WeTune: Automatic Discovery and Verification of Query Rewrite Rules

摘 要

查询重写是数据库查询处理过程中的一个基本问题，与查询优化、物理数据的独立性维护、数据集成、语义缓存、数据仓库和决策支持等问题密切相关。

有研究提出了一个自动发现新重写规则的规则生成器 WeTune，本文主要验证 WeTune 生成规则的可行性，对 WeTune 生成重写规则的每一步，都进行程序的验证运行，验证其能够达到预期目标。并运用生成的规则进行一些查询语句的查询重写，观察 WeTune 是否成功地优化了现有优化规则无法优化的查询，提高了其性能。并且在验证过程中遇到了 WeTune 的 reduce 即减少冗余规则的运行时间较长的问题，针对这一运行时间过长的问题，根据其算法减少数据集的大小，让相似规则作为一个数据集进行多次迭代运行，减少不必要的时间运行，从而减少 reduce 的运行时间。

关键词：WeTune；查询重写；数据库；重写规则发现；SQL 求解器

1 引言

1.1 选题背景

有效管理大量的数据对几乎所有的计算机应用都是至关重要的。数据库管理系统（DBMS）是一个软件系统，它允许一个或几个用户轻松有效地管理和操作大量的数据。随着应用变得越来越复杂，人们期望数据库能存储越来越多的信息。数据库的容量正在迅速增加，针对大容量数据库的数据检索效率已经成为一个主要的问题。在编写语句的时候，程序员往往只关心查询结果的正确性，而忽略了不同语句的性能会有很大的差异。在大型或复杂的数据库环境中，这种性能上的差异会更大。而以数据库为基础的网络应用一直是互联网应用的骨干，从网上购物到银行业务。对于许多网络应用，数据库查询延迟对用户体验至关重要。例如，据报道，延迟增加 500ms 可以使网站的流量减少 20%，而当网站的加载时间超过 3 秒时，用户往往会放弃网站。

一个查询优化器会为不同的系统选择不同的查询策略，这些系统的表间关系不同，表内数据的分布也不同。影响数据库性能的因素有很多，不可能定义一个万能的方法。在系统开发和维护期间在系统的开发和维护过程中，需要结合自己的经验和对系统的理解，不断调整系统的运行状况。为了达到满意的效果，必须根据需要对系统进行测试和修改。一

个好的查询计划可以成倍地提高查询性能。这并不是一个空想。查询优化处理器有望提供较佳的查询计划，所以数据库中的查询研究很重要。

查询重写将关系数据库查询转换为等效但更有效的查询，这对数据库支持的应用程序的性能至关重要。这种重写依赖于预先指定的重写规则。在现有系统中，这些重写规则是通过人工洞察发现的，并在多年中慢慢积累。现在关于查询优化的方法有很多其中比较有代表性的有全局查询处理、关系代数等价变换、语法树的变换、索引技术，另外还有基于遗传算法的查询优化处理这些都是当前流行的查询优化策略。本文主要通过研究自动发现新重写规则的规则生成器 WeTune，验证其生成规则的正确性，并与现存优化规则进行对比。发现其现有的缺陷进行优化。

1.2 选题意义

现如今关系数据库查询语句依靠手工工作来发现重写规则是不够的。查询语言丰富的特性和微妙的语义使得证明等价性和制定规则具有挑战性。因此，手写重写规则集的增长非常缓慢，并错过了许多重写机会。在 web 应用程序开发中，对象-关系-映射(ORM)框架的普遍使用使情况变得更糟。ORM 将程序员从明确构建构造 SQL 查询中解放出来，但也会导致非直观的查询，这些查询的模式逃避了人类制定的规则。为了了解错过重写的影响，WeTune 研究了 Github 上几个流行的开源 web 应用程序中的 50 个真实世界的查询。所有这些查询都被开发人员重写了，以解决他们的性能问题。即使是最新版本的 SQL Server 也未能将其中的 27 个查询(54%)重写为一个更有效的表单，这是由开发人员手工修复的。一个这样的查询会引起多达 37 秒的延迟，而它的对等重写只需要 0.3 秒。

