Founder 方正信产集团数据库中心(上海)

非关系数据库查询引擎内核分析

Xqilla 剖析

XML 数据库查询引擎-Xqilla 详解

李浩@查询引擎组

2014/1/18

ul XQilla 解析

目录

键入章标题(第 1 级) **1**

键入章标题(第 2 级)..... 2

键入章标题(第 3 级)..... 3

键入章标题(第 1 级) **4**

键入章标题(第 2 级)..... 5

键入章标题(第 3 级)..... 6

0. 概述

1. Xqilla 是什么？

Xqilla 为一款开源软件，其主要用来完成对于 XML 查询语句标准 xquery/xpath 的 FLOWR 语句支持；其主要在是在另一款开源软件 xerces-c/c++的基础上构建而来并且其主要使用 c++来编写。最新的 XQilla release 完成了对 XQuery 3.0 的支持，其中包括了对于高级的 order 函数的支持以及一些优化的更新。 对于具体的 xqilla 新特性可以参考 xqllia 官网上的最新 release notes 介绍，这里我们不做过多的介绍，大家只需要记住一点即可：xqilla 完成了对于 XQuery 中的 FLOWR 语句支持，从而能够使得我们使用这些 FLOWR 语句，完成对于 XML 增删改查等操作。

2. XQILLA 包括哪些功能？

XQilla 提供了对 XML 进行操作的功能，使得我们可以通过 FLOWR 语句来操作 XML 文件，其包括了高级排序函数，FLOWR 语法支持，以及私有函数的支持；同时，其也支持了 W3C 中的全文检索功能。

3. Xqilla 代码结构介绍

我们可以从 XQilla 的官方网站上下载到其最新的代码，其中包括了词法解析，语法解析，抽象语法树， XML 轴操作，DOM-API，查询优化，XQuery Update 等一系列的功能模块，这些模块的具体介绍将在下面的章节中进行详细介绍。

名称	修改日期	类型	大小
analyzer	2013/10/12 16:11	文件夹	
ast	2014/2/11 15:35	文件夹	
axis	2013/10/12 16:11	文件夹	
config	2013/10/12 16:11	文件夹	
context	2013/10/12 16:11	文件夹	
debug	2013/10/12 16:11	文件夹	
dom-api	2013/10/12 16:11	文件夹	
events	2013/10/12 16:11	文件夹	
exceptions	2013/10/12 16:11	文件夹	
fastxdm	2013/10/12 16:11	文件夹	
framework	2013/10/12 16:11	文件夹	
fulltext	2013/10/12 16:11	文件夹	
functions	2013/10/18 16:43	文件夹	
items	2013/10/12 16:11	文件夹	
lexer	2013/10/12 16:11	文件夹	
mapm	2013/10/12 16:11	文件夹	
operators	2013/10/12 16:11	文件夹	
optimizer	2013/10/12 16:11	文件夹	
parser	2013/10/12 16:11	文件夹	
runtime	2013/10/12 16:11	文件夹	
samples	2013/10/12 16:11	文件夹	
schema	2013/10/12 16:11	文件夹	
simple-api	2013/10/12 16:11	文件夹	
update	2013/10/12 16:11	文件夹	

图零：xqilla 的代码结构

4. Xqilla-xpath/xquery 查询引擎分析

下面我们将以一个 FLOWR 语句为蓝本，从该 FLOWR 语句出发来逐层的剖析 XQilla 这一个非结构化的查询引擎的工作情况，以期望能够解决当今对于 XQilla 内核剖析文档不多且不够深入的这一情况。从而帮助读者能够通过对于 XQilla 这一专门用来处理 XML 非关系型的数据库的查询引擎的剖析来认识到关系型数据库的查询引擎与非关系数据库的查询引擎之间的异同，从而帮助读者对于数据库的查询引擎的认识更加全面和深入。 我们选取 W3C 的 Test Suit 中关于 FLOWR 语句的 test case 作为我们分析的实例，其语句如下：

```
query let $sorted :=for $auction in doc("test/test.xml")//closed_auctions/closed_auction,
                    $person at $count1 in doc("test/test.xml")//people/person,
            $item at $count2 in doc("test/test.xml")//regions//item
            where $item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person and
            $auction/price>300
            return <result>
              {<item item="{$item/@id}"
                name="{$item/name/text()}"
                count="{$count2}"/>,
              <author name="{$person/name/text()}"
                gender="{$person//gender/text()}"
                count="{$count1}"/> ,
              <price price="{$auction/price/text()}"/>
              }</result>
    for $sortedResult at $count in $sorted
    return <p>{$count}.{$sorted}</p>;
```

图一：查询语句 A

对于上述的 FLOWR 语句中的一些关键词或者函数以及其语法读者可以参考相应的 XQuery 教学书籍，在这里我们将不再详述，本书的重点是讨论 XML 查询引擎的内部工作原理。对于一个 XQuery FLOWR 语句，当系统获得用户输入后，该语句首先会被送到词法解析器中进行相应的词法有效性验证，例如用户所输入的是否符合我们对于一个单词的定义，一个变量是否是按照我们所规定的规则来书写等。在完成了对用户语句的词法验证后，系统将进入下一个过程语法分析阶段。在该阶段中，系统将根据语法规则来完成对用户查询语句在语法层面上的处理，该处理的产生物通常为一颗抽象的查询语法树。直到此时，我们可能觉得对于那些熟知关系数据库查询引擎的读者来说，上述的过程不会陌生，也许在心中有了自己对于非关系数据库-XML 查询引擎的一幅全景图，诸位切勿心急，待我慢慢道来，也许读完此书你会有：“原来如此，这就是非关系数据库查询引擎啊”的感觉，此时我们期望你已经对于非关系数据库的查询引擎做到了然于心了。 对于词法解析时候所用到的规则我们可以在 src/lexer 中的 XQLexer.l 中找到对于词法规则的描述；同时，我们可以在 src/parser 中的 XQParser.y 中找到对于语法规则的描述。对于词法分析和语法分析这一对，其是让机器理解我们人类语言的一个有力工具，在我们的语法分析中，乔姆斯基范式以及乔姆斯基文法对于语法分析有着相当大的影响，而乔姆斯基本身就是一位语言学家。我们在制定了一套对于 XQuery FlOWR 语句进行描述的语法规则后，即可使得计算机能够理解我们对其发起的 XQuery 查询语句。对于.l 和.y 文件有两大利器：Lex/Flex 和 Yacc/Bison，对于这些工具在此不再进行详细的介绍，对于 Lex/Yacc 不熟悉的读者可自行参考相关资料，在这里我们就不在详述。

```
#include "storage/lmgr.h"
#include "utils/date.h"
#include "utils/datetime.h"
#include "utils/numeric.h"
#include "utils/xml.h"


/*
 * Location tracking support --- simpler than bison's default, since we only
 * want to track the start position not the end position of each nonterminal.
 */
#define YYLLOC_DEFAULT(Current, Rhs, N) \
    do { \
        if ((N) > 0) \
            (Current) = (Rhs)[1]; \
        else \
            (Current) = (-1); \
    } while (0)

/*
 * The above macro assigns -1 (unknown) as the parse location of any
 * nonterminal that was reduced from an empty rule.  This is problematic
 * for nonterminals defined like
 *      OptFooList: / * EMPTY * / { ... } | OptFooList Foo { ... } ;
 * because we'll set -1 as the location during the first reduction and then
```

```
#include <xqilla/utils/XPath2Utils.hpp>
#include <xqilla/utils/XPath2NSUtils.hpp>
#include <xqilla/utils/UTF8Str.hpp>
#include "../config/xqilla_config.h"

#define YYPARSE_PARAM qp
#define YYDEBUG 1
#define YYERROR_VERBOSE

// this removes a memory leak occurring when an error is found in the query (we throw an exception from inside
// yyerror, preventing the bison-generated code from cleaning up the memory); by defining this macro we use
// stack-based memory instead of the heap
#define YYSTACK_USE_ALLOCA  1
#if HAVE_ALLOCA_H
#include <alloca.h>
#elif defined __GNUC__
#undef alloca
#define alloca __builtin_alloca
#elif defined _AIX
#define alloca __alloca
#elif defined _MSC_VER
#include <malloc.h>
#else
#include <stddef.h>
#ifdef __cplusplus
extern "C"
#endif
```

