并行数据库中的查询自动优化

学　　院　软件学院

专　　业　软件工程

年　　级　2006 级

姓　　名　刘宇杰

指导教师　张坤龙、王轶凡

2010 年　6 月　15 日

摘　　要

并行数据库为海量数据分析、大规模并发事务处理等应用提供高效可靠的数据管理服务。查询处理器作为数据库管理系统中的核心模块，能针对输入的 SQL 查询语句，产生多个候选查询计划并选择较优的查询实现方案作为输出。然而，查询优化器的工程复杂性使得其实现代价高昂。同时，传统的硬编码实现形式使得优化器的维护成本随代码量的增长迅速提高。针对这一问题，论文给出了一种基于树重写的专用语言 TRL。该语言被用于描述查询优化的变换规则，并由其编译器产生针对并行数据库的查询优化器。使用 TRL 语言可以使查询处理器的开发和维护人员在较高的抽象层次进行工作，从而提升开发效率、减少错误，降低开发难度和维护成本。

关键词：查询优化；树重写；程序设计语言；并行数据库；
规则引擎

ABSTRACT

Parallel databases provide reliable and efficient data management service for applications that deal with large scale of data, such as OLAP and OLTP. The query processor, which is responsible for transforming a SQL query to the concrete physical execution plan, requires considerable engineering complexity and expensive cost of maintenance. To address the problem, this paper presents TRL, a domain specific language for rule-based tree rewriting, as a tool for constructing the query processor. TRL keeps the developer and maintainer of the query processor working at a higher level, which promotes productivity and reduces the cost of writing and maintaining a query processor.

Key words：Query Optimization；Tree Rewriting；Programming Languages；
Parallel Database；Rule Engine

目　　录

第一章　概述

1.1　并行数据库中的查询优化

并行数据库为海量数据分析、大规模并发事务处理等应用提供高效可靠的数据管理服务。并行数据库通常运行于计算机集群之上，通过数据划分和作业流水线，实现对多个 CPU、磁盘等计算资源的高效利用，从而提升系统的处理能力。

查询优化器作为数据库管理系统中的核心模块，根据输入的 SQL 查询语句，产生多个候选查询计划并选择较优的查询实现方案作为输出。查询优化通常根据预设的代价估算模型以及数据表的统计信息，对候选查询计划的代价进行预估和选择。由于 SQL 语言具有高度的声明性，在许多应用场合下，其对应的朴素执行方案效率难以为应用接受[2]，因此实现良好的查询优化器对于数据库系统具有重要意义。

与传统的 SQL 优化器相比，并行数据库的优化需要考虑跨节点数据通讯开销，数据的局部性对性能的影响，IO 开销和网络开销等因素。并行数据库通常将数据划分成多组无交集合，并分发到各个计算节点以实现数据并行处理[1]。这种将数据分散到不同节点进行处理的并行化手段称为划分并行（Partitioned Parallelism）。SQL 语言的物理查询计划树由一组物理操作符（Physical Operator）构成，若两个物理操作符之间的数据不形成依赖关系，则其操作可以完全独立地在不同处理器上同时进行。物理操作符本身会消耗和产生元组流（Tuple Stream），操作符产生的元组有时可以喂送给下一个操作符，这种形式的并行性称作流水线并行（Pipelined Parallelism）[10]。查询优化器往往需要通过挖掘这两种形式的并行性，使大规模数据上的查询操作更加高效地利用处理器和磁盘资源，缩短执行时间。此外，并行数据库中的查询在执行过程中会产生重排操作（Repartition），每次重排操做会消耗大量的 IO 以及网络资源，因此查询优化器需要尽可能减少查询过程中的重排操作，减轻磁盘和网络压力。

然而，查询优化器的工程复杂性使得其实现代价高昂。同时，传统的硬编码实现形式使得优化器的维护成本随代码量的增长迅速提高。针对这一问题，本文给出了一种基于树重写的专用语言 TRL。该语言将用于描述查询优化的变换规则，并由其编译器产生针对并行数据库的查询优化器。

1.2　基于规则的查询优化

为了应对查询优化器的工程复杂性，基于规则的查询优化器[7]于 80 年代出现。该类优化器包括一个“代价估计”系统，以对不同的查询执行方案的资源消

耗进行评估。规则引擎的工作过程可分为两个步骤：1) 产生一组候选的查询计划，称作“搜索空间”；2) 对搜索空间中的查询计划进行代价评估并选取最高效的查询计划[2]。

从黑盒的角度来看待查询优化器，对应一棵输入的 SQL 语句的抽象语法树，优化器应当给出一棵物理查询计划树作为输出。在实际的执行过程中，优化器会按一定规则“逐步”的对 SQL 进行变换，其过程可看作为一系列树的“重写”操作，每次“重写”操作即为规则的一次应用。

因此，查询优化器的工作过程可抽象成一组树重写操作的序列，优化器的开发者只需要给定重写规则，即可由程序产生相应的优化器源代码。

1.3 树重写语言 TRL 的特性

本文给出的树重写语言 TRL 即用来描述此类规则，TRL 的编译器会自动将规则文件翻译成对应的优化器源代码。使用 TRL 构造查询优化器，可以使得开发和维护人员可以在较高的抽象层次进行工作。与其它抽象屏障一样，TRL 增加查询优化器的开发效率，减少错误，并降低开发难度和维护成本。此外，它也为构造“自适应查询优化器”提供了一种优雅的解决方案。

树重写语言中较具代表性的有 XSLT[13]和 TXL 程序设计语言[12]。XSLT 被设计用于 XML 树的重写，它采用了显示的规则应用语义和一遍遍历的执行模式。由于 SQL 的优化规则常会迭代多次，因此 XSLT 的执行模式不便于优化过程的表达。TXL 程序设计语言使用了基于发现的树重写策略，然而 TXL 对树节点的匹配仅限于类型匹配和较简单的结构匹配，并要求完全符合事先定义的推导模式；另一方面，在早前对基于规则引擎的查询优化器的研究中，Graefe[5]发现优化规则会出现对子树的进行完全匹配的需求。TXL 这两点上无法满足 SQL 优化规则模式匹配的丰富性需求。此外，TXL 的规则应用更倾向于构造规约系统，其规则的替换部分无法直接以描述性的方式构造复杂节点，这在某种程度上限制了规则的表达能力。