本文研究并验证了 WeTune 规则生成器的正确性，它可以自动发现新的重写规则，而无需任何人工努力。WeTune 的灵感来自于编译器的超优化，它通过穷举搜索找到语义上等效的最优代码序列，WeTune 的目标是通过对所有潜在规则的暴力枚举，然后对生成的每个规则进行正确性检查，自动发现重写规则。在这个发现过程中，WeTune 依靠启发式过滤掉那些不太可能提高性能的规则，也就是那些重写后的查询中包含的每种类型的运算符比原始查询中更多的规则。其余的重写规则被认为是有希望的。通过使用这些规则重写真实世界的查询，并通过综合生成的数据库表来衡量重写查询的性能优势，之后可以根据经验确定这些有前途的规则的有用性。那些导致有益重写的规则是 WeTune 发现的有用规则。

WeTune 作为一个新提出的查询重写规则，对比当前的查询重写规则有很大的提升与进步，但也有一定的缺陷，它在减少冗余规则时，进行启发式搜索，迭代的进行寻找，因

为数据集过于庞大，导致用时过长，本文根据对减少冗余规则的算法的研究，进行优化改进，缩短查找时间。更快的进行 reduce。

2 相关工作

2.1 查询优化的技术

数据库查询优化从物理和逻辑层面总体上分为两大部分，涉及数据库的物理设计、逻辑设计、体系结构设计以及数据库管理系统的设计这几个方面。物理数据库设计是指建立高效的存储结构和高效的物理存储布局，它包括文件组织、存储映射算法、存储介质等方面，是提高数据库查询效率的基础。从逻辑设计的角度来看，规范化理论和模式分解理论为关系型数据库的逻辑设计提供了坚实的理论基础和有效的工具。在模式规范化和分解理论的基础上进行设计，可以在很大程度上消除数据的冗余，避免一些更新异常，提高数据库的完整性，从而提高数据库的可维护性和可靠性。

目前关于查询优化的研究主要分为两个方面，其一是外部优化，主要是利用现有的优化器，最大限度地发挥软件和硬件的潜力，提高查询性能。它针对影响查询的多种因素，覆盖了从系统分析、设计到实现的各个阶段，其中包括数据的存储与组织、SQL 语句的优化、前端开发工具的使用技巧及后台数据库的参数调整等。另一方面，对内部优化的研究是对查询优化器(Query Optimizer)的工作机制的研究。最初，查询优化不是重写查询，而是对计划进行优化。在计划优化中，经常会出现多种访问路径，只能选择一种的情况。根据选择访问路径所依据的原则，查询优化可以分为两种类型： 基于规则的查询优化和基于成本的查询优化。

2.2 自动发现查询重写规则的必要性

手动的重写规则是不够使用的，这些规则或多或少是为程序员直接编写的 SQL 查询而设计的。人类编写的查询通常遵循直观的模式，可以通过人工分析来提炼出有用的规则。然而，在现代网络开发中，程序员不再明确地构建 SQL 查询。相反，他们通常利用对象关系映射（ORM）框架，这使他们能够编写面向对象的代码来操作数据库中的内容。底层框架根据应用逻辑自动生成 SQL 查询。ORM 生成的 SQL 查询不仅对程序员来说是不透明的，而且对人类规则开发者来说也可能是反直觉的。表 1 显示了来自网络应用 GitLab[10]的两个例子，这是一个流行的开源版本管理网站。这两个 SQL 查询都是由 ORM 框架（Active Record）生成的，是运行开发者的 Ruby 代码的结果。