上述图中第一副图为关系数据库 PostgreSQL 中的.y 文件中的说明部分的代码示意，下面为非关系数据库 XML 数据库中的.y 文件中的说明部分的代码示意，因为该部分代码在由 Yacc/Bison 生成所对应的 cpp 文件时会将该部分拷贝到所生成的 cpp 文件中，因而该部分主要是一些头文件包含和一些定义等。

紧随语法文件.y 中第一部分说明部分之后的是定义部分，在部分中给出了在语法分析过程中所用到的一些变量或类型的类型定义。下面图中的第一幅图为非关系数据库 XML 数据中语法文件.y 关于语法变量的一些定义，而第二幅为关系数据库 PostgreSQL 中语法文件.y

中所对于语法变量的定义的描述。

```
%token _CLOSE_APOS_            "' (close)"
%token _LBRACE_                "{"
%token _LBRACE_EXPR_ENCLOSURE_ "{ (expression enclosure)"
%token _RBRACE_                "}"
%token _SEMICOLON_             ";"
%token _BANG_                  "!"
%token _HASH_                  "#"

%token <str> _INTEGER_LITERAL_ "<integer literal>"
%token <str> _DECIMAL_LITERAL_ "<decimal literal>"
%token <str> _DOUBLE_LITERAL_ "<double literal>"
%token <str> _ATTRIBUTE_ "attribute"
%token <str> _COMMENT_ "comment"
%token <str> _DOCUMENT_NODE_ "document-node"
%token <str> _NODE_ "node"
%token <str> _PROCESSING_INSTRUCTION_ "processing-instruction"
%token <str> _SCHEMA_ATTRIBUTE_ "schema-attribute"
%token <str> _SCHEMA_ELEMENT_ "schema-element"
%token <str> _TEXT_ "text"
%token <str> _EMPTY_SEQUENCE_ "empty-sequence"
```

```
%type <node>    select_no_parens select_with_parens select_clause
                simple_select values_clause

%type <node>    alter_column_default opclass_item opclass_drop alter_using
%type <ival>    add_drop opt_asc_desc opt_nulls_order

%type <node>    alter_table_cmd alter_type_cmd opt_collate_clause
%type <list>    alter_table_cmds alter_type_cmds

%type <dbehavior>   opt_drop_behavior

%type <list>    createdb_opt_list alterdb_opt_list copy_opt_list
                transaction_mode_list
                create_extension_opt_list alter_extension_opt_list
%type <defelt>  createdb_opt_item alterdb_opt_item copy_opt_item
                transaction_mode_item
                create_extension_opt_item alter_extension_opt_item

%type <ival>    opt_lock lock_type cast_context
%type <ival>    vacuum_option_list vacuum_option_elem
%type <boolean> opt_force opt_or_replace
                opt_grant_grant_option opt_grant_admin_option
                opt_nowait opt_if_exists opt_with_data
```

在语法文件.y 的第三部分为语法规则定义部分，而该部分为整个语法文件的核心部分，通过对语法规则的描述，我们就可以完整的描述一个语言规则，例如当我们读到一个特定的单词后我们在语法层面上所需要做的动作是什么？当我们读完一个完整的句子时，我们所做的动作又是什么？下图分别为：非关系数据库以及关系数据库的语法规则描述示意图。

