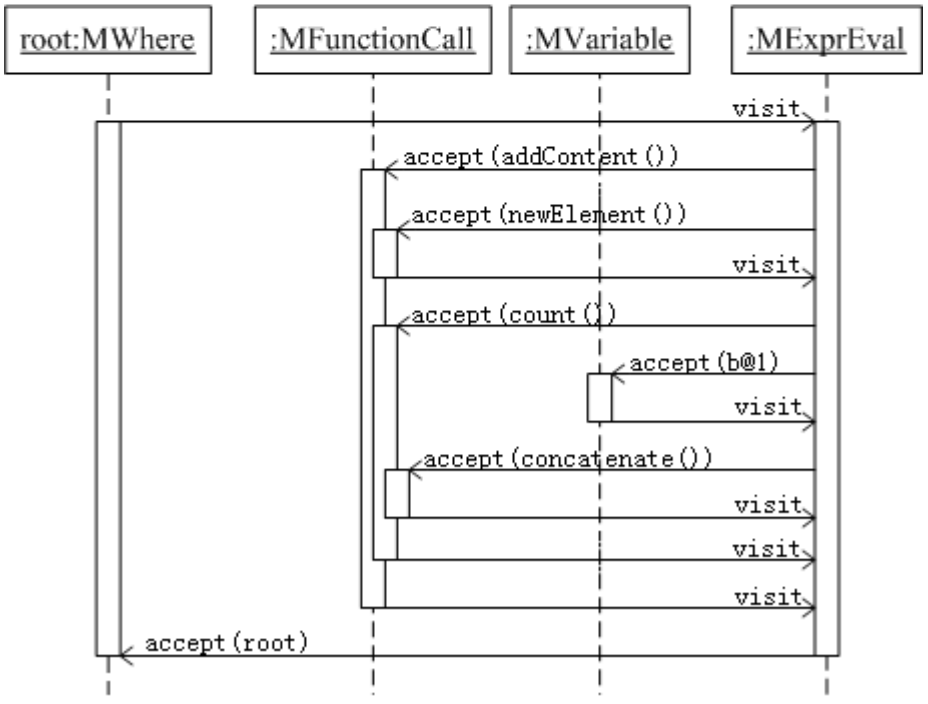
该 FXQL 语言整体上是一个 where 表达式。where 语句的主体表达式为原语函数 ADDCONTENT，该原语的两个参数又分别为原语函数 NEWELEMENT 和 COUNT；where 表达式有一个绑定表达式，它绑定了一个变量 b@1，这个变量为主体表达式中的 COUNT 原语提供参数。

若采用积极求值的策略，查询首先执行 where 语句的绑定表达式，根据绑定变量的求值表达式 CONCATENATE(1, 2)将变量 b@1 的值计算出来；然后执行 where 语句的主体表达式——ADDCONTENT 原语。计算 ADDCONTENT 原语的步骤是：先计算第一个参数，即求原语 NEWELEMENT 的值；然后计算第二个参数，以刚求得的变量 b@1 的值作为实参求原语 COUNT 的值；最后调用 addContent()函数计算 ADDCONTENT 原语，并将其求值结果作为最终查询结果返回。（查询过程如图 6(a)所示）

若采用延迟求值策略，查询首先执行 where 语句的绑定表达式，将绑定变量 b@1 的求值表达式（CONCATENATE 原语）及其求值环境存入闭包 closure；然后执行 where 语句的主体表达式——ADDCONTENT 原语。计算 ADDCONTENT 原语的步骤是：先计算第一个参数，即求原语 NEWELEMENT 的值；然后计算第二个参数，发现需要使用参数 b@1，此时根据闭包中存储的表达式和求值环境将变量 b@1 的具体数值算出，代入到 COUNT 原语进行计算；最后调用 addContent()函数计算 ADDCONTENT 原语，并将其求值结果作为最终查询结果返回。（查询过程如图 6(b)所示）



(a)



(b)

图 6 积极求值策略与延迟求值策略序列图比较

第 4 章 执行引擎的设计与实现

4.1 执行引擎的算法设计

4.1.1 基于表达式的延迟求值算法设计

基于执行机制的延迟求值策略规定了 FXQL 表达式的执行规则，在 FXQL 语言的 7 种表达式中，有 2 种表达式的延迟求值算法与积极求值算法不同，其余 5 种相同。下面给出与积极求值算法不同的表达式执行算法的伪码描述。

(1) 函数调用表达式

本算法实现对函数调用表达式的延迟求值。

执行函数调用表达式时，根据函数的类别来确定具体的执行动作。如果是原语函数，则首先求其参数值，然后用求出的参数值作为参数调用相应的原语函数，该原语函数的返回值即为函数调用表达式的求值结果。如果是绑定函数（自定义函数），则首先创建一个当前执行环境的子环境，第二步将参数和当前求值环境绑定形成闭包，并调用新创建的子环境提供的操作将闭包存储到子环境的参数变量中，然后调用当前环境提供的操作读取函数体，最后在子环境下求函数体的值，该函数体的求值结果即为函数调用表达式的求值结果。

引擎执行函数调用表达式的算法如下所示：

```
visitMFunctionCall(MFunctionCall functionCall)
输入：参数 functionCall 为函数调用表达式
输出：value 为函数调用表达式 functionCall 的求值结果
局部变量：变量 env 为当前的执行环境
算法：
      switch(functionCall.functionID){
          case MFUNCTION_IDS.FIND_FOREACH:        // foreach 原语函数
              求 functionCall 中各参数的值;
              value = 用求出的参数值调用 foreach 语言语函数得到的返回值;
              break;
          ……        // 其它原语函数
          case MFUNCTION_IDS.FNID_BINDING:        // 自定义函数
              在静态上下文中查找函数;
              创建子环境;
              for(每一个函数参数 args[i]){
                  构造参数的闭包;            // 和当前环境绑定
                  将闭包存入子环境的参数变量中;
                  }
```

```
                }
                读取绑定函数的函数体;
                this.pushEnv(subEnv);          // 将当前求值环境设为子环境
                对函数体求值;                   // 在子环境中求值
                this.popEnv(subEnv);           // 恢复当前环境
            }
            break;
        }
    return value;
```

(2) where 表达式

本算法实现对 where 表达式的延迟求值。

执行 where 表达式时，第一步执行绑定表达式：若为变量绑定，则将绑定表达式和当前环境绑定成为闭包；若为函数绑定，则调用当前环境提供的操作将函数体和函数的若干参数作为整体绑定到函数名上。第二步根据绑定表达式的执行结果对主体表达式进行求值，此时得到的求值结果即为 where 表达式的最终求值结果。

引擎执行 where 表达式的算法如下所示：

```
visitMWhere(MWhere where)
输入：参数 where 为 where 表达式
输出：value 为 where 表达式的求值结果
局部变量：变量 env 为当前执行环境
算法：
    for(where 表达式中所有的绑定表达式){
        if(变量绑定){
            构造绑定表达式的闭包 closure;       //和当前环境绑定
            调用 env 提供的操作将求出的 closure 存储到绑定变量中;
        }
        else      // 函数绑定
            调用 env 提供的操作将函数体和若干参数作为整体绑定到函数名上;
    }
    求主体表达式的值;
    return value;
```

4.1.2 基于数据模型的延迟求值算法设计

根据 XQuery 要求的查询功能，原系统共提供 79 个原语函数。下面给出其中部分原语函数延迟求值算法的伪码描述。

(1) foreach()原语

本算法实现迭代输入序列的每一项，并用指定函数对其进行处理。

在执行原语函数 foreach()时，如果输入序列 inSequence 为空，则不进行任何处理；

如果 inSequence 为序列，则递归调用 foreach()函数分别对头序列和尾序列的各项迭代，并将迭代结果连接成为结果序列 resultSequence；如果 inSequence 为闭包，则首先计算出闭包的值，然后调用 foreach()方法迭代闭包的计算结果；如果 inSequence 为各项结果已知的序列（不含闭包），则将序列中各项根据绑定的函数求值，并将求值结果添加到结果序列 resultSequence 中。

resultSequence 即为迭代的结果序列。

引擎执行 foreach()原语函数的延迟求值算法如下所示：

Value foreach(Value inSenquence, QName bindingFnName, MExprEval visitor)