鉴于此，本文提出的树重写语言 TRL 在 TXL 语言和 XSLT 语言的基础上做出了改进以适应构造 SQL 语言计划器和优化器的需求，包括：

1）灵活的模式匹配机制，可匹配递归结构，允许匹配节点的部分特征，模式中可描述多节点之间的关系；

2）采用描述性为主的节点构造方式，允许以类似 XML 描述的方式直接构造节点，描述结构中也允许以编程的方式生成节点；

3）提供多样的规则应用模式，允许程序员指定搜索匹配的作用域，可以搜索整棵子树也可以仅限于当前节点；

4）提供表操作（List Processing）原语，方便对节点列表的处理；

5）提供内置的 XPath[15]表达式支持方便节点的筛选；

6）提供外部函数接口，允许 TRL 与其它语言的互操作；

7）不限定树的输入输出格式，开发者可以编写自己的前端与后端适配程序以便于规则引擎与上下文模块的无缝集成。

1.4 论文的研究内容及组织结构

基于规则的查询优化引擎本身包含较多内容，涉及模块也具有较高的复杂性，因此本文将仅着重讨论有关树重写规则语言的设计和实现。有关代价估计和搜索算法的内容不在本文讨论的范围内。

本文第二章讨论了基于规则的查询优化以及树重写语言的相关工作，第三章中给出了树重写语言 TRL 的设计细节和程序片断示例，第四章讨论了 TRL 语言的具体实现方案，第五章对 TRL 语言的发展做出了总结和展望，并讨论了如何运用 TRL 语言构造并行数据库的查询处理程序。

第二章　相关工作

2.1 Starburst 规则引擎

IBM 公司的 Starburst 项目[7]是早期规则引擎的代表。Starburst 采用了查询图模型（Query Graph Model，以下简称为 QGM）作为 SQL 语句的内部表示。Starburst 的查询优化分为查询重写阶段（Query Rewrite Phase）和计划优化（Plan Optimization）两个阶段。在查询重写阶段，优化器根据一组给定的规则对 QGM 进行变换，QGM 变换的等价性和正确性由规则保证。规则之间可以进行分组和顺序调整，以实现需要的优化步骤。Starburst 对规则使用启发式的前向推理模式，其产生的结果并不保证最优化。在计划优化阶段，Starburst 采用了一种类似上下文无关文法的产生式语言对逻辑操作符和物理操作符的实现关系进行描述：一个逻辑操作符的某个物理实现对应一个产生式，整个查询计划的物理实现被看作一次推导（derivation）。最终，优化器对所有产生的物理查询计划进行自底向上的代价估计并择优。

2.2 Cascades 规则引擎

Cascades 规则引擎衍生[5]自它的前身 Volcano 和 EXODUS。与 Starburst 不同，Cascades 中的规则不进行分组，也不再区分两个优化阶段。在优化过程中，每次迭代通过 promise 函数来确定本次变换的规则。Cascades 中的规则分为变换规则（Transformation Rules）和实现规则（Implementation Rules），变换规则仅进行代数变换，实现规则将代数表达式替换成物理的操作符树。Cascades 使用一种记忆（memoization）机制进行查询计划的枚举，当引擎发现子树已经被优化过，则直接从历史表中获取结果，否则进行规则应用。

2.3 XSLT 语言

XSLT[13]作为 XSL 的一部分被设计用于 XML 文档的变换和重新生成。XSLT 的设计初衷是为了满足 XSL 中所需要的变换能力，而非普遍意义的 XML 重写语言。XSLT 中的变换被定义为一种规则，它描述了如何根据源树生成结果树的过程。每条规则包含模式（Pattern）和模板（Template）两部分：模式用来对源树的节点进行匹配，模板用来产生结果树。在产生结果树的过程中，源树中的元素可以被筛选、重排，任意复杂的新节点可以被添加至结果树中。XSLT 采取显式的规则应用模式，只有显式调用 apply 函数时才会触发相应的模式匹配，另一

方面，只有被 select 函数选中的节点才会被处理。在搜索可用规则的过程中，有可能会出现多条规则同时匹配的状况。XSLT 会采用（可由程序员设置的）优先级的方法进行规则选择，若仍无法解决冲突则抛出异常。

在模板实例化的过程中，XSLT 总是针对当前节点进行操作。某些操作会临时改变当前节点，在操作完成后，当前节点会恢复到操作执行之前的引用。XSLT 使用 XPath 语言进行节点筛选，以方便条件处理和文本生成。XSLT 还提供了 foreach 原语用来迭代地构造节点，choose 原语用来选择性构造节点，以及 copy 原语用来复制节点或属性。

2.4 TXL 程序设计语言

TXL 程序设计语言[12]将程序看作“重写”的过程。每个 TXL 程序的执行分为解析（Parse）、变换（Transform）和反解析（Unparse）三个阶段。在解析阶段，解析器接受完整的输入文本，根据 TXL 程序的文法对其进行解析并产生完整的语法树。在变换阶段，TXL 将语法树作为输入，经过一系列变换生成对应的输出树。在最后的反解析阶段，TXL 反解析生成的语法树生成输出文本。

TXL 程序使用一种接近于扩展巴克斯-诺尔范式（Extended Backus-Nauer Form）的记法对输入文本的文法进行描述。内置的解析程序采用了完全的回朔机制，能够处理大部分的上下文无关文法，包括左递归形式以及歧义性文法。

在产生语法树之后，TXL 根据给定的规则对树进行模式匹配和变换。TXL 的变换分为变换函数和变换规则两种。变换函数根据输入的树节点进行直接匹配，若匹配不成功则程序终止。变换规则会试图搜索树（包括其孩子节点）中所有的模式匹配并应用规则，直到整棵树中不存在新的匹配为止。TXL 允许使用 where 条件字句对规则的匹配加以限定。在规则的执行过程中，程序员可以显式的使用 construct 原语构造新的节点，以及使用 deconstruct 原语对节点的子结构进行模式匹配和命名。