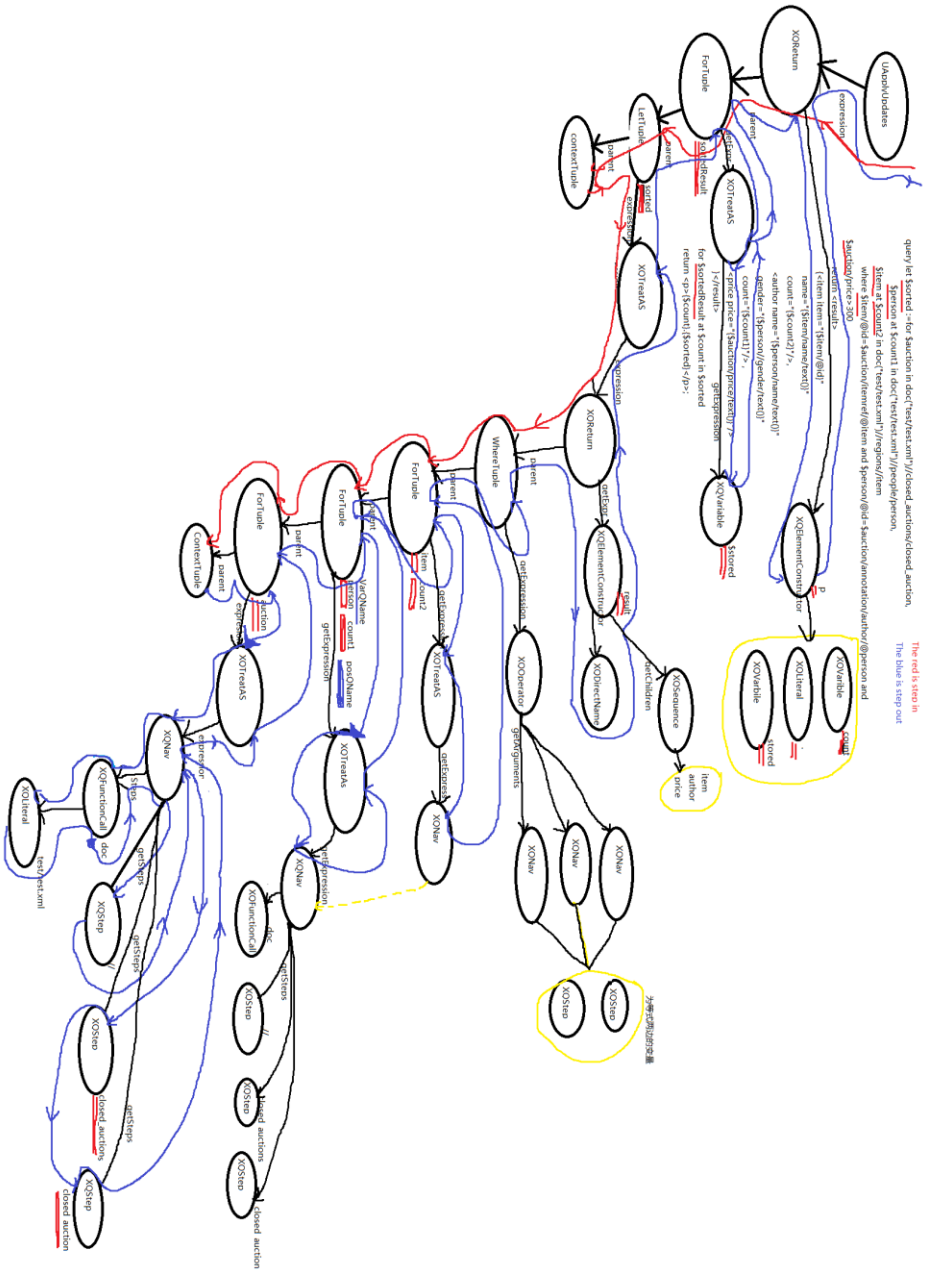
```
Sequence_XSLT:
    SequenceAttrs_XSLT _XSLT_END_ELEMENT_
  {
    $$ = $1;
  }

SequenceAttrs_XSLT:
    _XSLT_SEQUENCE_ _XSLT_SELECT_ Expr
  {
    // TBD xsl:fallback - jpcs
    $$ = PRESERVE_NS(@2, $3);
  }
  ;

If_XSLT:
    IfAttrs_XSLT SequenceConstructor_XSLT _XSLT_END_ELEMENT_
  {
    ASTNode *empty = WRAP(@1, new (MEMMGR) XQSequence(MEMMGR));
    $$ = WRAP(@1, new (MEMMGR) XQIf($1, $2, empty, MEMMGR));
  }
  ;

IfAttrs_XSLT:
    _XSLT_IF_ _XSLT_TEST_ Expr
  {
    $$ = PRESERVE_NS(@2, $3);
  }
```

```
/* Don't bother trying to fold the first two rules into one using
 * opt_table.  You're going to get conflicts.
 */
privilege_target:
            qualified_name_list
                {
                    PrivTarget *n = (PrivTarget *) palloc(sizeof(PrivTarget));
                    n->targtype = ACL_TARGET_OBJECT;
                    n->objtype = ACL_OBJECT_RELATION;
                    n->objs = $1;
                    $$ = n;
                }
            | TABLE qualified_name_list
                {
                    PrivTarget *n = (PrivTarget *) palloc(sizeof(PrivTarget));
                    n->targtype = ACL_TARGET_OBJECT;
                    n->objtype = ACL_OBJECT_RELATION;
                    n->objs = $2;
                    $$ = n;
                }
            | SEQUENCE qualified_name_list
                {
                    PrivTarget *n = (PrivTarget *) palloc(sizeof(PrivTarget));
                    n->targtype = ACL_TARGET_OBJECT;
                    n->objtype = ACL_OBJECT_SEQUENCE;
                    n->objs = $2;
                    $$ = n;
```

在完成了对词法/语法解析的简单介绍后，下面就进入我们的正文环节，对于 XML 数据库查询引擎内核的分析，在此部分我们将从 FLOWR 出发来进行对 XML 数据库查询引擎的剖析工作。

对于上述的查询语句在 AccessControlchecker 阶段所产生的语法树见图(具体的在 AccessCheckControl 阶段所做的工作为相应的权限检查，同时在分析这个操作之前需要将

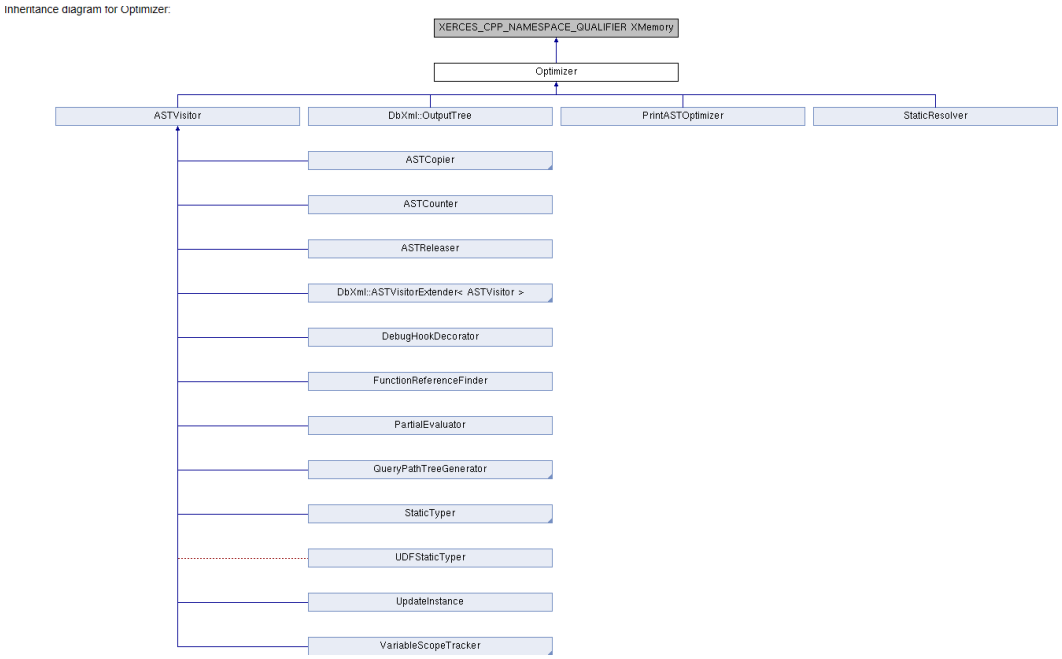
此消息流程以图的形式表示出。），而在完成了权限检查阶段后，接下来所做的事情是进行静态分析操作，即： StaticResolve 阶段。



图二：查询语句 A 的查询语法树

在该阶段的优化过程中将沿着在语法阶段所产生的查询语法树的根本进行遍历。首先将会是由 StaticResolver::optimize(XQQuery *query)函数来进行该阶段的优化工作，从上面的讨论可以知道，对于 Optimizer 的 optimize 函数，如果 Optimizer 的子类对该方法进