输入：参数 inSequenc 为被迭代的序列

参数 bindingFnName 为绑定函数名

参数 visitor 为求值访问者

输出：处理后得到的结果序列

中间变量：变量 resultSequence 存储迭代的结果序列

算法：

```
生成一个存储迭代结果的序列 resultSequence;
if(inSequence 为空);       // 不执行任何操作
else if(inSequence 为 SequenceList 实体)
    连接头序列和尾序列的迭代结果;
else if(inSequence 为 Closure 实体){
    利用闭包中存储的环境和表达式求出闭包的值;
    resultSequence = foreach(inSequence.value,bindingFnName,visitor);
}
else if(inSequence 为 Value 实体){
    从 visitor.env 中读取绑定函数;
    求 inSequence 中每一项的的值，并将其追加到结果序列中;
}
else
    错误报告;
return resultSequence;
```

(2) quantifiedEvery()原语

本算法实现检查序列中是否所有的项都满足绑定函数定义的条件。

在执行 quantifiedEvery()原语函数时，如果输入序列 inSequence 为空，则说明该项不存在，返回假；如果 inSequence 为序列，则分别判断其头序列和尾序列中的各项是否满足绑定函数定义的条件，当头序列和尾序列中各项均满足条件时，返回真；否则返回假；如果 inSequence 为闭包，则首先计算出闭包的值，然后重新调用 quantifiedEvery()方法对闭包的求值结果进行判断，并返回其判断结果；如果 inSequence 为各项结果已知

的序列（不含闭包），则对序列中的每一项进行判断，如果其中某一项不符合绑定函数定义的条件，返回假；如果序列中所有项均符合判断条件，返回真。

引擎执行 quantifiedEvery()原语函数的延迟求值算法如下所示：

Value quantifiedEvery(Value inSequence,QName bindingFnName,MExprEval visitor)

输入：参数 inSequence 为待检查的序列；
参数 bindingFnName 为绑定函数名；
参数 visitor 为求值访问者；

输出：如果 inSequence 中所有项都满足名为 bindingFnName 的绑定函数定义的条件，则返回真；否则，返回假；

算法：

```
if(inSequence 为空)
    return FALSE;
else if(inSequence 为 SequenceList 实体){
    if(头序列中含不满足条件的项)
        return FALSE;
    else if(尾序列中含不满足条件的项)
        return FALSE;
    else            // 头、尾序列中各项都满足判断条件
        return TRUE;
}
else if(inSequence 为 Closure 实体){   // 对闭包的求值结果判断
    利用闭包中存储的环境和表达式求出闭包的值;
    return quantifiedEvery(inSequence.value, bindingFnName, visitor);
}
else if(inSequence 为 Value 实体){
    从 visitor.env 中读出绑定函数;
    while(inSequence 中的每一项 temp){
        在环境 visitor.env 中存储绑定变量 temp 的值;
        求绑定函数的值;
        if(求得的值为假)
            return FALSE;
    }
    return TRUE;
}
else
    错误报告;
```

(3) concatenate()原语

本算法实现两个序列的连接。在执行原语函数 concatenate()时，根据函数的两个参数生成一个序列实体并返回。

引擎执行 concatenate()原语函数的延迟求值算法如下所示：

SequenceList concatenate(Value arg1, Value arg2)

输入：　参数 arg1、arg2 为被连接的两个序列

输出：　连接得到的结果序列

算法：

```
return new SequenceList(arg1, arg2);
```

(4) count()原语

本算法实现求序列中项的数目。

在执行原语函数 count()时，如果输入序列 inSequence 为空，序列项数计数器 result 值为 0；如果 inSequence 为序列，则分别递归调用 count()函数计算头序列和尾序列的项数，并累加两次递归的返回值；如果 inSequence 为闭包，则首先计算出闭包的值，然后调用 count()方法求闭包的值的项数；如果 inSequence 为各项结果已知的序列（不含闭包的序列），则通过对其各项的遍历累加 result 值。序列项数计数器 result 的值即为输入序列的项数。

引擎执行 count()原语函数的延迟求值算法如下所示：

SingleInteger count(SequenceList inSequence)

输入：　参数 seq 为序列

输出：　seq 中项的数量

中间变量：result 为序列项数计数器，初值为 0

算法：

```
result = 0;                    // 计数器置 0
if(inSequence 为空);           // 不做任何操作
else if(inSequence 为 SequenceList 实体){
    result =  头序列项数;
    result +=  尾序列项数;
}
else if(inSequence 为 Closure 实体){
    利用闭包中存储的环境和表达式求出闭包的值;
    result = count(inSequence.value);
}
else if(inSequence 为 Value 实体)
    for(inSequence 的每一项)          // 遍历序列各项
        result++;
else
    错误报告;
```

return result;　　// 返回序列项数

(5) subsequence()原语

本算法实现求子序列。

在执行原语函数 subsequence()时，如果 inSequence 为序列，则首先判断子序列在输入序列中的位置：若子序列位于头序列（或尾序列）中，则递归调用 subsequence()函数在头序列（或尾序列）中求子序列；若子序列的一部分在头序列中，另一部分在尾序列中，则分别计算两部分的起始位置和长度，并递归调用 subsequence()函数分别在头序列和尾序列中求出子序列的两部分，利用序列连接函数将两部分连接成子序列；

如果 inSequence 为闭包，则首先计算出闭包的值，然后递归调用 subsequence()方法求子序列；

如果 inSequence 为各项结果已知的序列（不含闭包），则计算子序列的终止位置，并利用 FilterExpr 类提供的方法计算子序列。

中间变量 resultSequence 即为求得的子序列。

引擎执行 subsequence()函数的延迟求值算法如下所示：

Value subsequence(Value inSequence, BaseValue start, BaseValue length)

输入：参数 inSequence 为原始序列

参数 start 为子序列的起始位置（从 1 开始计算）

参数 length 为子序列的长度

输出：inSequence 中从 start 开始，长度为 length 的子序列

中间变量：变量 resultSequence 用于存储子序列

算法：

```
生成子序列 resultSequence;
根据 start 和 length 参数计算出子序列的终止位置 end;
if(inSequence 为 SequenceList 类实体){
    if(子序列在 head 中)
        在 inSequence.head 中求子序列;
    else if(子序列在 tail 中)
        在 inSequence.tail 中求子序列;
    else{        // 子序列在 head 和 tail 中各包含一部分
        分别在 inSequene.head 和 inSequence.tail 中求子序列的两部分;
        连接子序列的两部分;
    }
}
else if(inSequence 为 Closure 类实体){
    根据 Closure 实体提供的表达式和求值环境计算出闭包的值;
    resultSequence = subsequence(inSequence.value,start,length);
}
```

```
else if(inSequence 为 Value 类实体){
    resultSequence = new FilterExpr.Slice(inSequence,start,end);
}
else
    错误报告;
return resultSequence;
```

(6) and()原语

本算法实现原语函数与操作的延迟求值。

在执行原语函数 and()时，首先对第一个操作数 arg 进行判断，若 arg 为假，原语函数返回值为假；否则根据第二个操作数的求值表达式和求值访问者求出其结果 value，并对 value 进行判断，若 value 为假，返回值为假；否则返回值为真。

引擎执行 and()原语函数的延迟求值算法如下所示：

SingleBoolean and(SingleBoolean arg, MExpr expr, MExprEval visitor)

输入：参数 arg 为第一个与操作数

参数 expr 为第二个与操作数的求值表达式

参数 visitor 为第二个与操作数的求值访问者

输出：与操作结果

算法：

```
if(arg == FALSE)
    return FALSE;
else{
    value =  表达式 expr 在其求值访问者提供的环境中的求值结果;
    if(value == FALSE)
        return FALSE;
    else
        return TRUE;
}
```