第三章　　TRL 语言的设计

3.1　概览

作为一种树重写语言，TRL 程序执行过程可被看作为一系列树到树的变换。每次变换都将依据某条规则进行，称作规则的一次应用。如图 3-1 所示，每条规则会按照一定模式对输入进行匹配，若匹配成功则触发变换。通常，程序会尝试对输入树进行迭代匹配，直到所有规则都不再适用为止（有关终止性的问题将在 3.7 节中更详细地讨论）。

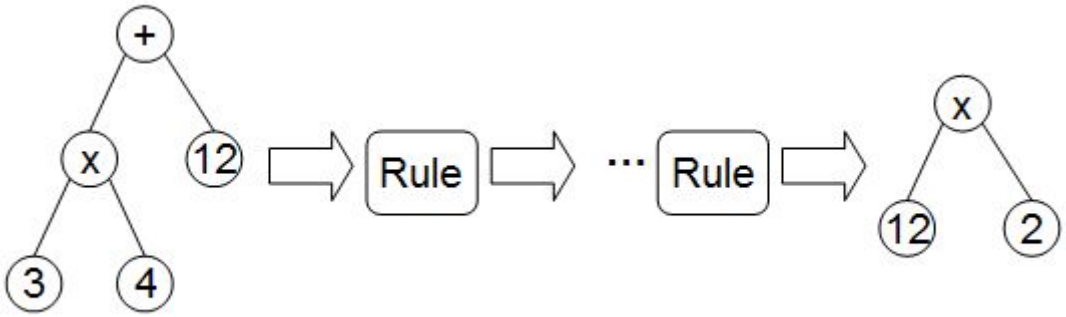

图3-1　TRL程序的执行过程

递归、分支、模式匹配和基本的表达式运算在 TRL 语言中得到了原生支持，因此 TRL 可以用于算法实现。与传统的命令式程序设计语言不同，TRL 是一种纯的（Pure）语言。TRL 语言中不支持赋值操作，所有的变量一旦创建则无法再做任何修改。由于实现复杂性的考虑，当前版本的 TRL 语言没有支持高阶函数和 lambda 表达式。

语言的类型系统内置了五种基本类型：整型（Integer）、浮点型（Float）、字符串（String）、列表（List）和树节点（Node）。节点类型的对象可以拥有属性（Attribute），节点对象的属性可在运行时产生，也可以在树重写过程中被引用。TRL 语言是动态类型化的（Dynamically-typed），然而这一点并不影响 TRL 的强类型特征：所有的函数在被调用时都会做动态类型检查以保证类型安全，在类型强制失败发生时 TRL 运行时将会抛出异常。

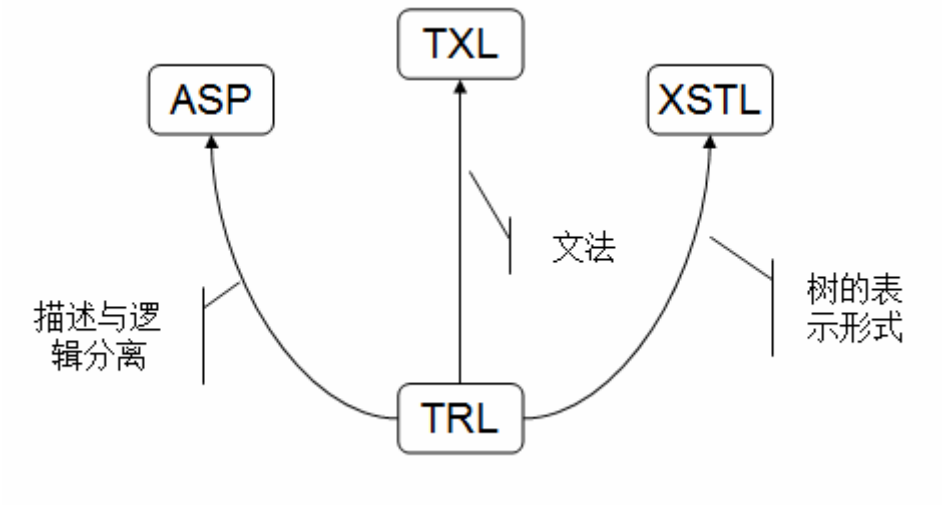

图3-2　TRL语言文法风格承袭

如图 3-2 所示，TRL 语言从 TXL 语言处继承了主要的文法结构：每个 TRL 程序由一组规则定义构成。TRL 借鉴了 XSLT 语言的树形描述，采用标签对的记法表示树结构的层次关系，并（部分地）支持 XPath 风格的表达式以方便节点筛选。此外，TRL 还采纳了 ASP 语言中分离描述性标签和逻辑代码的编程风格。

3.2　规则

规则（Rule）是 TRL 语言的最基本构成单元，每条规则由模式匹配与树重写两个部分定义。模式匹配描述了规则适用的条件，有时一个模式匹配中可能包含多个节点，其中有且仅有一个“主匹配节点（Main Node）”。当匹配成功时，TRL 会将替换结构“重写”到“主匹配节点”原先在树中的位置上。

```
rule AreaOfCircle
   match
      <float r>
   replace
      <Area>
        <script>
          value = 3.14 * r * r
        </script>
      </Area>
rule
```

图3-3　AreaOfCircle程序

图 3-3 中，规则 AreaOfCircle 描述了圆面积的计算方法：它将浮点类型的节点替换为 Area 类型的节点，产生并计算其 value 属性。关键字“match”后缩进的代码块即为规则的模式匹配部分，它匹配所有的浮点类型的节点并对其应用规则。注意到，程序中使用了“r”对匹配的节点命名，称作一个“绑定变量（Binding Variable）”，它将在树重写部分中被引用到，节点 r 也是本条规则的“主匹配节点”。关键字“replace”之后的代码块给出了替换规则。与 XML 语言的文法相似，TRL 语言也使用标签对来表示树节点的层次结构。图 3-3 中，“<Area>…</Area>”标签对将会产生一个 Area 类型的节点，“<script>…</script>”标签对中包含的代码计算了属性“value”，并将它添加到“上下文节点（Context Node）”的属性表当中（此时的上下文节点是外层的 Area 节点）。