行过重载，那么将会调用子类的重载函数，否则就会调用基类的 optimize 函数。



区别于 PostgreSQL 的做法，由于 PostgreSQL 由 c 语言来编写，故而其无法做到 c++ 的函数重载功能，但是其采用了一个巧妙的方法来实现类似函数重载的做法，具体做法参见 PostgreSQL。在 StaticResolver::optimize 函数中会调用 XQuery 的 staticResolution 方法来进行对查询语句的静态解析操作。在该函数中所做的工作为：首先对用户所定义的函数进行静态解析，其具体的操作均由 XQUserFunction 来完成，该函数为处理用户自定义的函数入口，其继承自 FuncFactory，FuncFactory 类为一个纯虚函数，其作用是提供一个接口类，供其子类来完成具体的实现类，FuncFactory::createInstance 来创建一个具体的函数实例。在处理完用户自定义的函数后，接下来处理由用户引入的模块所带来的全局变量，如果在引入的模块中存在着一些变量定义则，会将这些变量加入到全局变量定义中，该项功能由 declareGlobalVar 函数来完成对全局变量的定义，同时检查这些全局变量的命名空间范围，在完成对用户定义函数和变量的声明正确性的检查后，接下来所做的工作是对用户定义的函数的正确性检查，该项功能由用户定义函数对象的 staticResolutionStage2 来完成。在完成对查询语句的前置处理（即：对函数，模块的声明，定义检查等）后，接下是处理查

询语句的处理,即真正进入查询语句的静态解析。而该项工作则是由查询语法树的根部进行。

由于该抽象语法树的根部为一个 UApplyUpadates 对象，故此首先将由 UApplyUpdates::staticResolution 函数开始进行，而在该函数中完成对类 UApplyUpdates 的成员函数 expr_的优化，而 UApplyUpdates 类的成员变量 expr_的类型为一个 XQReturn 对象。进而处理流程会进入到 XQReturn 对象的 XQReturn::staticResolution 函数中来处理 UApplyUpdates 的成员变量 expr_。对于 XQReturn 对象，我们知道对于一个形如 xxx return xxxx 的查询语句，对其进行语法处理后，其生成的 XQReturn 语句中包括两个部分：(1) return 部分；(2) return 语句所在语法树的父节点。故而，在函数 XQReturn::staticResolution 中会处理 XQReturn 的两个部分。

处理的是 XQReturn 的父节点。从语句所生成的语法树得知该 XQReturn 对象的父节点为 ForTuple 节点,因此在 XQReturn::staticResolution 函数中所包含的两个处理步骤的第一步处理 XQReturn 的父节点，就变为对 ForTuple 节点进行的 staticResolution。在 ForTuple::staticResolution 处理函数中，我们可以看出其第一步所做的操作时对其父节点进行 staticResolution 处理。而 ForTuple 的父节点又为一个 LetTuple 对象，故而，处理过程又会进入到 LetTuple::staticResolution 函数中，在对 LetTuple 的处理过程中首先是对该节点的父节点进行处理，而此 LetTuple 节点的父节点又为一个 ContextTuple 对象，在对 ContextTuple 的 staticResolution 处理是将其自身返回。在对 ContextTuple 的处理后，则会返回到 LetTuple::staticResolution 函数的处理过程中，接下来是对 LetTuple 对象中 Let 语句的变量进行相应的设置，此时会根据该 LetTuple 对象中的 QName 对 LetTuple 对象中的 var_(XQVariable 类型)成员变量进行设置。在完成了对 LetTuple 的成员变量 var_的设置后，接下来需要对 LetTuple 的 expr_对象进行优化，即 let 语句的主体

进行相应的优化。我们由 let $var:=xxx 的语句可以看出，对 let 语句的处理主要在于对于该语句主体的处理部分。而对于上述查询语句中的 let 语句的主体为 一个 ForTuple 对象，考虑我们可能存在这个对于语句的执行结果进行某些类型转换操作，例如：let $a:=xs:integer (for xxx)。因此并不是将 let 语句的主体部分直接使用而是将此语句的主体部分封装在一个 XQTreatAs 类型中。因此在 XQTreatAs::staticResolution 函数中我们首先要进行处理的是对 XQTreatAs 类型中的类型进行相应的 staticResolution，而 XQTreatAs 类中的成员变量_exprType 的类型为一 SequenceType 类型，故而会进一步的进入 SequenceType::staticResolution 中进行处理，而最终会进入到 ItemType::staticResolution 函数中进行最终的类型处理。函数 ItemType::staticResolution 所做的工作是：首先获得默认的命名空间，而后根据其类型进行判断：其类型是否为定义所给出的类型，如果该类型与我们所给出类型定义不相符，系统抛出类型不匹配异常。

在系统完成对 XQTreatAs 中的类型处理后，接下来的工作是对 XQTreatAs 中的 expression 进行 staticResolution 处理。从语法树可以看到 XQTreatAs 中的 expr_对象为一个 XQReturn 对象。而在对 XQReturn 对象的 staticResolution 过程中，首先完成的是对 XQReturn 对象的父节点的 staticResolution，进而系统的处理流程进入 WhereTuple::staticResolution 函数中，因为 WhereTuple 节点为 XQReturn 在语法树上的父节点。同样道理，我们在处理 WhereTuple 的 staticResolution 的过程中，也会首先处理 WhereTuple 在查询语法树的父节点。而这个 WhereTuple 的父节点为一个 ForTuple，这是处理流程会进入到 ForTuple 的父节点中，而从我们的查询语句中可以看出，该 for 语句中包括三个:一个为处理变量 item，另外一个为处理变量 person，第三个为处理变量 auction，而这个三个 ForTuple 的关系为：ForTuple("item") 的父节点为

ForTuple("person")，而 ForTuple("person")的父节点为 ForTuple("auction")，而这些我们也可以从抽象查询语法树上清晰的得出这个结论。而 ForTuple("auction")的父节点为一个 ContextTuple，因此优化的最后流程进入到对这个 ContextTuple 的优化过程，而系统对于 ContextTuple 的 staticResolution 优化则是执行 return this;操作。在完成了对 ContextTuple 的优化操作后，系统将返回其调用者 ForTuple("auction") 的 ForTuple::staticResolution 函数中并取得变量的名称，uri 等信息，然后根据这些信息设置完成 ForTuple 成员变量的 var_，posVar_，scoreVar_的设置，然后对该 For 语句“doc("test/test.xml")//closed_auctions/closed_auction”进行 staticResolution 操作。而对于语句“doc("test/test.xml")//closed_auctions/closed_auction”则是将其处理为四个 XQStep 对象（XQStep("doc("test/test.xml")", XQStep("//"), XQStep("closed_auctions"), XQStep("closed_auction"))）并将这个四个封装在一个 XQNav 对象中，而将这个 XQNav 对象封装在成一个 XQTreatAs。故而对这个语句进行 staticResolution 处理首先会由 XQTreatAs 对象的 staticResolution 开始。我们知道对于一个变量我们在声明的时候总是带有对该变量类型的说明。对于一个没有类型说明的变量，我们有两种处理方式：（1）语法错误，当我们获得一个变量时，如果系统没有检测到对该变量的类型声明，则系统抛出一个变量定义错误；（2）使用一个默认类型，当系统检测到一个变量声明，发现该变量并没有一个类型描述，则系统采用一个默认的类型来描述该变量。而在 XQilla 中则采用第二中方式对于没有给定类型的变量，则系统给出一个默认的类型描述。因此对于变量$auction 系统给出了一个默认的类型说明。回到我们对于 XQTreatAs 的讨论中，该对象完成了对语句“doc("test/test.xml")//closed_auctions/closed_auction”的封装，而该语句有描述变量$auction 的行为，从所给出的查询语句中我们并没有给出对变量$auction 的类型描述，因此系统给出一个默认类型。至此，我们可以得出一个结论：