4.2 Visitor 模式及其在执行引擎中的应用

Visitor（访问者）模式表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素的类的前提下定义作用与这些元素的新操作[7]。Visitor 模式适用于数据结构相对稳定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由地演化[8]。Visitor 具有以下优点：

- 易于增加新的操作。在 Visitor 模式下，增加新的操作只需要增加一个新的访问者类。
- 易于修改和维护。在 Visitor 模式下，有关的各个行为集中在一个访问者对象中，而不是分布在各个结点中。

Visitor 模式的上述特点与对 FXQL 各表达式操作的定义完全符合，所以使用 Visitor 模式来实现查询执行引擎。

4.2.1 Visitor 模式实现表达式访问者

FXQL 表达式类体系结构如图 7 所示。类 MExpr 为所有 FXQL 表达式的抽象父类，它定义了抽象方法 accept()用于接受访问者的访问。子类 MWhere、MBindingFnRef、MVariable、MType、MFunctionCall、MConstant 和 MCondition 是由 FXQL 文法定义的 7 种表达式，另外还定义了子类 MBinding 专用于表示 where 表达式中的绑定表达式，它们除实现父类 MExpr 定义的抽象方法外，还根据自身的需要定义了自己的成员和方法。为了实现对各表达式的访问，定义接口 IMExprVisitor，它为每一种 FXQL 表达式声明了访问方法，并在 3 个具体访问者类 MExprEval、MExprEvalForDemo 和 MExprToString 中实现（表达式访问者结构如图 8 所示）。

MExprEval 用于实现对 FXQL 表达式求值，其成员包括：用于表示求值环境的 Environment 类实体 env；用于表示求值结果的 Value 类实体 value 和用于表示闭包的 Closure 类实体 closure。MExprEval 中继承了接口 IMExprVisitor 定义的对每一种 FXQL 表达式的求值方法。

MExprEvalForDemo 除用于实现对 FXQL 表达式的求值功能，还在求值的过程中输出求值步骤。MExprEvalForDemo 类除定义用于实现表达式求值的成员和方法外，还定义了用于控制缩进格式的整型标记值及其打印方法。因为 MExprEvalForDemo 类只是在 MExprEval 类基础上添加了输出功能，所以，这个访问者只在有打印需求的时候才被调用。

MExprToString 类是原系统中已经实现的访问者，它用于获取 FXQL 表达式的文本表示。

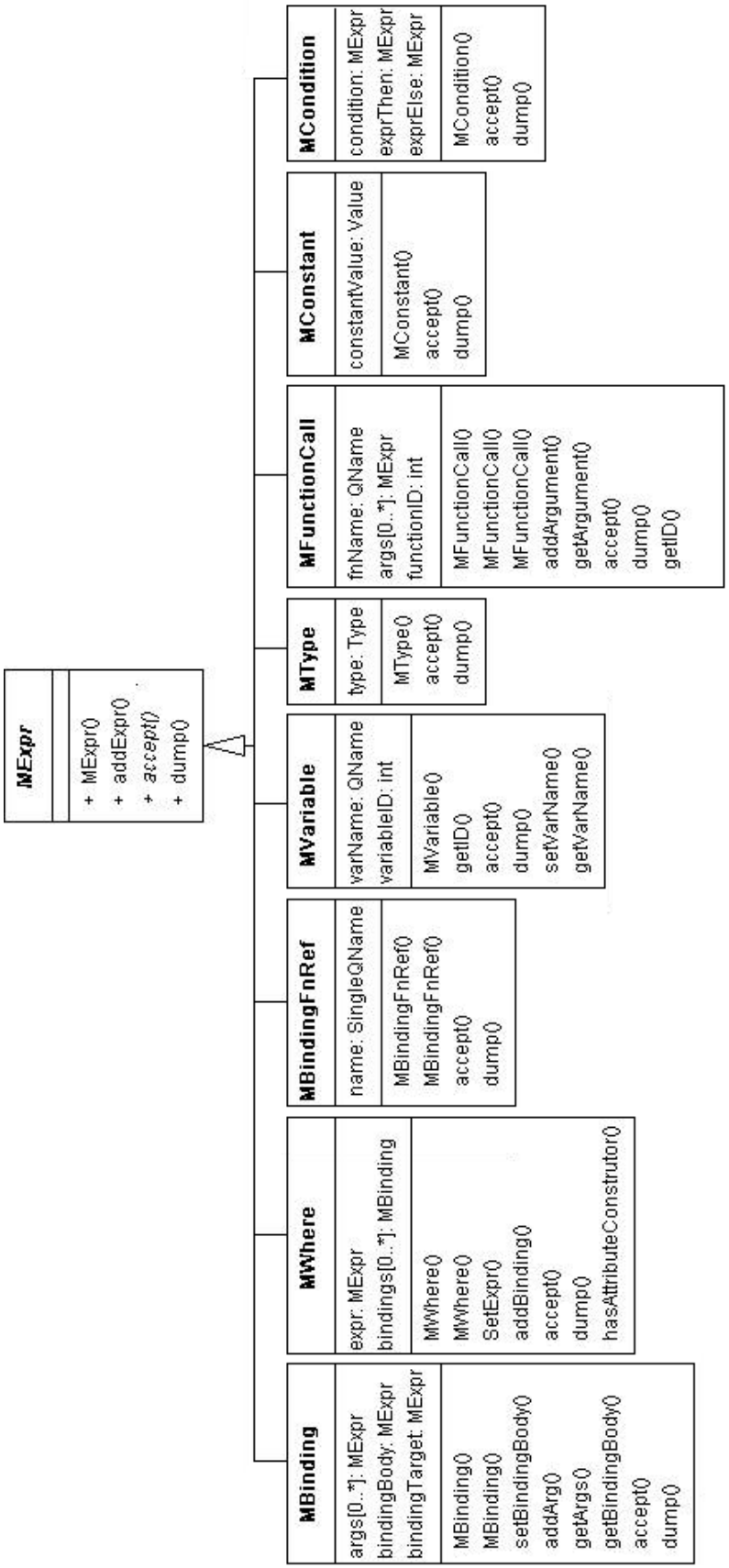


图 7 FXQL 表达式的类体系结构

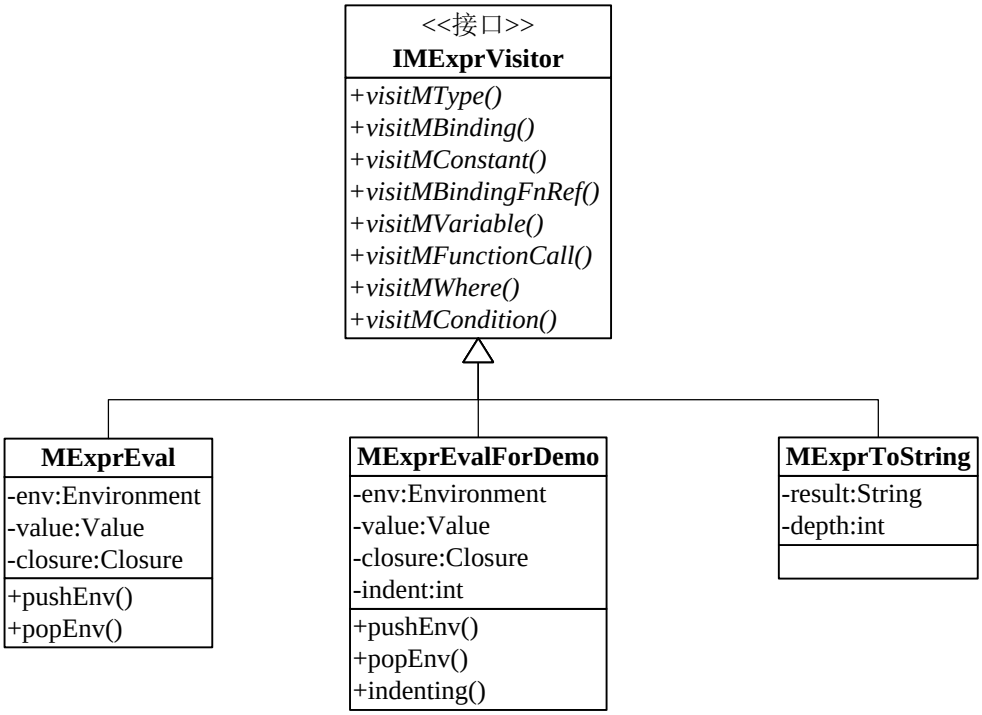


图 8 FXQL 表达式访问者结构

4.2.2 Visitor 模式实现 Value 访问者

表示查询执行引擎中所有数据类型的 Value 类体系如图 9 所示。类 BaseValue 是接口 Value 的一个具体实现，表示系统中的基本数据类型。其中，Closure 和 SequenceList 是为延迟求值策略设计的闭包和序列结构；QNameValue 表示合法的限定名（qualified name, QName）；DecimalValue、DoubleValue、BinaryValue、BooleanValue、FloatValue、StringValue 和 IntegerValue 表示各种简单数据类型；GenericValue 表示一般数据类型。与原系统相比（如图 4 所示），利用延迟求值策略的查询引擎系统的数据模型只增加了 Closure 类和 SequenceList 类。