表 2.1　GitLab 中 ORM 框架生成的反直觉查询示例

原始查询	现有数据库最佳优化结果	理想结果（WeTune）
q0: SELECT * FROM labels WHERE id IN (SELECT id FROM labels WHERE id IN (SELECT id FROM labels WHERE project_id =10) ORDER BY title ASC)	q1: SELECT * FROM labels WHERE id IN (SELECT id FROM labels WHERE project_id =10)	q2: SELECT * FROM labels WHERE project_id =10
q3: SELECT id FROM notes WHERE type='D' AND id IN (SELECT id FROM notes WHERE commit_id =7)	Unchanged	q4: SELECT id FROM notes WHERE type='D' AND commit_id =7

第一个查询$q0$旨在选择所有 project_id 为“10 ”的 git 合并请求。具体来说，它使用一个子查询，根据标签表计算出 project_id 为“10 ”的 id 值集合。然后，它从标签表中选择所有 id 属于这组子查询值的所有行。这个查询在两个地方是违反直觉的，而且效率很低。首先，计算匹配 id 的子查询包含另一个内部子查询，这两个子查询几乎是相同的。第二，内部子查询的 ORDER BY 子句是不必要的，因为最外层的 IN 操作符将子查询 SELECT...ORDER BY 视为无序列表。理想情况下，查询中的冗余应该由查询优化器通过重写规则来识别，这样得到的优化查询类似于表 1 中的$q2$。然而，在 MySQL、PostgreSQL 和 MS SQL Server 中，只有 PostgreSQL 和 MS SQL Server 可以将查询部分重写为$q1$，其中删除了 ORDER BY 和两个子查询中的一个。

表 1 中的另一个查询（q3）从类型为“D”且列 id 为“7 ”的 notes 表中获取 id 值。对于这个查询，IN-selection 是多余的，因为（1）子查询中使用的表和外面使用的表是一样的，也就是 notes 表；（2）子查询预测的列和 IN-selection 中使用的列是一样的，也就是 notes 表的主键。因此，子查询可以被取消，查询可以被转化为简单的查询，如$q4$。不幸的

是，现有的三个数据库（MySQL、PostgreSQL 和 MS SQL Server）都错过了这样的机会，而保持查询不变。上述低效结构在人类生成的查询中不太可能出现，但在由 ORM 生成的查询中却很常见。由于 ORM 生成的查询是在不同的程序位置甚至是第三方库中运行应用程序代码的结果，开发者对潜在的冗余（如重复的子查询）和低效（如不必要的 ORDER BY）不了解。因此，开发人员很难识别和修复由此产生的性能问题。

3 本文方法

3.1 WeTune 面临的技术挑战

WeTune 的高级别方法是直截了当的。然而，为了使其发挥作用，必须解决规则枚举和验证所面临的几个技术挑战。这些挑战是查询重写所特有的，在编译器优化中并不存在。

(1) 将重写规则表示为三元组<q_{src}, q_{dest}, C>的形式以使其适合于枚举，一个规则由一对查询组成，这些查询必须是通用的，不与具体的表和列相联系。将查询用 U-expression 的形式进行书写使源查询等同于目的查询

（2）确定一个列举的重写规则是否正确，我们可以采用现有的查询等价检查器（SPES），也可以运用 SMT 解析器，根据我们规则本身的特点，创造更加实用的规则等价验证器。