在 XQTreatAs 对象中包括两个方面：（1）语句的类型；（2）语句。在 XQTreatAs 的 staticResolution 函数中我们首先要完成对类型的 staticResolution，在完成对类型的 staticResolution 后，放可进行对语句自身的 staticResolution。由 XQParse.y 中对于类型的描述：

```
 // [121]    TypeDeclaration    ::=    "as" SequenceType
TypeDeclaration:
 /* empty */
 {
  $$ = WRAP(@$, new (MEMMGR) SequenceType(new (MEMMGR) SequenceType::ItemType(SequenceType::ItemType::TEST_ANYTHING
  ), SequenceType::STAR));
 }
 | _AS_ SequenceType
 {
  REJECT_NOT_XQUERY(TypeDeclaration, @1);

  $$ = $2;
 }
 ;
```

我们可以看出对于没有给定类型说明时，系统则将其处理为一个 SequenceType 类型，而对于给定类型则，按照所指定的类型来处理。对于本例中的情况，系统则是将其按照默认类型：SequenceType 来处理。故而上述 XQTreatAs 中所述的对类型的 staticResolution 处理实质上是对 SequenceType 的 staticResolution 处理。在 SequenceType::ItemType::staticResolution 中的主要任务是：如果系统给定了返回类型则根据给定的类型检查该类型是否是已经定义的合法类型；否则，根据 schema 中的说明来检查该类型是否已在 schema 中给出描述。

在完成对类型的 staticResolution 后，XQTreatAs 的处理流程来到对语句的处理。而上面我们提到过对于语句“doc("test/test.xml")//closed_auctions/closed_auction”系统则是将该语句包装为一个 XQNav 对象，故而此时的处理流程则会进入 XQNav::staticResolution 中。

而在其处理的过程中对于 XQxxx 对象中如果只是简单对 ASTNode 对象的封装，例如对于 XQPromoteAnyURI* strText 中的成员变量 expr_为一个原始 ASTNode 对象，而不其子类的形式，则我们在会将该 XQxxx 对象在 staticTyping 的过程中将 XQxxx 对象使用其

直接类型 ASTNode 来代替，而将该 XQxxx 从相应的查询语法树上进行删除掉，这样保持了整个查询语法树的整洁，而原始的查询语法树与 ASTNode 树之间存在着等价的关系，我们可以从一种形式转换到另外一种形式，并且在我们生产查询计划的时候也是使用 ASTNode 这种形式来生成。

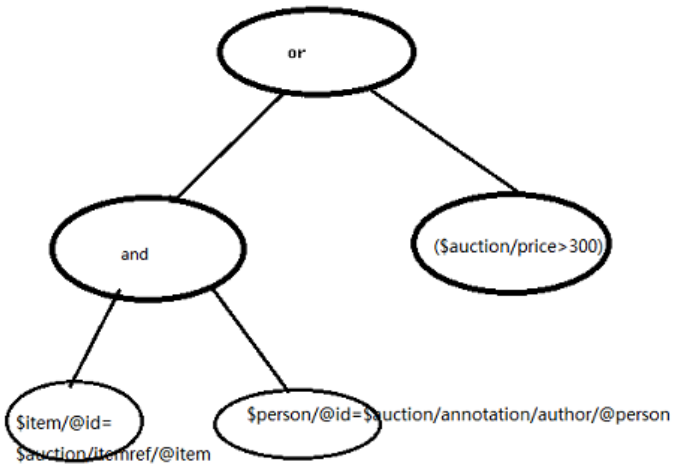
[注意：在对于 XQNav 对象中的 XQFunctionCall 进行 staticResolution 操作时，其 XQFunctionCall::staticResolution 函数中会将自身从语法树上删除掉，取而代之的是一个从函数表中所查询出的结果一个 ASTNode*对象，当该函数在函数表中不存在，即：无法从函数表中查找到该函数的定义和说明，系统会抛出一个函数不存在的异常信息。更加确切的说是一个 FunctionDoc 对象，而该对象又继承自 XQFunction。而对于 FunctionDoc 进行 staticResolution 操作时候，其主要的操作为对其参数进行解析，即会调用 resolveArguments 来完成对函数参数的 staticResolution 操作。而对于参数的处理时会将函数参数转为期望的类型，而后被封装在一个个对应的类型中，例如，将 XQAtomize 封装在 XQPromoteUntyped，而后将这个 XQPromoteUntyped 对象继续封装在 XQPromoteNumeric 中，继续对其进行处理，将其封装在 XQPromoteAnyURI 中，最后以 XQTreatAs 的形式返回，在此类型转换过程中如果函数参数类型与期望的类型不一致则系统会抛出类型不符合的异常。接下来是对 XQStep 的 staticResolution 操作，而 XQStep 内部则是由一个 NodeTest 类型的*nodeTest_构成，因此对于 XQStep 的 staticResolution 操作其实质是对 NodeTest 进行 staticResolution 操作。而在 NodeTest::staticResolution 函数中其会根据对象的类型来进行处理，当对象的类型为 SequenceType::ItemType 时候，会将该 SequenceType::ItemType 类型进行转换为 NodeTest 类型，因为 SequenceType::ItemType 类型处理起来并不方便而且效率不高。]

按照上述的处理 ForTuple 的方式将语法树上的三个 ForTuple 处理完成，而后系统将会

返回到 WhereTuple 中，上文中我们已经知道在处理 WhereTuple 时候，首先会进入到处理其父节点的子树上，而 WhereTuple 的父节点就是我们上文中重点讨论的 ForTuple。在我们处理完 WhereTuple 的父节点后，我们进入处理 WhereTuple 的表达式阶段，而 WhereTuple 中的表达式即为查询语句中的 where 语句。在明确了 WhereTuple 中的表达式的类型后，我们对于 WhereTuple 的表达式的处理就转入到对 where 语句中的条件语句的处理中来。