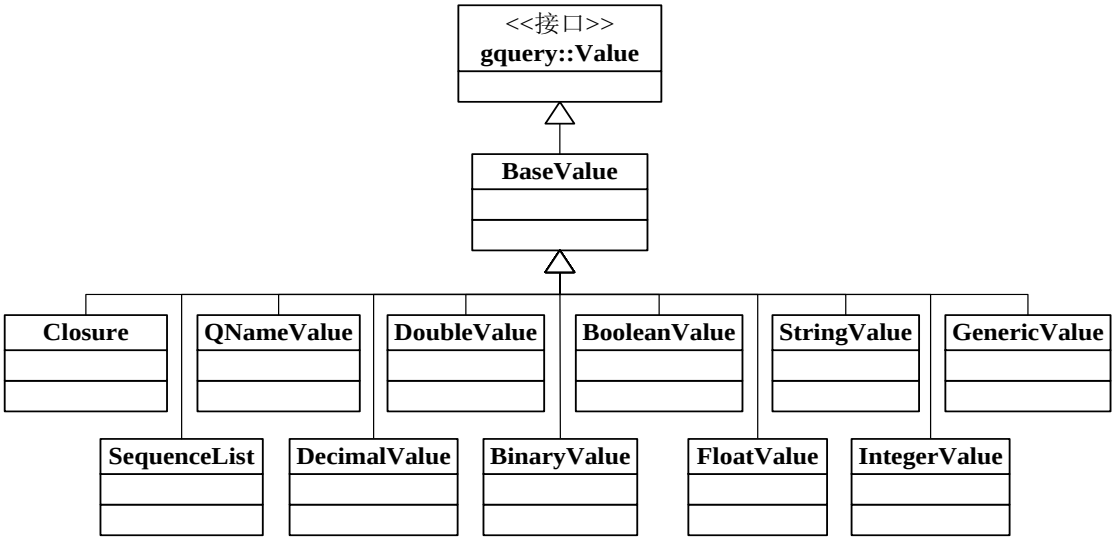


图 9 Value 类体系结构

因为采用了延迟求值策略，所以查询结果会出现 Closure 类或者 SequenceList 类的实体，这在要求使用具体数值（如最终查询结果或操作数必须为数值）的情况下需要用积极求值的方法进行计算。因此需要使用 Visitor 模式实现对不同的 Value 类实体的计算操作。

Value 访问者如图 10 所示。在 Value 类体系中，Closure 类数值和 SequenceList 类数值是可以被计算的，其余类型均不能被计算，只能用于访问。因此，接口 IValueVisitor 声明了 Closure、SequenceList 和其他数值的访问方法。CValueVisitor 类是具体访问者，它继承接口 IValueVisitor，具体实现了对数据类型的计算。

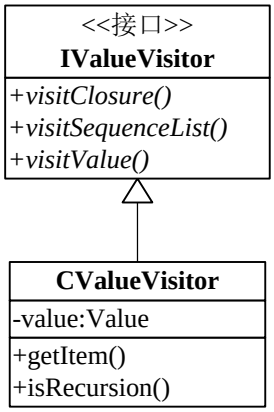


图 10 Value 访问者结构

第 5 章 优化效果分析

5.1 优化效果

5.1.1 理想效果

FXQL 是一种函数式语言，其中的任意一个函数都有可能嵌套多层。在实际查询中，一方面需要对数据库中的数据进行筛选，另一方面，查询的数据量很大。当语法树中的某个结点因不符合筛选条件而不能参加下面的运算时，在延迟求值的策略下，这个结点的所有子孙都不会被计算求值，由此查询性能将得到大幅度提高。在理想的情况下，因为大量计算被延迟，查询执行时间将会缩短，查询效率会得到明显的提高，是数据查询系统得到显著的优化效果。

5.1.2 实际效果

在测试中，共使用 48 个测试用例，其中包括 16 个自定义用例和 32 个 W3C 测试用例。测试通过控制台输出将优化后的查询执行时间与未经优化的系统作比较，用多次测量取平均值的方法获得用例的平均查询执行时间。

设 $i \in [1,7]$，T_{1i} 为测试用例在未优化的系统中的查询执行时间，T_{2i} 为用例在优化后系统中的查询执行时间。则：

$$T_1 = (T_{11} + T_{12} + T_{13} + T_{14} + T_{15} + T_{16} + T_{17})/7;$$

$$T_2 = (T_{21} + T_{22} + T_{23} + T_{24} + T_{25} + T_{26} + T_{27})/7;$$

$$\text{优化比率 } r = (T_2 - T_1)/ T_1 * 100\%;$$

优化比率 r 的值越大，说明测试用例在优化后的查询时间越短，优化效果越明显。若 r 为负数，则说明查询效率降低，查询没有得到优化。各用例的优化效果如图 11、图 12、图 13 所示，其中图 11 显示了自定义测试用例的优化效果；图 12 和图 13 显示了 W3C 提供的测试用例的优化效果。

从优化效果图可以看出，在优化的查询执行引擎中，部分用例的查询效率得到提高，优化比率最高达到 77%，平均提高 16.41%；另一部分用例的执行速度与原系统相比略有下降，平均下降幅度为 6%。