3.2 WeTune 的架构

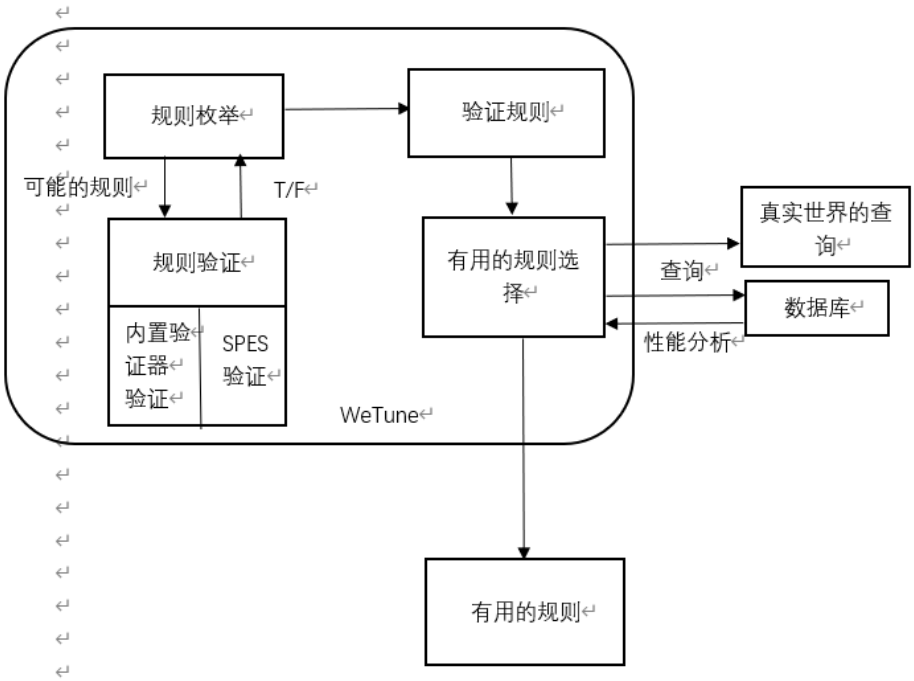


图 3.1 WeTune 的架构

WeTune主要分为两个部分，分别是对规则的枚举以及对规则的验证，由图表3.1可知，首先 WeTune 对操作符根据不同的节点进行规则的枚举，得到的规则需要进行验证相等并从中找到最宽松的约束集，验证的方法有两种，一种是用 WeTune 内置的规则验证器进行验证，另外还可以用已经存在的规则验证器 SPES 进行验证。之后将得到的已经找到的有希望的规则，与现实中的一些查询进行应用，从得到的已被验证的有希望规则中，结合现实查询的查询重写使用情况和规则应用情况，找到并得出有用的规则。得到的规则即是WeTune自动查找得到的有用规则。

3.3 针对 WeTune 的创新

在 WeTune 进行规则集优化的过程中，需要找到最宽松的规则集，去除现有规则的冗余性，在查询得到最宽松规则集的过程中，使用的算法主要是类迭代的进行搜寻，在搜寻过程中所用到的时间与规则集大小有着很大的关系，规则集如果过于庞大，在运行程序的过程中，会造成运行时间过长。

而在进行减少冗余规则的过程中，逐一减少规则，验证是否能够得到相同结果，通过此来减少冗余规则得到最宽松的规则集结果，但是在类迭代的整个数据集中，对于源计划模板与约束条件所生成的具体化规则，规则集中的其中大部分规则对其都是无法进行规则重写的，所以在运行过程中因为需要对每一条规则进行约束判断，造成不必要的时间产生。本文对此做出创新，减少数据集的大小，让相似规则作为一个数据集进行多次迭代运行，减少不必要的时间运行。

4 复现细节

4.1 WeTune 的规则枚举

4.1.1 计划模板枚举器

WeTune 将重写规则建模为三元组<q_{src}, q_{dest}, C>，其中q_{src}是源查询计划模板，q_{dest}是目的查询计划模板，C是一组约束。查询计划模板是逻辑查询计划树的一个片段，其运算符包括选择、投影等。与具体查询中的操作不同，查询计划模板中的表名、属性和谓词是符号化的。约束集C由一组谓词组成，每个谓词都描述了源查询计划模板和目的查询计划模板中的符号之间的某种关系。该规则规定，如果C中的所有约束条件得到满足，那么q_{src}和q_{dest}在语义上是等同的。给定一个 SQL 查询q，如果q中的某些片段与q_{src}相匹配，则可以用满足C的相应片段q_{dest}来替换匹配的片段。