规则中可包含条件字句对匹配进行限制。条件字句中允许构造任意复杂的谓词（以及谓词表达式），仅当谓词为真时匹配才会成功。当需要构造较复杂的表达式时，TRL 提供了“let”子句用来构造局部变量。与局部变量一样，绑定变量也可以在树重写部分中被引用。

```
rule AreaOfTriangle
   match
      <Triangle t>
   let
      a2 = t.a * t.a
      b2 = t.b * t.b
      c2 = t.c * t.c
   where
      c2 = b2 + a2
   replace
      <Area>
         <script>
           value = a * b / 2
         </script>
      </Area>
rule
```

图3-4　AreaOfTriangle程序

图 3-4 中的程序计算了直角三角形的面积。“let”子句中，程序分别计算了三角形三条边的平方值并声明为局部变量。接下来的“where”子句限制了规则的适用范围：仅当匹配的三角形为直角三角形时（谓词为真）才会触发规则应用，

否则替换结构中的计算公式是无效的。

3.3 模式匹配

模式匹配是树重写语言 TRL 的重要特性之一，TRL 提供的模式匹配策略包括类型匹配和结构匹配两种。由于 TRL 目前还没有加入静态类型声明和子类型支持，节点对象的类型声明是完全开放的，程序员可以声明任意类型的标签对，TRL 将类型等价看作为名字等价：两个类型被认为相等，当且仅当它们被声明为相同的类型。通常，在声明类型不同时，两个对象的类型被认为是不兼容的，即使它们可能拥有完全相同的属性和子节点结构。特别的，TRL 提供了抽象基类型“node”和“any”以方便对某些特殊模式的表达需求。“node”类型会匹配所有的节点类型，“any”类型则可匹配任何类型，包括前文中提到的所有基本类型。

对于需要子类型的场合，TRL 提供了“类型表达式”作为轻量级的支持。类型表达式允许程序员对多个类型限定式进行逻辑组合，包括：

1）类型合取：对于类型限定式 t 和 s，定义 t & s 为同时满足限定式 t 和 s 的所有类型集合；

2）类型析取：对于类型限定式 t 和 s，定义 t | s 为满足限定式 t 或者 s 的所有类型集合；

3）类型求补：对于类型限定式 t，定义~t 为不满足 t 的所有类型集合。

```
1) <Triangle & RegularPolygon t>
2) <Triangle | Rectangle | Pentagon s>
3) <Triangle & ~IsocelesTriangle & ~EquiTriangle t>
4) <Rectangle & ~Square r>
5) <Rectangle | Rhombus s>
```

图3-5　类型表达式示例

图 3-5 给出了一组类型匹配的示例：1）匹配同时属于三角形和正多边形的节点（等边三角形）t；2）匹配边数为 3、4 或 5 的多边形节点 s；3）匹配非等腰、非等边的三角形节点 t（非等边的类型限定是冗余的）；4）匹配长和宽不等的矩形节点 r；5）匹配同时是矩形和菱形（正方形）的节点 s。

TRL 语言的结构匹配特性为描述树节点的层次结构特征，以及节点之间的关系提供了支持。在 TRL 程序中可以通过“with”语句块附加对匹配节点的限制，“with”语句块包含对该节点的邻近节点的存在性断言。对于含有“with”语句快的模式，匹配成功当且仅当“with”语句块中的全部存在性断言均得到满足。

“with”语句块的嵌套层数不受限制，由此可构造出匹配多层子树的模式。

语言支持的节点关系断言包括：

1）父关系：指定节点为该节点的父节点，使用“parent”修饰符。

2）子关系：指定节点为该节点的子节点，使用“child”修饰符。

3）祖先关系：指定节点为该节点的祖先节点，使用“ancestor”修饰符。

4）后代关系：指定节点为该节点的后代节点，使用“descendant”修饰符。

5）兄弟关系：指定节点与该节点拥有同一父节点，使用“sibling”修饰符。

语言支持的存在性断言包括：

1）存在：存在对于上下文节点具有指定关系以及模式的节点，使用“exist”修饰符。

2）唯一：存在唯一的对于上下文节点具有指定关系以及模式的节点，使用“unique” 修饰符。

3）多个：存在多于一个对于上下文节点具有指定关系以及模式的节点，使用“multiple” 修饰符。

4）不存在：不存在对于上下文节点具有指定关系以及模式的节点，使用“none” 修饰符。

```
<Bookshelf b> with
   exist parent <Library l>
   exist child <Textbook t>
   unique child <Dictionary d>
   none child <Apple>
   <Drawer dr> with
      <Notebook n>
      <Postcard p>
      more <CD>
   end
end
```

图3-6　结构模式匹配示例

图 3-6 中给出了一个结构模式匹配的例子。该模式将匹配 Bookshelf 类型节点 b，该节点还需要：1）存在 Library 类型的父节点 l；2）存在 Textbook 类型的子节点 t；3）存在唯一的 Dictionary 类型子节点 d；3）不存在 Apple 类型的子节点；4）存在 Drawer 类型的子节点 dr，dr 拥有 Notebook 类型子节点 n，Postcard 类型子节点 p，以及多个 CD 类型的子节点。“exist”和“child”分别为默认情况下的存在性修饰符合关系修饰符（图 3-6 中的 Drawer 节点、Notebook 节点和

Postcard 节点），被“more”和“none”关键字修饰的节点无法被命名（图 3-6 中的 Apple 节点和 CD 节点）。

3.4 树重写过程

树重写过程发生在模式匹配成功后。在树重写过程中，“主匹配节点”将被重写成替换结构。替换结构中对树的描述采用了类似 XML 语言的标签对记法，标签中的标识符代表该节点的类型。在需要以可编程的方式构造节点时，TRL 允许将逻辑代码作为一组“<script>…</script>”标签对的内容，嵌入到上下文中。许多与节点构造相关的内部函数都要求在一定的上下文中生效，错误的上下文环境将引发异常。此外，TRL 还提供了一组控制结构以支持选择性或迭代地构造节点。