由此可见，优化后的系统只对部分测试用例起到了优化作用，某些用例的查询效率甚至有所降低，因此没有达到理想的优化效果。

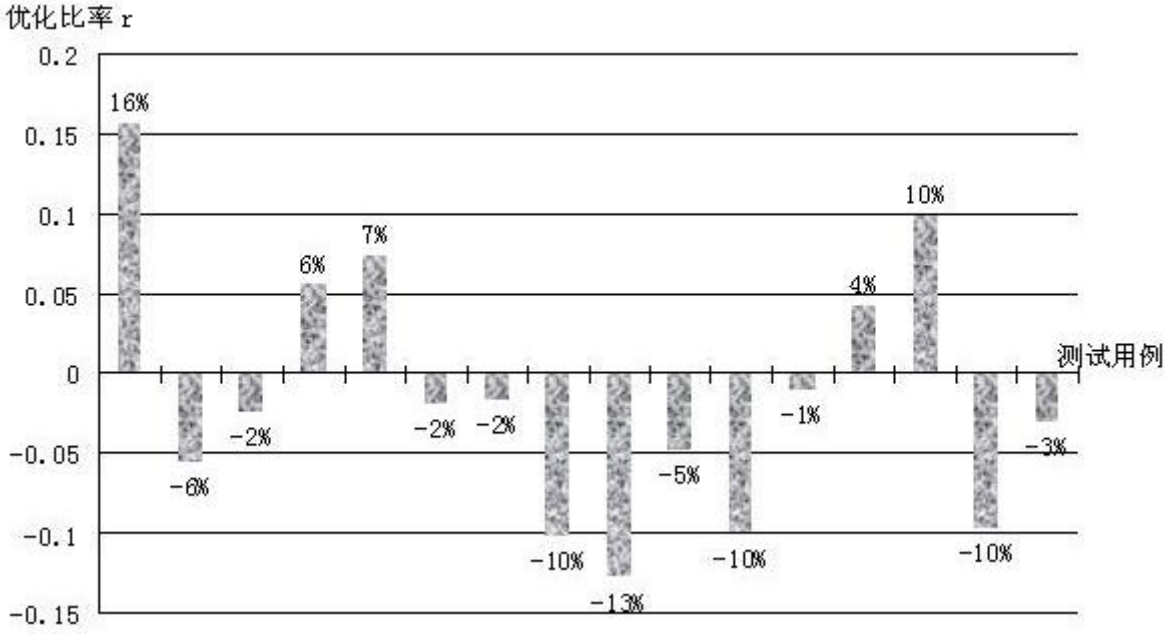


图 11 自定义测试用例优化效果

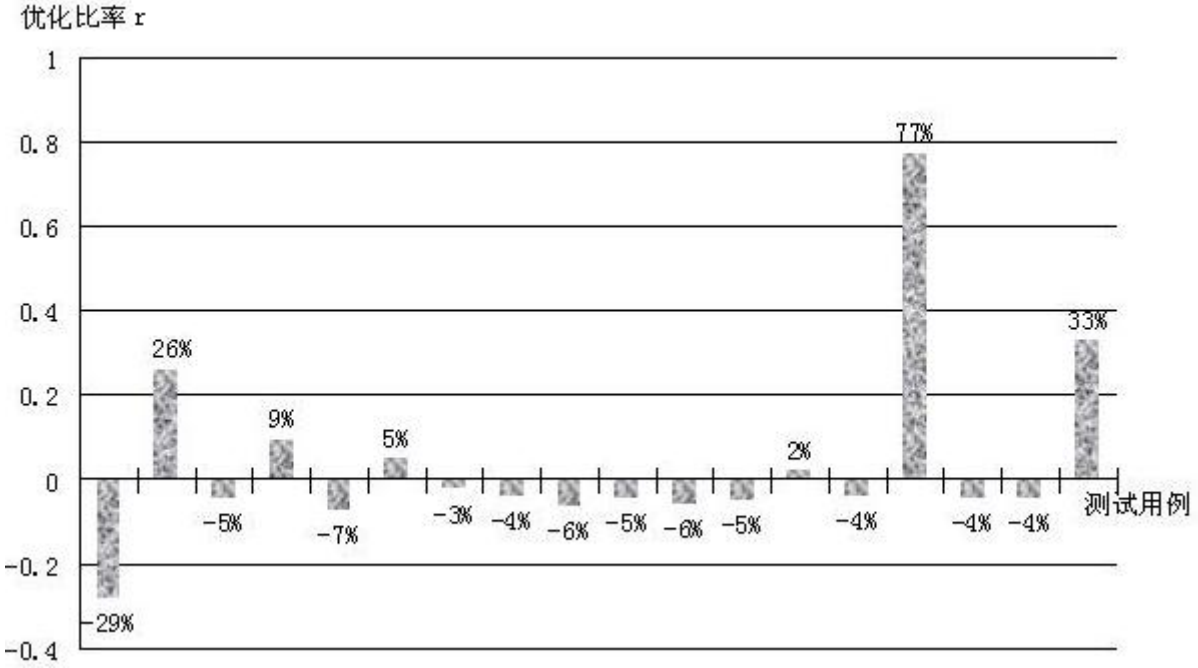


图 12 W3C 测试用例优化效果-1

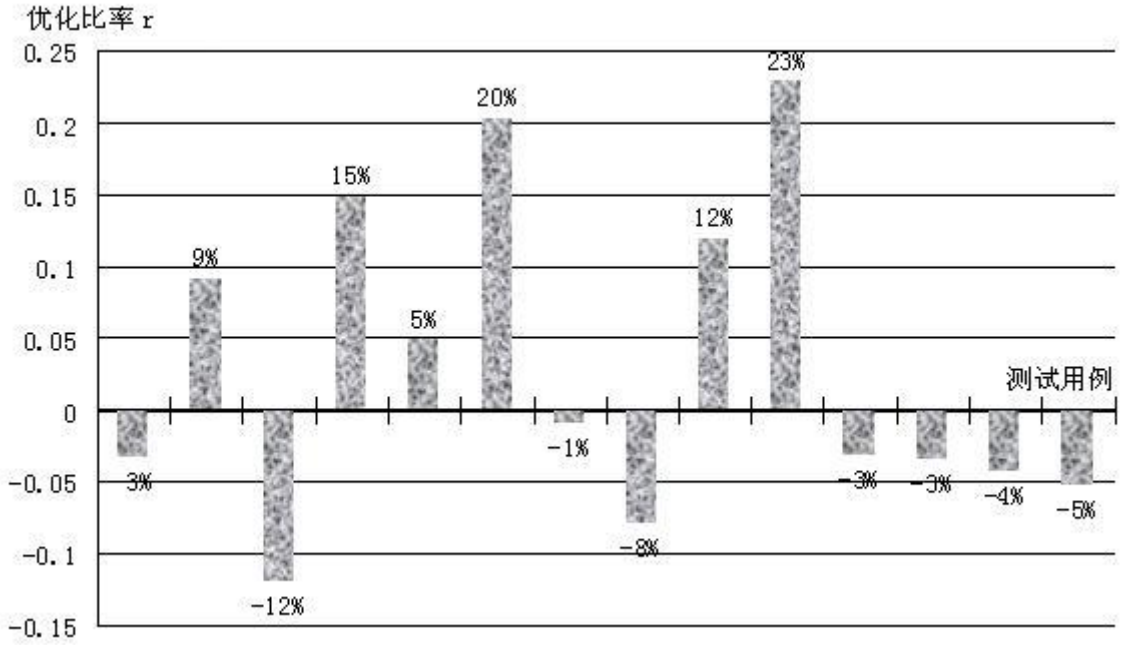


图 13 W3C 测试用例优化效果-2

5.2 优化效果分析

实际优化效果与理想效果存在差距的原因如下：

(1) 基于数据模型的优化力度小

面向数据集成的 XQuery 查询执行引擎是一个可以查询多种数据源类型，支持分布式空间数据集成和共享的系统，其查询功能多种多样，FXQL 语言设计了 79 个原语以满足所有的查询需求。采用延迟求值策略的查询引擎中添加了新的数据结构用于表示闭包和含有闭包的序列，要想充分达到延迟效果，一方面需要重写一套既符合允许闭包作为序列项的序列结构，又支持 XQuery 各种查询功能的原语；另一方面需要完成对 Value 类系统中各种数据类型的数值的访问。因为实现时间短，所以在基于数据模型的延迟求值部分，只重写了部分有代表性的原语，如序列的连接、求序列项数、求子序列等等。因此，如果重写所有的原语，将会达到更好的优化效果。

(2) 测试用例复杂度低

延迟求值的策略是一柄双刃剑，在避免多余的计算的同时，需要事先保存表达式和求值环境。如果创建的闭包数目很多，将会占用大量存储空间，产生负面作用。在大规模的查询中，创建闭包对系统产生的影响将远远小于表达式计算造成的系统消耗，因此可以起到提高查询效率的作用。但是在测试中，测试用例只是简单的查询，复杂度小，由闭包引起的系统消耗就会明显的表现出来，因此不能表现出延迟求值策略带来的优化效果。

5.3 其他方面的优化

对面向数据集成的 XQuery 查询执行引擎的优化除采取延迟求值的策略避免多余计算过程外，还可以考虑其他方面的优化。

1. 基于流水线的 XQuery 查询优化

   系统在接受了一个 XQuery 查询后，需要先后经过语法分析，查询分解，规范化和静态类型检查等步骤，最后形成 FXQL 语言，整个过程是在语法树上完成的。在查询分解的步骤中，可以通过适当调整语法树，在保证查询内容不变的前提下改变查询执行时的数据流的方法提高查询效率。