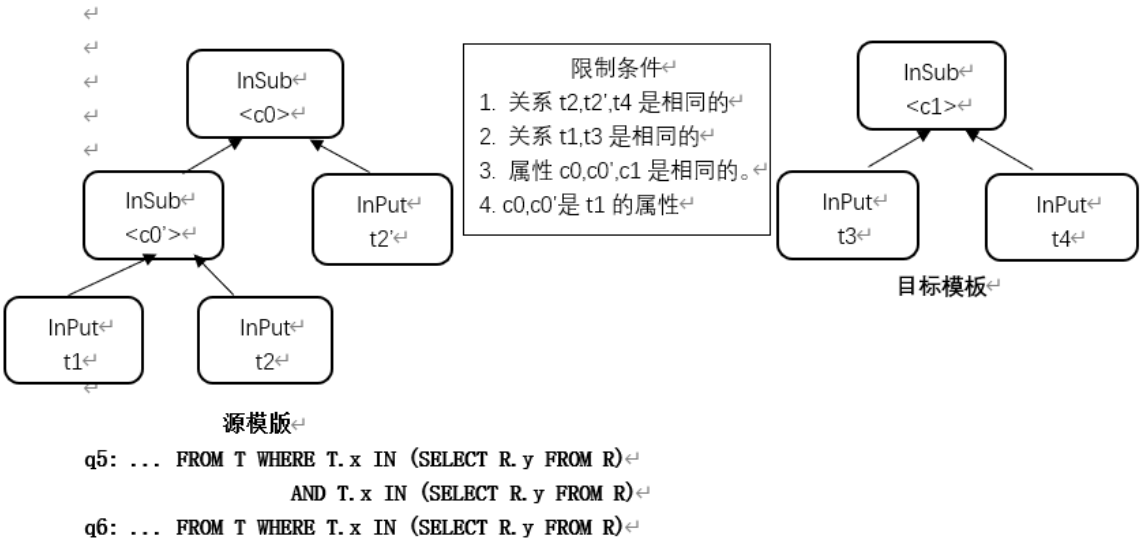


图 4.1　WeTune 发现的示例规则

图 4.1 显示了一个重写规则的例子，它可以消除 SQL 查询中多余的 IN-subquery。源模板q_{src}被表示为 InSub$_{c0}$(InSub$_{c0'}$ (t1, t2), t2')。操作符 InSub$_{c0}$有一个左子 InSub$_{c0'}$和一个右子 t2'。InSub 是查询计划模板中的一个操作符，代表 IN-subquery。它用两个 IN-subquery 运算符表示查询（例如，$q5$），这两个 IN-subquery 通过 AND 连接。目的模板q_{dest}是 InSub$_{c1}$（t3，t4）。约束集C指定了以下约束：$t2, t2'$和$t4$是相同的关系；$t1$和$t3$是相同的关系；$c0$, $c0'$和$c1$是相同的属性。图中还显示了一个从 Gitlab[10]中得到的 SQL 查询$q5$。这个查询与q_{src}匹配。因此，它可以被一个更好的查询$q6$所取代，该查询遵循q_{dest}所规定的模式，在C中的约束条件。这种低效率模式是非常反直觉的，因为它的两个内部子查询几乎是相同的。然而，研究的数据库中没有一个能成功优化这个查询。

规则枚举器列举了潜在的重写规则。为此，它首先列举了所有可能的计划模板。为了限制搜索空间，它限定了模板的大小，使模板中的运算符的数量在某个小的阈值之内。然后，对于每一对计划模板，它列举了所有潜在的约束。最后，它选择有希望的规则，这些规则有可能提高查询的性能。

查询计划模板是一棵树，其节点是具有符号输入或参数的关系代数运算符

表 4.1　WeTune 支持的 SQL 运算符

操作符名字	符号	输入	描述
Input	Input$_r$	0	Input$_r$()表示由指定的初始输入关系r
Projection	Proj$_a$	1	Proj$_a$(R)将其输入关系投射R到由指定的属性上 a