用于支持节点构造的内部函数包括：

1）函数 yield_child(object o)：将对象 o 作为当前上下文节点的子节点，o 的子节点将会被递归拷贝。如果当前的上下文节点不存在，则 o 将被作为根节点。

2）函数 yield_attr(attribute a)：将属性对象 a 拷贝，并添加到当前节点的属性列表中。

3）函数 copy_child(node n)：将节点 n 的所有子节点（递归地）拷贝成当前节点的子节点。如果当前上下文节点不存在，程序将抛出异常。

4）函数 copy_attr(node n)：将节点 n 的所有属性拷贝至当前节点。如果当前上下文节点不存在，程序将抛出异常。

```
rule AddBook
   match
      <Bookshelf bs>
   replace
      <Bookshelf>
         <Book> /*NewBook*/ </Book>
         <script> copy_attr(bs) </script>
         <script> copy_child(bs)</script>
      </Bookshelf>
rule
```

图3-7　节点的扩展示例

节点属性的构造可直接使用“<属性名> = <表达式>”的记法，表达式中被

引用到的对象将被拷贝。在出现属性名重复时（例如从多个节点拷贝属性并发生），程序将抛出运行时异常。

图 3-7 中的程序给出了一个扩展节点内容的示例。规则 AddBook 将匹配 Bookshelf 类型的节点 bs，并将为它新增一个 Book 节点。bs 节点的所有属性和子节点都将被拷贝到新产生的 Bookshelf 节点当中。

与大部分程序设计语言相似，控制结构“if-else”和“cond-case”用于选择性构造节点构造。其中“if-else”结构的 else 部分可被省略，“cond-case”可构造对多个分支的选择。控制结构“for-in”和“for-from-to”可用于迭代地构造节点。“for-in”结构遍历给定的列表类型，并逐个对每个对象执行操作。“for-from-to”结构可用于执行指定步数的迭代过程。

```
rule AddBookAndUpdateCount
   match
      <Bookshelf bs>
   params
      <Book b>
   replace
      <Bookshelf>
         <script> yield_child(b) </script>
         <script> copy_child(bs) </script>
         <script>
           for attr in bs/@
               if attr.name != “count”
                  yield_attr(attr)
            count = bs.count + 1
         </script>
      </Bookshelf>
rule
```

图3-8　节点的扩展及更新示例

图 3-8 中的程序对图 3-7 的程序做了扩充。规则 AddBookAndUpdateCount 在增加一本书的同时将 bs 节点的“count”属性加一。因此程序使用了“for-in”结构遍历了 bs 节点的属性，对除“count”以外的属性进行拷贝。最后，程序为新的 Bookshelf 节点生成重新计算的“count”属性。此外，图 3-8 中的程序引入了规则参数，关键字“params”之后声明了 Book 类型的参数 b。随后在替换结

构中 b 被作为 yield_child 方法的参数被引用，节点 b 将被拷贝成为上下文节点（Bookshelf）的子节点。

3.5 表达式和表操作

TRL 语言提供了丰富的表达式语法支持。表达式可出现在“where”条件子句、选择语句的条件、属性构造等结构中。此外，TRL 语言部分支持了 XPath 表达式文法以方便节点和属性的选择。

表达式按照（从低到高）优先级可分为以下层次：

1）逻辑表达式：谓词之间的逻辑或（or）、逻辑与（and）和逻辑非（not）操作。

2）类型断言表达式：判断对象是否为指定类型。

3）比较表达式：判断对象之间的等价关系和大小关系，包括相等（=）、不等（!=）、小于（<）、大于（>）、不小于（>=）、不大于（<=）。

3）算术表达式：实现算数乘（*）、算数除（/）、算数求模（mod）、算数加（+）、算数减（-）的计算。

4）XPath 表达式：实现常见的 XPath 节点选择操作，包括子节点选择、属性选择，轴（axis）操作等。

5）属性访问表达式：访问对象的指定属性。当对象属性不存在时，程序将抛出运行时异常。

由于树重写操作常常需要对一组节点作批量操作，TRL 语言提供了一组表操作（List Processing）函数以支持此类操作：

1）表构造函数 list(object o1, object o2, …)：将对象 o1，o2…构造成一张新表并返回，函数可接受可变长度参数。

2）左连接函数 list_cons(object o, list l)：将对象 o 连接在表 l 的头部，并返回生成的新表。

3）表头寒暑 list_head(list l)：返回表 l 头部的对象。

4）表尾函数 list_tail(list l)：返回从表 l 第二个元素开始的新表。表 l 只有一个元素则返回空表，表 l 为空时程序将抛出运行时异常。

3）表连接函数 list_append(list l1, list l2)：返回表 l1 和表 l2 连接后的新表。

4）表长度函数 list_length(list l)：返回表 l 的长度。

5）表空函数 list_empty(list l)：返回表 l 是否为空。

6）表索引函数 list_at(list l, int i)：返回表 l 的第 i 个元素。在表的长度小于 i 时，程序将抛出运行时异常。

3.6 执行和终止

上文介绍了 TRL 程序中规则的声明结构以及语义，然而，程序的真正执行开始于一次“动作（Action）”。TRL 支持“调用（Invoke）”和“变换（Transform）”两种动作，其具有不同的执行语义。

“调用”的效果相当于面向对象程序设计语言中的函数调用。与面向对象中的方法调用类似，一次调用需要由某个对象发起，并指定被调用的规则。TRL 会将该对象作为被调用规则的主匹配节点进行模式匹配，若匹配失败则抛出运行时异常。调用也可以出现在“<script>…</script>”中以实现规则的递归调用。

```
rule Factorial
   match
      <int i>
   replace
      <script>
         if i > 1
            yield_child(i * Factorial(i - 1))
         else
            yield_child(1)
      </script>
rule

Factorial(5)
```

图3-9 传统风格的阶乘计算程序

图 3-9 给一个出了“传统风格”的阶乘计算程序。程序的最后一行即为一次“调用”动作，此处的 Factorial 与普通意义上的函数相同，函数的第一个参数为调用的发起对象。该程序中，整数常量 5 将作为 Factorial 规则的主匹配对象触发模式匹配，规则的替换结构中对自身进行了递归调用。