   假设一个查询需要两个数据源 A 和 B，其中数据源 A 中只有某些数据参与查询，那么将会有两种查询方案：一种方案是先对两个数据源中的数据做连接，然后在连接结果中筛选；另一种方案是先筛选数据源 A 得到参与实际查询的部分数据，然后将筛选结果和数据源 B 连接。因为连接操作的计算量是两部分数据的笛卡尔乘积，所以先执行筛选将其中的一部分数据缩小后再执行连接会大大提高其计算效率，进而提高整个查询的效率。

2. 基于冗余的 XQuery 查询优化

   基于冗余的查询优化分为两个方面。其一为提取重复计算的公共表达式，即声明一个变量用于表示被重复计算的表达式，而后面所有使用到这个表达式的地方全部用声明的变量表示。这样可以使公共表达式只经过一次计算，从而减少重复的计算。其二为循环不变量外提，即将在循环中计算的不变量提取到循环的外面进行计算。这样该不变量只会被计算一次，从而避免了在循环中的重复计算。

结　论

本文研究了面向查询优化的 XQuery 查询引擎系统。在原有基于 Mediator 体系结构的 XQuery 查询引擎系统的基础上，采用延迟求值的求值策略对其执行机制进行优化。

本文针对执行查询计划的过程中出现的不必要的计算，设计了可以存储表达式和求值环境的闭包结构。闭包保存表达式的求值信息而不是对表达式进行求值计算，还可以替代表达式的求值结果，作为操作数进入下一步的运算。只有在运算中确实需要实际数值的时候，才计算闭包中保存的表达式的值，因此闭包起到了延迟求值的作用，避免了多余的运算，从而在执行机制的层次上达到了优化的效果。

本文针对 XQuery 数据模型，并结合延迟求值的思想设计了一种新的存储序列的结构，它使序列中地位原本相同的各项有了先后之分。这样的结构使对序列采用延迟求值的求值策略成为可能，从而在数据结构的层次上达到了优化的效果。

虽然本课题取得了一定的研究成果，但是系统中难免存在一些问题有待完善和进一步研究：第一，对系统延迟求值的优化不充分。本文完成了基于执行机制的延迟求值策略的设计与实现，但是基于数据模型的延迟求值还没有彻底实现；第二，优化切入点过于单一。本文只从求值策略的角度实现了对查询执行引擎的优化，另外还可以考虑基于流水线和基于冗余的优化方法。

参考文献

1. 刘丽.什么是 GIS.(2004-12-3).
   http://www.gisforum.net/show.aspx?id=301&cid=14
2. 地理信息系统论坛.GIS 的应用领域有哪些.(2005-2-20).
   http://www.gisforum.net/show.aspx?id=887&cid=14
3. 裘宗燕.程序设计语言原理.2005 年.
   http://www.is.pku.edu.cn/~qzy/plan/slides/plan05.pdf
4. Howard Katz Editor, Don Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, Philip Wadler. XQuery from the experts.Addison Wesley.2003:81-86
5. W3C.XQuery1.0:an XML Query Language,2004.Available via the WWW as http://www.w3.org/TR/xquery
6. Darshan Singh.Essential XQuery - The XML Query Language.2004-2-2.
   http://www.yukonxml.com/articles/xquery/?print=1
7. Erich Gamma,Richard Helm,Ralph Johnson,John Vlissides.设计模式 可复用面向对象软件的基础.李英军,马晓星,蔡敏,刘建中 等.北京:机械工业出版社.2000:218-228
8. 阎宏.Java 与模式.北京:电子工业出版社.2002:947-956
9. W3C.XML Query Use Cases,2003.Available via the WWW as http://www.w3.org/TR/xpath-functions/

附录一 执行引擎算法

优化的系统同时支持积极求值和延迟求值两种求值策略。下面给出执行引擎采用延迟求值策略执行查询计划的具体算法，共计 13 个，包括 2 个表达式延迟求值算法和 11 个原语函数延迟求值算法。

一、 函数调用表达式

visitMFunctionCall(MFunctionCall functionCall)

功能：求函数调用表达式的值

输入：functionCall 为函数调用表达式

输出：value 为函数调用表达式 functionCall 的求值结果

局部变量：变量 env 为当前的执行环境

算法：

```
switch(functionCall.functionID){
    case MFUNCTION_IDS.FIND_FOREACH:        // foreach 原语函数
        求 functionCall 中各参数的值;
        value =  用求出的参数值调用 foreach 语言语函数得到的返回值;
        break;
    ……          // 其它原语函数
    case MFUNCTION_IDS.FNID_BINDING:        // 自定义函数
        在静态上下文中查找函数;
        创建一个子环境 subEnv;
        for(每一个函数参数 args[i]){
            构造 args[i]的闭包 closure;      // 和当前环境绑定
            将 closure 存入 subEnv 的参数变量中;
            }
        }
        读取 FunctionCall 中指定的绑定函数;
        this.pushEnv(subEnv);       // 将当前求值环境设为子环境
        对函数体求值;               // 在子环境中求值
        this.popEnv(subEnv);        // 恢复当前环境
    }
    break;
  }
return value;
```

二、 where 表达式

visitMWhere(MWhere where)

功能：求 where 表达式的值
输入：参数 where 为 where 表达式
输出：value 为 where 表达式的求值结果
局部变量：env 为当前的执行环境
算法：

```
for(where 表达式中所有的绑定表达式){
    if(变量绑定){
        构造绑定表达式的闭包 closure;        //和当前环境绑定
        调用 env 提供的操作将求出的 closure 存储到绑定变量中;
    }
    else      // 函数绑定
        调用 env 提供的操作将函数体和若干参数作为整体绑定到函数名上;
}
求主体表达式的值;
return value;
```

三、 原语函数——concatenate()

1．两个参数

SequenceList concatenate(Value arg1, Value arg2)
功能： 连接两个序列
输入： 参数 arg1、arg2 为被连接的两个序列
输出： 连接得到的结果序列
算法：

```
return new SequenceList(arg1, arg2);
```

2．一个参数

SequenceList concatenate(Value arg1)
功能： 构造只含有一项的序列
输入： 序列中的项
输出： 构造得到的序列
算法：

```
return new SequenceList(arg1, null);
```

四、 原语函数——count()

SingleInteger count(Value inSequence)
功能： 求序列树中项的数量
输入： 参数 inSequence 为序列树
输出： inSequence 中项的数量
中间变量：result 为序列项数计数器，初值为 0
算法：

```
result = 0;
if(inSequence 为空);
else if(inSequence 为 SequenceList 实体){
    result = count(inSequence.head);        // 求头序列项数
    result += count(inSequence.tail);       // 求尾序列项数并累加
}
else if(inSequence 为 Closure 实体){
    利用闭包中存储的环境和表达式求出闭包的值;
    result = count(inSequence.value);
}
else if(inSequence 为 Value 实体)
    for(inSequence 的每一项)
        result++;
else
    错误报告;
return result;
```

五、 原语函数——subsequence()

Value subsequence(Value inSequence, BaseValue start, BaseValue length)

功能：求子序列

输入：参数 inSequence 为原始序列

参数 start 为子序列的起始位置（从 1 开始计算）

参数 length 为子序列的长度

输出：inSequence 中从 start 开始，长度为 length 的子序列

中间变量：resultSequence 用于存储子序列

算法：

```
生成子序列 resultSequence;
根据 start 和 length 参数计算出子序列的终止位置 end;
if(inSequence 为 SequenceList 类实体){
    len_head = countForSub(inSequence.head);// 计算 head 长度
    if(start<len_head){         // 子序列在 head 中开始
        if(end<len_head)        // 子序列在 head 中结尾，在 head 中求子序列
            resultSequence = subsequence(inSequence.head,start,length);
        else                    // 子序列在 tail 中结尾
            resultSequence = concatenate(       // 分别计算子序列并连接
                    subsequence(inSequence.head,start,len_head-start),
                    subsequence(inSequence.tail,1,length-(len_head-start)));
    }else{                      // 子序列在 tail 中开始
```

```
                start = start – len_head;    // 将起始位置更新为在 tail 序列中的位置
                resultSequence = (inSequence.tail,start,length); // 在 tail 中求子序列
            }
        }
        else if(inSequence 为 Closure 类实体){
            根据 Closure 实体提供的表达式和求值环境计算出闭包的值;
            resultSequence = subsequence(inSequence.value,start,length);
        }
        else if(inSequence 为 Value 类实体){
            resultSequence = new FilterExpr.Slice(inSequence,start,end);
        }
        else
            错误报告;
        return resultSequence;
```