我们从查询语句中可以看到其 where 语句中为相同的布尔操作符(全部是 and 或者全部是 or 操作符)，因此在系统在构建该查询语法树时候只会产生一个布尔类型操作符对象来保存布尔条件中的左右两边表达式，例如 A and B and C 系统将会产生一个 And 对象来描述该布尔表达式，并将 A and B 作为该 And 表达式的一个参数，将 C 作为 and 表达式的另外一个参数。但是如果 where 语句中存在着不同的布尔操作符的话，系统将会把这些布尔操作符统一的封装在 XQEffectiveBooleanValue 中,由该对象来描述我们 where 语句中的布尔条件表达式。因此对于 WhereTuple 中的表达式处理就转为对 XQEffectiveBooleanValue 的处理。接着我们上面的讨论，我们将进入 XQEffectiveBooleanValue::staticResolution 中，在该函数中我们首先要处理的是 XQEffectiveBooleanValue 中的表达式，而对于我们的查询语句中的条件语句 where $item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person or $auction/price>300，从该查询语句中我们可以看出其又 and 和 or 两个布尔操作符，在语法分析的时候将上述的条件语法处理为如下的形式：($item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person) or ($auction/price>300)。因此，对于在处理 XQEffectiveBooleanValue 的表达式时，其首先会进入到

Or::staticResolution 函数中进行处理。该 Or 对象的中的参数为 or 操作符两边的表达式，到这里我们就进一步的明确了 ,我们当前所要解决的问题为对 or 对象进行 staticResolution。对于 or 对象的 staticResolution 的主要工作是对其参数进行 staticResolution。在处理每一个参数的时候，首先会将该参数封装在一个 XQEffectiveBooleanValue 对象中，然后在对该 XQEffectiveBooleanValue 进行处理。对于语句 ($item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person) or ($auction/price>300)系统所产生的查询语法树如下：



($item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person) or ($auction/price>300)
语法树

对于该语法子树的处理是首先处理该语法树的左子树部分，然后处理该语法树的右子树部分 ;而在对左子树的处理过程中 ,对于叶子节点中的$item/@id=$auction/itemref/@item，$person/@id=$auction/annotation/author/@person，$auction/price>300，系统会进入到 GeneralComp::staticResolution 中，因为系统会将这些表达式转换为 GeneralComp 对象，由这些 GeneralComp 对象来描述上述的比较表达式。故而，对于上述的语法树的叶子节点 ,GeneralComp ,系统首先会进入 GeneralComp::staticResolution 中处理该比较操作符。例如上述语法树的最左叶子节点：

$item/@id=$auction/itemref/@item，在此 GeneralComp 中，其中成员变量 name 为 comp 表明其是一个比较操作对象，operation_为 EQUAL 表明该比较操作时执行相等判定的比较操作，其中的 args_成员变量用来描述比较操作符两端的操作数。对于 $item/@id=$auction/itemref/@item 这个表达式中等号操作符两边的操作数为：$item/@id 和$auction/itemref/@item，因此 args_成员变量中的第一个元素为：$item/@id ,第二个元素为 :$auction/itemref/@item。从我们上面对于路径的讨论知道，系统对于每一级路径均处理成一个 XQStep 对象然后统一封装在一个 XQNav 对象中。分析到这里，我们可以清楚的得出一个结论：GeneralComp 的成员变量 args_中为两个 XQNav 对象。在分析完 GeneralComp 对象的构成后，我们进入 GeneralComp::staticResolution 函数的分析。在该函数中会依次对 args_中的元素进行 staticResolution 操作，但并不是直接对 XQNav 对象执行 staticResolution 操作，而是将 args_中的元素使用 XQAtomize 封装，然后执行对 XQAtomize 对象的 staticResolution 操作，来完成对 XQNav 对象的 staticResolution 操作。对于 XQNav 对象的 staticResolution 操作前述已经讨论，这里不再浪费笔墨讨论。采用同样的方法在完成对 and 节点右子树的处理后，系统返回到该语法树的根节点 or 节点，然后接着处理右子树。在完成对该语法树的处理后，我们就完成了对 where 语句，即：WhereTuple 的 staticResolution 操作。系统返回到 WhereTuple::staticResolution 的调用者---XQReturn 中，继续对 XQReturn 对象的成员变量 expr_，即：return 语句的 staticResolution 中。

从我们查询语句中的 return 语句可以看出该 return 语句为一个 XQElementConstructor 对象，故而此时的处理流程会进入到 XQElementConstructor::staticResolution 中来，下面就来详细的分析 XQElementConstructor::staticResolution。在该函数中我们首先会处理命名空间（namespace），考虑到我们在生成返回节点的时候，需要考虑返回节点的命名

空间问题。

在进行命名空间的处理时候，我们首先会考虑该命名空间是否是一个 Literal 类型字符串，如果命名空间的类型非常量字符串，系统会给出相应的异常信息。在对命名空间进行上述初步的处理后，下面的公司是对于命名空间的属性进行 staticResolution 处理，而后检查该返回元素节点的属性进行重复属性判定，当发现具有重复属性时，系统给出相应的异常信息，例如以 return 语句中<item>节点为例：

<item item="{$item/@id}" name="{$item/name/text()}"count="{$count2}"/>。

如果该节点中存在两个名为 item 的属性时候，此时系统在检查步骤中应当反映一个异常信息，用以通知用户节点的属性重复。

在完成了对返回元素节点的属性的检查后，接下来的工作是对返回语句中的 elementConstructor 的返回节点名称进行 staticResolution 操作。在完成上述的操作后，系统将会对返回节点的子节点进行 staticResoultion 操作。例如本例中的<item>，<author>，<price>节点，就属于 return 语句的<result>节点的子节点。而对于这些节点子节点，系统在进行语法分析时候将这些子节点以 XQSequence 的形式组织。

在 elementConstructor::staticResoultion 中最后处理部分是对于这些 return 语句中的子节点进行处理。系统并不是直接对这些子节点进行处理，而是将这些子节点分别封装在 XQContentSequence 中，然后以 XQContentSequence 的形式进行处理，而每一字节点又是一个 elementConstructor 对象，因此系统会继续递归的按照上述的流程对该 elementConstructor 对象进行处理，这里不再赘述。在处理上述的子节点时候，对于如下的语句：