与“调用”动作不同，“变换”动作将搜索以发起对象为根节点的整棵子树，对于所有发现匹配的结构进行规则应用，直到无法再找出可匹配的模式为止。可以看出，使用变换动作有可能构造出永远无法终止的程序。例如，若将图 3-9 中最后一行对 Factorial 规则的调用动作更改为变换动作，该程序将不会终止：Factorial 规则可匹配任何的整数类型节点，并将它映射成一个新的整数类型节点，从而程序将反复进行此类整数域上的变换。然而，变换动作的语义在某些场合下具有不可替代的表达能力。例如，在查询计划树的优化过程中，优化规则常

常被用来描述代数变换公里，程序需要搜索整棵查询计划树并“找出”相匹配的结构并应用规则。此时即可使用变换动作实现此类“发现式”的树重写过程。

```
rule Factorial
   match
      <int i>
   replace
      <MidResult>
         <script> Result = 1  Factor = i </script>
      </MidResult>
rule

rule Factorial
   match
      <MidResult m>
   where
      m.Factor > 1
   replace
      <MidResult>
         <script>
            Result = m.Result  * m.Factor
            Factor = m.Factor  - 1
         </script>
      </MidResult>
rule

transform 5 using Factorial
```

图3-10 树重写风格的阶乘计算程序

图 3-10 中的程序采用树重写的风格重新实现了阶乘计算。程序首先将整数类型的节点转化为 MidResult 类型，该类型代表了阶乘计算多步变换的中间结果。程序维护了不变式：Result 与 Factor 的阶乘的乘积等于输入数的阶乘，通过每次重写 MidResult 节点逐步减小 Factor 的值，直到 Factor > 1 的条件为假。程序最后一行的“transform”结构即“变换动作”，它以整数型节点 5 为根节点搜索 Factorial 规则适用的模式并应用。该程序中，第一条 Factorial 规则将首先被应用，之后第二条 Factorial 规则被反复应用 5 次后程序终止。

第四章　TRL 语言的实现

4.1　系统构成

作为一个完整的编程系统，TRL 语言由编译器、解释器和运行时三个部分构成。编译器负责完成规则语言到目标语言的翻译，当前的 TRL 实现使用 C 语言作为翻译的目标语言。C 语言本身具有良好的表达能力，高效的 C 语言实现版本很多也易于获得。因此，将 C 作为目标语言再进行二次编译，系统可以利用其高效和稳定的优点，同时也降低了自身的实现难度。

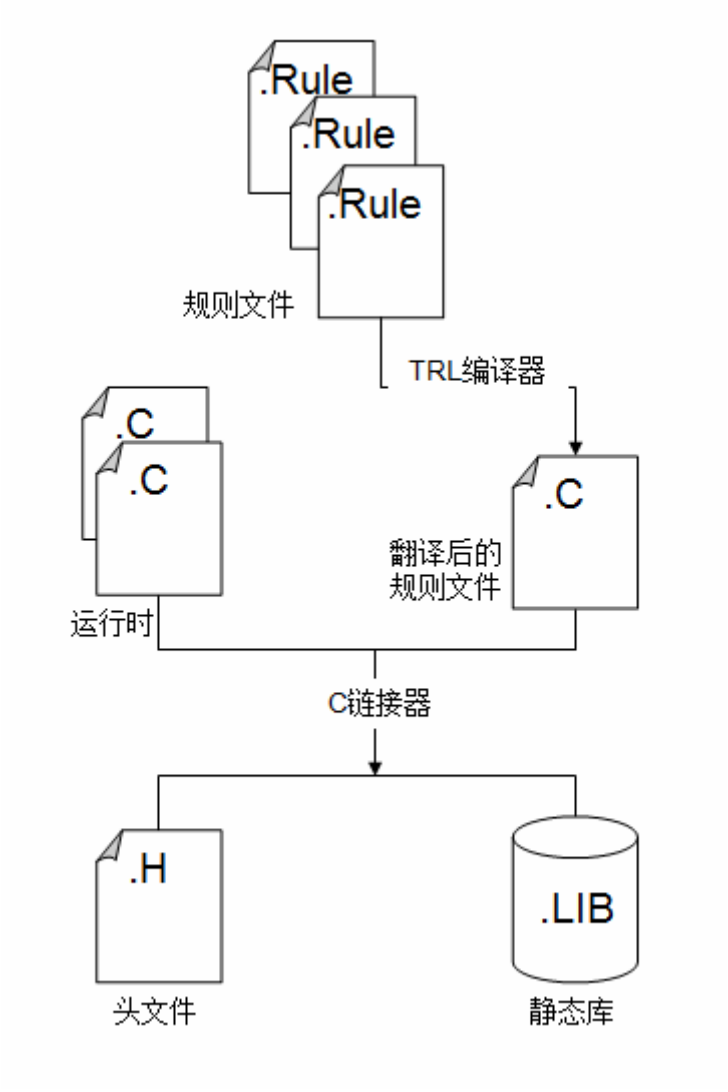

图4-1 TRL程序的编译

语言的运行时部分以 C 语言程序源代码的形式提供。TRL 程序的编译过程如图 4-1 所示：在 TRL 的编译器将规则文件翻译成 C 语言程序之后，系统将调用 C 语言的编译器和链接器将运行时代码与规则翻译生成的结果做静态链接，

生成静态库文件（LIB）和头文件（.h）。为了方便调试，实现提供了 TRL 语言的解释器。在解释执行模式，解释器可单步执行程序，追踪每次应用的规则，打印绑定变量的值以及当前树的状态。

4.2 树操作接口

TRL 定义了一组树操作接口用于构造前端和后端的适配程序，所有实现了这组接口的树表示都可以被 TRL 接受为输入，如图 4-2 所示。对于常见的树表示形式，如 XML 文件、JSON 文件等，通常可基于其现有的解析器较容易地实现 TRL 的树操作接口。树操作接口也可以被直接实现为内存中的数据结构，此时无需中间表示形式的 I/O 操作即可以将 TRL 生成的规则引擎无缝地集成入上下文模块中。