六、 原语函数——foreach()

Value foreach(Value inSenquence, QName bindingFnName, MExprEval visitor)

功能：在序列 inSequence 中进行迭代，用名为 bindingFnName 的绑定函数进行处理。

输入：参数 inSequenc 为被迭代序列的序列

参数 bindingFnName 为绑定函数名

参数 visitor 为求值访问者

输出：处理后得到的结果序列

中间变量：resultSequence 为存储迭代结果的序列

算法：

```
        生成一个存储迭代结果的序列 resultSequence;
        if(inSequence 为空);       // 不执行任何操作
        else if(inSequence 为 SequenceList 实体)
            resultSequence = concatenate(          // 分别迭代 head 和 tail 并连接结果
                                foreach(inSequence.head,bindingFnName,visitor),
                                foreach(inSequence.tail,bindingFnName,visitor));
        else if(inSequence 为 Closure 实体){
            利用闭包中存储的环境和表达式求出闭包的值;
            resultSequence = foreach(inSequence.value,bindingFnName,visitor);
        }
        else if(inSequence 为 Value 实体){
            从 visitor.env 中读取绑定函数;
            while(inSequence 中的每一项 temp){
```

向环境 visitor.env 中存储绑定变量 temp 的值;
绑定函数求值;
将求得的值追加到结果序列中;
}
}
else
错误报告;
return resultSequence;

七、 原语函数——foreachAt()

Value foreach(Value inSenquence, QName bindingFnName, MExprEval visitor)

功能：在序列 inSequence 中进行迭代，用名为 bindingFnName 的绑定函数进行处理，并记录当前处理项在序列中的位置。

输入：参数 inSequenc 为被迭代序列的序列
参数 bindingFnName 为绑定函数名
参数 visitor 为求值访问者

输出：处理后得到的结果序列

中间变量：resultSequence 为存储迭代结果的序列
i 为循环计数器，初值为 0

算法：

生成一个存储迭代结果的序列 resultSequence;
if(inSequence 为空); // 不执行任何操作
else if(inSequence 为 SequenceList 实体)
resultSequence = concatenate(// 分别迭代 head 和 tail 并连接结果
foreach(inSequence.head,bindingFnName,visitor),
foreach(inSequence.tail,bindingFnName,visitor));
else if(inSequence 为 Closure 实体){
利用闭包中存储的环境和表达式求出闭包的值;
resultSequence = foreach(inSequence.value,bindingFnName,visitor);
}
else if(inSequence 为 Value 实体){
从 visitor.env 中读取绑定函数;
while(inSequence 中的每一项 temp){
循环计数器 i 递增;
向环境 visitor.env 中存储绑定变量 temp 的值;
绑定函数求值;
将求得的值追加到结果序列中;
}

```
        }
        else
            错误报告;
        return resultSequence;
```

八、 原语函数——filter()

Value filter(Value inSequence, QName bindingFnName, MExprEval visitor)

功能： 用名为 bindingFnName 的绑定函数过滤序列 inSequence

输入： 参数 inSequence 为带过滤的序列

参数 bindingFnName 为绑定函数名

参数 visitor 为求值访问者

中间变量：变量 resultSequence 为存储过滤结果的序列

算法：

```
        生成一个存储过滤结果的序列 resultSequence;
        if(inSequence 为空);       // 不执行任何操作
        else if(inSequence 为 SequenceList 实体)
            resultSequence = concatenate(      //分别过滤 head 和 tail 并连接过滤结果
                        filter(inSequence.head,bindingFnName,visitor),
                        filter(inSequence.tail,bindingFnName,visitor));
        else if(inSequence 为 Closure 实体){
            利用闭包中存储的环境和表达式求出闭包的值;
            resultSequence = filter(inSequence.value,bindingFnName,visitor);
        }
        else if(inSequence 为 Value 实体){
            从 visitor.env 中读取绑定函数;
            while(inSequence 中的每一项 temp){
                向环境 visitor.env 中存储绑定变量 temp 的值;
                绑定函数求值;
                if(求得的结果符合过滤要求)
                    resultSequence = concatenate(resultSequence,visitor.value);
            }
        }
        else
            错误报告;
```

九、 原语函数——filterAt()

Value filter(Value inSequence, QName bindingFnName, MExprEval visitor)

功能： 用名为 bindingFnName 的绑定函数过滤序列 inSequence，并记录当前处理的项在序列中的位置

输入： 参数 inSequence 为带过滤的序列
参数 bindingFnName 为绑定函数名
参数 visitor 为求值访问者
中间变量：变量 resultSequence 为存储过滤结果的序列
变量 i 为循环计数器，初值为 0
算法：

```
生成一个存储过滤结果的序列 resultSequence;
if(inSequence 为空);       // 不执行任何操作
else if(inSequence 为 SequenceList 实体)
    resultSequence = concatenate(      //分别过滤 head 和 tail 并连接过滤结果
                filterAt(inSequence.head,bindingFnName,visitor),
                filterAt(inSequence.tail,bindingFnName,visitor));
else if(inSequence 为 Closure 实体){
    利用闭包中存储的环境和表达式求出闭包的值;
    resultSequence = filterAt(inSequence.value,bindingFnName,visitor);
}
else if(inSequence 为 Value 实体){
    从 visitor.env 中读取绑定函数;
    while(inSequence 中的每一项 temp){
        循环计数器 i 递增;
        向环境 visitor.env 中存储绑定变量 temp 的值;
        绑定函数求值;
        if(求得的结果符合过滤要求)
            resultSequence = concatenate(resultSequence,visitor.value);
    }
}
else
    错误报告;
```

十、 原语函数——quantifiedSome()

Value quantifiedSome(Value inSequence,QName bindingFnName,MExprEval visitor)

功能： 检查序列 inSequence 中是否含有某项满足名为 bindingFnName 的绑定函数定义的条件

输入： 参数 inSequence 为待检查的序列；
参数 bindingFnName 为绑定函数名；
参数 visitor 为求值访问者；

输出： 如果 inSequence 中所有项都满足名为 bindingFnName 的绑定函数定义的条件，则返回真；否则，返回假；

算法：

```
if(inSequence 为空)
    return FALSE;
else if(inSequence 为 SequenceList 实体){
    if(quantifiedSome(inSequence.head, bindingFnName, visitor))
        return TRUE;    // 若头序列某项满足判断条件，返回真
    else if(quantifiedSome(inSequence.tail,bindingFnName,visitor))
        return TRUE;    // 否则若为序列中某项满足判断条件，返回真
    else
        return FALSE;   // 若头、尾序列中各项都不满足条件，返回假
}
else if(inSequence 为 Closure 实体){  // 对闭包的求值结果判断
    利用闭包中存储的环境和表达式求出闭包的值;
    return quantifiedSome(inSequence.value, bindingFnName, visitor);
}
else if(inSequence 为 Value 实体){
    从 visitor.env 中读出绑定函数;
    while(inSequence 中的每一项 temp){
        向环境 visitor.env 中存储绑定变量 temp 的值;
        求绑定函数的值;
        if(求得的值为真)
            return TRUE;
    }
    return FALSE;
}
else
    错误报告;
```

十一、 原语函数——quantifiedEvery()

Value quantifiedEvery(Value inSequence,QName bindingFnName,MExprEval visitor)