<item item="{$item/@id}" name="{$item/name/text()}"count="{$count2}"/>

系统在 elementConstructor 过程中，在完成命名空间和属性重复性的检查后，系统会进入

到 XQAttributeConstructor 的 staticResolution 中，来处理该节点的各个属性节点。具体详细分析均大同小异这里不再耗费篇幅进行讨论。当处理完上述所有过程后，在 elementConstructor::staticResolution 返回之前，系统会根据命名空间的情况分别处理。如果用户指定了相应的命名空间则在返回之前会将该命名空间封装至一个 XQNamespaceBinding 对象中并将该对象返回，否则返回 elementConstructor 对象自身。在处理完上述情况后，此时我们就完成了对于 return 语句的处理。 又由上述我们知道，XQReturn 语句被封装在一个 XQTreatAS 对象中，系统在完成对 XQReturn 对象的处理后，返回到 XQTreatAS::staticResoultion 中，继续完成对表达式的类型的处理，完成对于表达式类型的处理后，系统将会回到 LetTuple::staticResolution 函数中下一不处理。在完成对于 LetTuple::staticResolution 的处理后，此时表明系统已经完成了在 ForTuple::staticResolution 中对于 ForTuple 父节点的处理。此时我们从图二查询语法树可以看出我们已经完成对于该语法树大部分的处理工作。在完成对 Let 语句父节点的处理后，系统将继续 ForTuple 节点在 staticResolution 阶段的处理--- Let 语句本身的处理。对于查询语句 let $sorted :=for $auction in doc("test/test.xml")…以及上述我们对于 Let 语句的讨论时已经论述："考虑我们可能存在这个对于语句的执行结果进行某些类型转换操作，例如：let $a:=xs:integer (for xxx)。因此并不是将 let 语句的主体部分直接使用而是将此语句的主体部分封装在一个 XQTreatAs 类型中，故而在对 LetTuple 的处理时，将以 XQTreatAS 开始，本例中的 let 语句为对一变量进行赋值。因此，在系统对于 XQTreatAS 的处理时，会以首先对该变量所对于的语法对象-XQVaraible 进行 staticResolution 处理。在完成对于该变量的处理后，系统就完成了 storedResult 这个 for 语句的处理。此时系统来到对查询语法树最顶层的 XQReturn 的 return 语句（ {$count}.{$sorted} ）的处理，其处理流程与上述的 XQReturn 语句相同这里不再赘述，读者可自行进行分析。完

成了对于 XQReturn 节点的处理后，系统返回到整棵查询语法树的根节点 UApplyUpdates 的 staticResolution 中，在完成对于 UApplyUpdates 节点的其余部分的 staticResolution 处理后，我们就完成了对于整个查询语句在查询优化的第二个阶段 staticResolution 的处理。此时，查询优化流程进行到第三个阶段---DbXmlStaticTyper 阶段，接下来我们就进入该阶段的分析。

在 static typing 阶段中，首先进行预处理：对系统中的 imported 模块进行 static typing 分析，而后对 imported 变量进行 static typing 分析，用户定义的全局变量进行 static typing，用户定义的函数进行 static typing 分析。在完成上述的预处理后，将会进入对查询语法树 static typing 分析。该查询语法树为经过上一步的 static resolution 操作后的语法树，该语法树作为本阶段分析的初始语法树。对于查询引擎在进行查询语句的优化过程中，上一步所产生的查询语法树作为下一步分析的输入，这样经过一定步骤的查询优化后，该查询语法树必定为我们最后想要的查询语法树，这样查询计划生成器(QPG, Query Plan Generator)便可以根据该查询语法树来生成相应的查询计划。

由语法分析我们知道查询语法树被以 XQuery 的形式来描述，而 XQuery 又继承自 ASTNode，广义上整棵查询语法树上的语法节点均为 ASTNode 对象。对于 static typer 阶段的优化器：StaticTyper 来说，其继承自 ASTVisitor 且其并没有对 ASTVisitor 中的 optimize 函数进行重载操作，因此对于 static typing 阶段的优化来说：并没有使用 StaticTyper 的 optimize 函数，而是直接使用基类 ASTVisitor 的 optimize 函数。在 ASTVisitor::optimize 函数中，系统根据节点的类型进入到不同的处理函数中。其中共有如下类型：

```
typedef enum {
    LITERAL,                    //字符类型
    NUMERIC_LITERAL,    //数字字符类型
    QNAME_LITERAL,      //QName 类型
```

SEQUENCE, //Sequence 类型

FUNCTION, //函数类型

NAVIGATION, //导航类型

VARIABLE, //变量类型

STEP, //XQStep 类型。

IF, ...

INSTANCE_OF,

CASTABLE_AS,

CAST_AS,

TREAT_AS,

OPERATOR,

CONTEXT_ITEM,

DOM_CONSTRUCTOR,

QUANTIFIED,

TYPESWITCH,

VALIDATE,

FUNCTION_CALL,

USER_FUNCTION,

ORDERING_CHANGE,

XPATH1_CONVERT,

PROMOTE_UNTYPED,

PROMOTE_NUMERIC,

PROMOTE_ANY_URI,

DOCUMENT_ORDER,

PREDICATE,

ATOMIZE,

EBV,

FTCONTAINS,

LETSCOREGETTER,

UDELETE,

URENAME,

UREPLACE,

UREPLACE_VALUE_OF,

UTRANSFORM,

UINSERT_AS_FIRST,

UINSERT_AS_LAST,

UINSERT_INTO,

UINSERT_AFTER,

UINSERT_BEFORE,

UAPPLY_UPDATES,

NAME_EXPRESSION,

CONTENT_SEQUENCE,

DIRECT_NAME,

RETURN,

```
    NAMESPACE_BINDING,
    FUNCTION_CONVERSION,
    SIMPLE_CONTENT,
    ANALYZE_STRING,
    CALL_TEMPLATE,
    APPLY_TEMPLATES,
    INLINE_FUNCTION,
    FUNCTION_REF,
    FUNCTION_DEREF,
    COPY_OF,
    COPY,
    MAP,
    DEBUG_HOOK
  } whichType;
```

从图二的查询语法树描述可以看出 整棵查询语法树的根节点为一 UApplyUpdates 类型。ASTVisitor::optimize 函数将根据该类型进入到 optimizeUApplyUpdates 中，对 UApplyUpdates 节点进行处理，而在处理该节点时候，首先处理该节点的表达式。从查询语法树上我们看到，该 UApplyUpdates 节点的表达式为一个 XQReturn 节点，故而此时的处理流程进入到 XQReturn 中，StaticTyper::optimizeReturn。该函数用来处理 XQReturn 语句在 DbXmlStaticTyper 阶段的优化。而在对 XQReturn 进行优化的过程中，我们首先会对该 XQReturn 对象的父节点进行优化处理。只有在完成对该 XQReturn 对象父节点的优化后，然后进行对 XQReturn 对象的表达式的优化，在完成上述的两个优化步骤后，方认为完了对 XQReturn 对象的优化。对于该 XQreturn 语句：return {$count}.{$sorted} ;从查找语句和所生成的查询语法树上可以看出该 return 语句在查询语法树上所对于的父节点为一个 For 语句： for $sortedResult at $count in $sorted。因此此时的优化处理进一步的来到对该 for 语句的处理中。而在处理 for 语句时候，同样首先要处理的是，for 语句的在查询语法树上的父节点，而该 for 语句在查询语法树上的父节点为一个 let 语句：let $sorted :=for $auction in doc...，与上面处理流程一样在处理 let 语句的过程中首先处理的也是其父节点，而 Let 语句的父节点为一个

ContextTuple ,而对于 ContextTuple 在本阶段的优化过程中的处理逻辑为将其自身返回。在完成对于 Let 语句的父节点处理后 ,接着处理 Let 语句的表达式 :for $auction in doc...。从上面的我们的讨论中可知，我们不是直接使用 Let 语句的表达式来进行处理，而是考虑到类型转换，是将其封装在一个 XQTreatAS 的对象中进行处理。下面我们就给出对其的详细分析。在 XQTreatAS 中，首先处理的是其的表达式，为查询语句中中间的 return 语句。

```
return <result>
  {<item item="{$item/@id}"
    name="{$item/name/text()}"
    count="{$count2}"/>,
  <author name="{$person/name/text()}"
    gender="{$person//gender/text()}"
    count="{$count1}"/> ,
  <price price="{$auction/price/text()}"/>
  }</result>
```