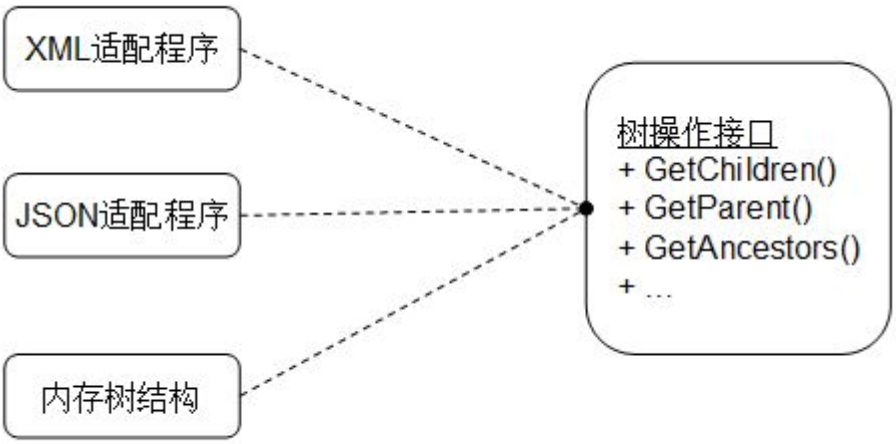

图4-2 TRL的树操作接口

TRL 的树操作接口包括：

1）Children(node n)：返回节点 n 的子节点列表。如果 n 为叶子节点，则返回空表。

2）ChildrenCount(node n)：返回节点 n 的子节点个数。

3）Parent(node n)：返回节点 n 的父节点。如果 n 为根节点，则返回空。

4）Ancestors(node n)：返回节点 n 的祖先节点列表。

5）Descendents(node n)：返回节点 n 的子孙节点列表。

6）Type(node n)：返回节点 n 的类型（字符串）。

7）Value(node n)：返回节点 n 的值。该函数仅对基本类型的节点有效，包括整数类型，浮点类型，字符串类型和列表类型。若应用于节点类型，函数返回空值。

4.3 外部函数

在实现树操作接口的过程中，程序会引用到 TRL 的类型结构定义。当前的 TRL 实现使用了 C 语言中的结构体对此类结构进行定义，因此，程序与 C 语言的互操作性较好。其它需要与 TRL 相互操作的语言应当具有与 C 语言一致的对象内存布局，或具有控制内存布局的能力，使用该语言中对原本的 C 结构进行重新封装和解释（保证二进制兼容性）后即可实现互操作。

在 TRL 语言中，使用“$”符号作为函数名的前缀即可调用一个外部函数，如图 4-3 所示。

```
rule Estimate
   match
      <TableScanPlan ts>
   replace
      <script>
         yield_child(&EstimateSize(ts))
      </script>
end
```

图 4-3 TRL 中的外部函数调用

图 4-3 中的外部函数调用&Estimate(ts)将被翻译成以下 C 语言代码片断：

```
extern TRL_OBJECT EstimateSize(TRL_OBJECT);
   ...
TRL_OBJECT obj = GetBindingVariable(“ts”);
TRL_OBJECT res = EstimateSize(obj);
yield_child(res);
```

规则中的$EstimateSize 函数调用被对应地翻译成为一条 C 语言中的 EstimateSize 函数调用，其原型被声明为：接受一个 TRL_OBJECT 类型的参数并返回 TRL_OBJECT。接下来 C 语言的链接器会对 EstimateSize 函数的外部实现进行链接，程序在执行到该函数时即会跳转至其外部实现。

TRL 语言要求外部函数的调用必须是无副作用的，即该函数必须是只读类型。任何对参数的修改均可能产生未定义的程序行为。

第五章　构造查询处理器

5.1　从 SQL 语法树到逻辑查询树

在不考虑优化的情况下，将输入的 SQL 抽象语法树重写为对应的逻辑查询树是一个机械过程，该过程可通过一些相互独立的规则进行描述。通常逻辑查询树以自底向上的方式进行构造，主要规则包括：

1）将表引用重写为表扫描节点（Table Scan）。

2）将多个表扫描节点重写为笛卡儿积（Product）节点。

3）将条件子句节点重写为选择（Selection）节点，并将拷贝笛卡儿积节点作为子节点。

4）将分组子句节点重写为聚积（Aggregation）节点，并拷贝选择节点作为其子节点。

5）将输出字段列表重写为投影（Projection）节点，并拷贝聚积节点作为其子节点。

6）消除冗余节点。

使用规则对翻译模式进行抽象使得程序获得了更清晰的逻辑以及正确性保证：规则的前置条件（匹配模式）和后置条件（替换结构）是显而易见的，程序员可根据这些条件更容易地对翻译程序的行为进行推理和断言。此外，由于规则的上下文无关特性，使用规则引擎构造的翻译程序可自动地处理嵌套查询。在重写过程中，程序还可添加错误报告、符号表管理等例程。

5.2　从逻辑查询树到物理查询树

与从 SQL 语法树到逻辑查询树的过程不同，将逻辑查询树重写为物理查询树的操作虽然可以采用朴素实现，但朴素方案的代价在许多的应用环境中无法被应用需求所接受。因此，逻辑查询树的优化是查询处理器中不可缺少的步骤。

逻辑查询树的优化可分为两类：

1）对逻辑查询树本身进行等价的代数变换。

2）适当地选择逻辑操作符（Logical Operator）的物理实现方案。

要获得最优逻辑查询树的最优物理实现，需要对所有可能的实现方案进行枚举并分别做代价估计。即使在代价估计准确的情形下，枚举物理查询树的计算复杂度也会相当可观。例如，若要获得对 4 张表连接操作的最佳实现方案，仅仅考虑左（或右）深连接树即有 4 的阶乘种情况，对于每种逻辑查询树的每个连接操作符，其物理实现也会有多种选择（如哈希连接、归并连接、嵌套循环连接等）。