功能： 检查序列 inSequence 中的所有项是否都满足名为 bindingFnName 的绑定函数定义的条件

输入： 参数 inSequence 为待检查的序列；
参数 bindingFnName 为绑定函数名；
参数 visitor 为求值访问者；

输出： 如果 inSequence 中所有项都满足名为 bindingFnName 的绑定函数定义的条件，则返回真；否则，返回假；

算法：

```
if(inSequence 为空)
    return FALSE;
else if(inSequence 为 SequenceList 实体){
    if(!quantifiedEvery(inSequence.head, bindingFnName, visitor))
        return FALSE;   // 若头序列中有不满足条件的项，返回假
    else if(!quantifiedEvery(inSequence.tail,bindingFnName,visitor))
        return FALSE;   // 若尾序列中有不满足条件的项，返回假
    else
        return TRUE;    // 若头、尾序列中各项均满足条件，返回真
}
else if(inSequence 为 Closure 实体){  // 对闭包的求值结果判断
    利用闭包中存储的环境和表达式求出闭包的值;
    return quantifiedEvery(inSequence.value, bindingFnName, visitor);
}
else if(inSequence 为 Value 实体){
    从 visitor.env 中读出绑定函数;
    while(inSequence 中的每一项 temp){
        向环境 visitor.env 中存储绑定变量 temp 的值;
        求绑定函数的值;
        if(求得的值为假)
            return FALSE;
    }
    return TRUE;
}
else
    错误报告;
```

十二、 原语函数——and()

SingleBoolean and(SingleBoolean arg, MExpr expr, MExprEval visitor)

功能：与操作

输入：参数 arg 为第一个与操作数

参数 expr 为第二个与操作数的求值表达式

参数 visitor 为第二个与操作数的求值访问者

输出：与操作结果

算法：

```
if(arg == FALSE)    // 若第一个操作数为假
    return FALSE;
else{
```

```
        value =  表达式 expr 在其求值访问者提供的环境中的求值结果;
        if(value == FALSE)        // 若第二个操作数的计算表达式的值为假
            return FALSE;
        else
            return TRUE;
    }
```

十三、 原语函数——or()

SingleBoolean or(SingleBoolean arg, MExpr expr, MExprEval visitor)

功能：或操作

输入：参数 arg 为第一个或操作数

参数 expr 为第二个或操作数的求值表达式

参数 visitor 为第二个或操作数的求值访问者

输出：或操作结果

算法：

```
    if(arg == TRUE)       // 第一个操作数为真
        return TRUE;
    else{
        value =  表达式 expr 在其求值访问者提供的环境中的求值结果;
        if(value == TRUE)    // 第二个操作数的计算表达式的值为真
            return TRUE;
        else
            return FALSE;
    }
```

附录二 系统测试用例

下面列出面向查询优化的 XQuery 查询引擎系统已经通过的测试用例，共计 48 个，包括 32 个 W3C 测试用例和 16 个自定义用例。因篇幅所限，仅列出各用例的测试目标、优化前后的执查询行时间和优化效率。其中 W3C 测试用例所用编号与 W3C 用例规范[9]完全一致。执行时间和优化效率的计算公式见第六章。

1 自定义测试用例

用例名称	测试目的	优化前执行时间(毫秒)	优化后执行时间(毫秒)	优化效率(%)
ts_00.xq	FLWOR 表达式的简单测试	166	140	15.66
ts_01.xq	FLWOR 表达式的测试	653	689.2	-5.54
ts_02.xq	条件表达式的测试	598.4	612.8	-2.41
ts_03.xq	量词表达式 some 的测试	583	550.8	5.52
ts_04.xq	量词表达式 every 的测试	596.8	553	7.34
ts_05.xq	多个数据源、多种表达式的综合测试	595	607	-2.02
ts_06.xq	自定义函数测试（无参数）	605	615	-1.65
ts_07.xq	自定义函数测试（1 个参数）	681	751	-10.28
ts_08.xq	自定义函数测试（多个参数）	594.8	671	-12.81
ts_09.xq	自定义函数测试（多个函数）	655	687	-4.89
ts_10.xq	concatenate()/count()原语测试	82	90.2	-10
ts_11.xq	filter()原语测试	563	569	-1.07
ts_12.xq	subsequence()原语测试	94	90	4.26
ts_13.xq	表达式、原语综合测试	891.4	803.2	9.89
ts_14.xq	and()/or()原语测试	600.8	659	-9.69
3.xq	中规模查询综合测试	729	751	-3.02

2 W3C 测试用例

用例名称	测试目的	优化前执行时间(毫秒)	优化后行行时间(毫秒)	优化效率(%)
1.2.4.1 Q1.xq	递归的自定义函数的测试	162	208.2	-28.52
1.2.4.2 Q2.xq	属性轴和原素构造的混合测试	176	130.2	26.02
1.2.4.3 Q3.xq	序列表达式的测试	136.2	142.4	-4.55
1.2.4.4 Q4.xq	count 函数的测试	178.4	162.2	9.08
1.2.4.5 Q5.xq	count 函数的测试	130.4	140	-7.36
1.2.4.6 Q6.xq	自定义函数的测试	208.8	198.4	4.98
1.3.4.1 Q1.xq	谓词的测试	527	540.4	-2.54
1.3.4.2 Q2.xq	谓词的测试	526.6	548.6	-4.18
1.3.4.3 Q3.xq	Dot 结点在谓词中的测试	518.5	550.8	-6.23
1.3.4.4 Q4.xq	量词的测试	532.8	556.8	-4.51
1.3.4.5 Q5.xq	except 表达式的测试	601	637	-5.99
1.4.4.2 Q2.xq	多表连接和排序的测试	731.2	767.2	-4.92
1.4.4.3 Q3.xq	多表连接的测试	652.8	641	1.81
1.4.4.4 Q4.xq	empty 函数的测试	681	709.2	-4.14
1.4.4.5 Q5.xq	多表连接、Max 函数和排序的混合测试	7500.8	1704.6	77.27
1.4.4.6 Q6.xq	聚合函数 Max 的测试	681	711	-4.41
1.4.4.7 Q7.xq	聚合函数 max 和谓词中的 or 表达式的测试	663	697	-4.22
1.4.4.10 Q10.xq	聚合函数 Max 和排序的测试	1354	909.4	32.84
1.4.4.11 Q11.xq	聚合函数 max 的测试	731	755.2	-3.35
1.4.4.15 Q15.xq	count 函数的测试	787	715.2	9.12
1.4.4.16 Q16.xq	排序、empty 函数和多文档连接的混合测试	671	751	-11.92
1.4.4.17 Q17.xq	量词的测试	875.2	745	14.88
1.4.4.18 Q18.xq	自查询和排序的混合测试	911.2	867	4.85
1.6.4.1 Q1.xq	contains 函数的测试	100.6	80.2	20.28

1.6.4.5 Q5.xq	contains 函数和 string 函数的测试	597	603	-1.01
1.1.9.1 Q1.xq	FLWR 表达式、元素构造和 attribute 轴的测试	605	653	-7.93
1.1.9.2 Q2.xq	嵌套 for 表达式的测试	168.2	148.2	11.89
1.1.9.3 Q3.xq	FLWR 表达式、元素构造的测试	148.4	114.4	22.91
1.1.9.5 Q5.xq	多文档连接的测试	616.8	637	-3.28
1.1.9.6 Q6.xq	if 表达式和 count 函数的测试	584.8	605	-3.45
1.9.9.11 Q11.xq	多个查询的结果连接的测试	138	144	-4.35
1.1.9.12 Q12.xq	自查询、排序和结点比较的测试	655	689	-5.19

致　谢

在本论文完成之际，我要对曾经给予我无私帮助的各位老师和同学们表示由衷的感谢。首先，向廖湖声教授和高红雨老师致以深深的谢意。在我完成毕业设计的一个学期中，廖老师和高老师对我的学习、研究课题的开展和完成给予了很大的帮助，我的毕业设计和论文正是在两位老师的精心指导下才得以顺利完成。廖老师为人正直、治学严谨、知识渊博；高老师做事认真负责、不忽视小节，他们是我学习的好榜样。与此同时，也向所有传授给我知识的老师表示由衷的感谢。

感谢同一课题组的陈悦、钟璐潞同学对我的帮助，在与她们进行讨论的过程中，我得到了很多有益的启发。

我能够圆满完成这次毕业设计任务，离不开我的父母和朋友在精神上的鼓励和支持，以及生活上的关怀和照顾，在此向他们表示感谢。

最后，感谢论文评审委员会的老师对我论文的指正。