与上述对于 XQReturn 处理流程相同，首先处理 return 语句的父节点，而该节点的父节点为 where 语句，这点我们按照就近原则也可以得出一个初略的答案。

```
$item at $count2 in doc("test/test.xml")//regions//item
where $item/@id=$auction/itemref/@item and $person/@id=$auction/annotation/author/@person or
$auction/price>300
return <result>
  {<item item="{$item/@id}"
    name="{$item/name/text()}"
    count="{$count2}"/>,
  <author name="{$person/name/text()}"
```

这也可以从图二中的查询语法树上可以看出。按照首先处理父节点的原则，沿着查询语法树进行遍历，直到查询语法树的叶子节点。在处理该 where 语句中，首先对该语句的父节点进行优化---for $auction in doc("test/test.xml")//closed_auctions/closed_auction, $person at $count1 in doc("test/test.xml")//people/person,$item at $count2 in doc("test/test.xml")//regions//item 语句进行优化。此时优化流程进入到 ASTVisitor::optimizeForTuple 函数中。ASTVisitor 中的 optimizeXXXTuple 函数被用来进行 XXX 语句的优化，例如：optimizeForTuple 被用来进行 For 语句的优化，而 optimizeWhereTuple 被用来对 Where 语句进行优化。而对于上述的 for 语句其中包括了两个：(1) $auction in doc("test/test.xml")//closed_auctions/closed_auction ; (2)

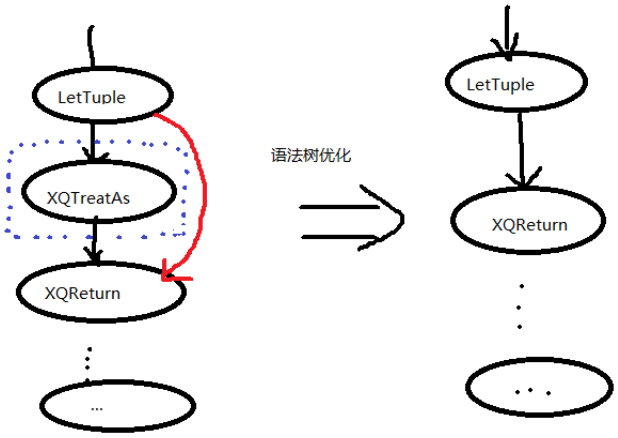
$person at $count1 in doc("test/test.xml")//people/person,$item

对于上述语句的优化同样采用：先对其父节点进行相应的优化，然后在对该语句本身进行优化的方式进行。$auction in doc("test/test.xml")//closed_auctions/closed_auction；其父节点为一个 ContextTuple，而对于 ContextTuple 在该阶段的优化则是直接将该对象返回。完成对其父节点的优化后，接下来要对该语句本身进行优化。在对语句进行优化时，首先判断对于语句中的变量（此处为$auction）是否进行过优化，如果没对该变量进行优化，则首先对该语句进行优化，而后对该变量进行优化（对该变量的作用域，变量声明等进行处理）。但如果已经完成对该变量，则表明对于该语句和语句所对于的变量均已经完成优化，此时将该优化后的对象直接返回。同样对于$person at $count1 in doc("test/test.xml")//people/person,$item 语句也采用与 $auction in doc("test/test.xml")//closed_auctions/closed_auction；语句相同的处理方式进行，这里不再详述。

采用同样的方式，处理完 Where，Return 语句后，此时的处理流程来到对 Let 语句的处理流程中，从查询语句中可以看出在完成对 Let 语句在该阶段的优化后，该查询语句的第一部分就完成了优化，接下来是进行对查询语句的第二部分：for $sortedResult at $count in $sorted return {$count}.{$sorted} ;进行优化。在对查询语句的第二部分优化进行分析之前，我们将继续进行对 Let 语句的分析自身的优化分析。

由上面的讨论可知，Let 语句被以 XQTreatAS 的形式封装，该优化阶段对 XQTreatAs 对象的优化由函数 StaticTyper::optimizeTreatAs 来完成。在该函数中首先进行的是对 XQTreatAs 对象所对于表达式进行优化，在完成了对其所对应表达式的优化后，系统将对该对象所定义的转换函数进行优化处理，因为对于 Let 语句来说，我们可以使用转换函数将表达式所返回的结果转换为一个我们所需要的类型。完成对于 XQTreatAs 对象的优化后，

系统将会检查 XQTreatAs 对象返回的类型是否与需要转换的类型想匹配，例如：我们不能将一个 Sequence 类型的数据转为为一个 double 类型，如果发生此类转换，系统则会抛出一个类型转换错误的异常信息。但如果我们没有使用此类的类型转换那么对于 Let 语句中的表达式进行 XQTreatAs 则无必要，执行此类封装只会增查询语法树的层数，因此如果发现此类情况我们将会去掉此类不必要的封装。在完成对 Let 语句父节点和自身表达式的优化后



系统将处理 Let 语句所声明的变量（变量类型，变量名称，变量的作用域等），在完成这些优化操作后，系统完成对 for $sortedResult at $count in $sorted 语句在语法树上父节点的处理（参照图二），而后处理该 For 语句的表达式，并完成该 For 语句中变量的优化处理（变量类型，变量名称，变量的作用域等）。在完成上述处理后，此时系统将会处理整个查询语句的最后一个语句：return {$count}.{$sorted} ;该语句为一个 XQReturn 语句其父节点为我们上述已经处理完成的 for $sortedResult at $count in $sorted 语句，而该语句的表达式为一 XQElementConstructor，对于 XQElementConstructor 的分析则与上述的第一个 XQReturn 语句中的 XQElementConstructor 相同，所有在这里我们不在赘述。在处理完该 XQReturn 语句后，我们就处理完整棵查询语法树，此时系统回到查询语法树的根节点 UApplyUpdates 中并完成对该节点的 static typer 优化处理：由于该节点在整棵的查询语法树和整个的分析过程中均不参与任何实际优化操作，故而在此阶段的优化过

程中将从查询语法树中删除，将其孩子节点：XQReturn 节点作为新的语法树的根节点，从而达到精简语法树的目标。

至此，我们完成了对该语句查询优化的第三阶段：staticTyper 的分析，此时我们的分析来到查询优化的第四个阶段：ASTReplaceOptimizer，即对经过上述三个过程后的查询语法树进行进一步的优化 :**检查该语法树是否存在冗余并将冗余的语法树分支进行替换或者删除**，从而获得一颗简洁的语法树。

5. 如何增加自己的 Optimizer 以及优化?

6. XQilla 的分布式化的一点想法

7. XQilla-fulltext 分析

8. XQilla-fulltext 分布式化的一点想法。

9. 结束语