出于复杂性考虑，可以采用启发式策略对规则进行应用。常见的启发式规则有：

1）谓词下推：若条件子句中的谓词 p 仅引用了表 t 的字段，则可将谓词 p 下推至表 t 的扫描操作之前。该项优化可减少表扫描操作产生的元组数量，以减少网络以及磁盘的 I/O 次数。

2）列引用下推：将所有被引用的字段下推至表扫描操作之前，消除不必要的列扫描操作。

3）子查询消除：将特定形式的子查询操作变换为连接操作或条件子句。

4）物理实现选择：启发式地选择更快速的操作符物理实现。例如，在发现连接操作的某一张表可以放于内存中时，选择一遍操作的连接算法。

启发式的规则应用策略有可能失去最优的实现方案，但恰当的规则涉及往往可在实践中获得良好的优化效果。

第六章　总结和展望

6.1　问题和改进

在使用 TRL 构造查询处理程序的过程中，仍存在着诸多有待解决的问题以及可改进的空间：

1）模式匹配算法：TRL 使用了朴素的模式匹配算法。算法首先遍历整棵树并枚举所有类型匹配的节点，再根据结构进行筛选。在节点具有多个类型相同的子节点时，算法可能需要遍历指数空间，这在模式复杂时会对性能有较大影响。

2）自动化树变换：使用 TRL 构造查询优化器时，规则应用策略是启发式的。启发式的方法一方面阻碍了优化器获得最优实现方案，另一方面，它要求规则的书写者为每条规则“设计”一个启发式的应用条件。此类条件与规则的耦合，对于优化程序的设计者提出了更高的“经验”和“技巧”上的要求。理想状况下，优化程序的设计者应能够仅仅书写代数变换规则，而无需关心其应用的触发条件，从而使树的变换和枚举过程完全自动化。或者，由语言提供更清晰的抽象机制描述规则的应用策略，实现半自动化的树变换。

3）静态类型系统：当前 TRL 语言的类型系统不支持静态声明，这使得程序无法在编译时期捕获许多类型错误。然而，引入静态的类型系统可能对目前的语言实现造成较大影响。例如，为所有的节点添加类似“构造函数”的例程，以及对节点构造脚本进行类型检查等等。

6.2　展望

并行数据库查询处理器构造是一个同时具有较高工程复杂性和计算复杂性的问题，本文提出了基于树重写语言 TRL 的解决方案以应对工程复杂性方面带来的挑战。在语言的设计和实现过程中，查询优化器中现有的各种例程和逻辑片段一直被作为衡量语言设计取舍的“试金石”。然而在新的需求出现时，语言是否能够以自身现有的抽象机制对其进行描述和控制是其评判设计优劣另一个重要准则，它决定着语言是否可以经受时间的考验。并行数据库的查询优化器为这种测验提供了一个良好的平台，并行数据库的实现包含着大量需考虑数据分布、计算分布等因素的性能调优和动态优化，如何对该过程进行规则建模，以及规则引擎本身的设计均是值得进一步探讨的课题。

参考文献

[1]David DeWitt, Jim Gray. Parallel database systems: the future of high performance database systems. Communications of the ACM, June 1992. Volume 35, Issue 6.

[2]Chaudhuri Surajit. An Overview of Query Optimization in Relational Systems. Proceedings of the ACM Symposium on Principles of Database Systems, 1998. pp. pages 34–43.

[3]Graefe G. Query Evaluation Techniques for Large Databases. In ACM Computing Surveys: Vol 25, No 2., June 1993.

[4]Graefe G., McKenna W.J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proc. of the IEEE Conference on Data Engineering, Vienna, 1993.

[5]Graefe G. The Cascades Framework for Query Optimization. In Data Engineering Bulletin. Sept. 1995.

[6]Graefe, G., Dewitt D.J. The Exodus Optimizer Generator. In Proc. of ACM SIGMOD, San Francisco, 1987.

[7]Haas L., Freytag J.C., Lohman G.M., Pirahesh H. Extensible Query Processing in Starburst. In Proc. of ACM SIGMOD, Portland, 1989.

[8]Lohman G.M. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In Proc. of ACM SIGMOD, 1988.

[9]Poosala V., Ioannidis Y.E., Haas P.J., Shekita E.J. Improved Histograms for Selectivity Estimation of Range Predicates In Proc. of ACM SIGMOD, Montreal, 1996.

[10]Hasan W. Optimization of SQL Queries for Parallel Machines. LNCS 1182, Springer-Verlag, 1996.

[11]JE Richardson, MJ Carey. Programming constructs for database system implementation in EXODUS. Proceedings of the 1987 ACM SIGMOD Conference, 1987.

[12]The TXL Programming Language, Version 10.5. http://www.txl.ca/docs, 2007.

[13]James Clark. XSL Transformations (XSLT) Version 1.0. W3C, 1999

[14]Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. Database System Implementation. Prentice-Hall, 2000.

[15]James Clark, Steve DeRose. XML Path Language (XPath) Version 1.0. W3C November 1999.

[16]Hal Abelson, Jerry Sussman, Julie Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1984.

[17]Robin Milner, Mads Tofte, Robert Harper, David MacQueen. The Definition of Standard ML, Revised Edition. MIT Press, May 1997.

[18]Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers, Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2006.

[19]Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.

Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[20]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-580 and 583, October 1969.

致　　谢

本文的工作完成于阿里巴巴集团研发院实习期间，王轶凡先生对本文的选题立意、实际工程意义等方面给予了许多有意义的帮助和指导。在 TRL 语言的设计和实现期间，王轶凡先生和刘缙先生提出了大量宝贵的建议，与你们的讨论使我受益良多。在此，我对提供本次机会的王轶凡先生表示深深的感谢。感谢孙冰先生、林晨曦先生以及阿里巴巴集团研发院的所有工程师，让我有幸在宽松、自由又充满信任的环境下工作。

天津大学计算机学院的张坤龙老师为本文提出了中肯的意见和建议。此外，在我四年的本科学习中，张坤龙老师从研究方法和态度等方面耐心而无私地给予了我珍贵的启蒙教导。我在这里对张坤龙老师表示诚挚的感谢和深深的敬意。