Selection	Sel$p_{p,a}$	1	Sel$p_{p,a}$(R)丢弃其输入关系 R 中不满足谓词的元组p，R 上的属性值 a 用于计算谓词 p
In-Sub Selection	$InSub_a$	2	$InSub_a(R_l,R_r)$ 丢弃左侧输入R_l中不存在的元组R_r。R_l中属性的值 a 用于状态检查
(Inner/Left/Right) Join	(I/L/R)$Join_{al,ar}$	2	(I/L/R)$Join_{al,ar}(R_l,R_r)$ 笛卡尔积其输入关系 R_l 和 R_r，然后丢弃在属性和上具有不匹配值的元组 a_l和 a_r。（L/R）Join 还保留不匹配的元组，并在右侧/左侧属性上填充 NULL。
Deduplication	Dedup	1	Dedup（R）丢弃其输入关系中重复的元组R。

WeTune 的枚举策略分别枚举了查询计划的树状结构和每个树状节点的运算符类型。更具体地说，枚举分三步进行：首先，WeTune 构建所有可能的树结构，内部有两种树节点：一种是有一个孩子的节点，另一种是有两个孩子的节点；其次，对于每个树结构，它详尽地将表 4.1 中列出的运算符分配给每个节点，以枚举具体的树。操作符的数量应该与节点的子节点的数量相匹配；最后，增加输入节点作为叶节点的子节点。图 4.2 显示了有两个运算符的查询计划模板的列举过程。蓝色块表示具有一个输入的运算符，而橙色块表示具有两个输入的运算符。为了减少列举空间，WeTune 只列举不包括输入节点在内的 4 个运算符的模板。此外，它还会过滤掉那些导致无效 SQL 查询的模板，例如错误地放置重复计算操作符。这些主要在代码 fragment 中将这些理论进行具体化，并成功枚举。

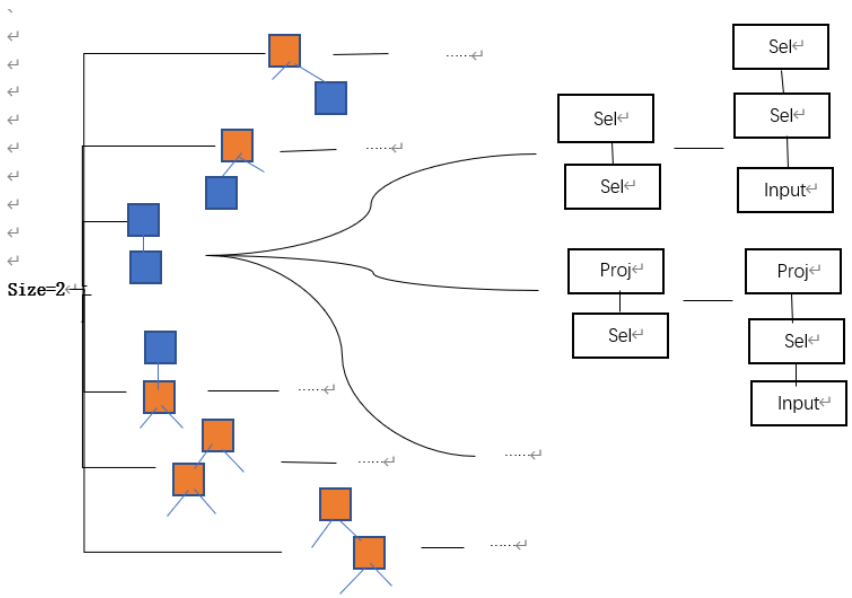


图 4.2　枚举查询计划模板的示例

4.1.2 约束枚举器

WeTune 将列举的模板配对为<q_{src}, q_{dest}>，并搜索能将这对模板变成有效重写规则的约束集。约束是一个谓词，它规定了q_{src}和q_{dest}中符号之间的关系。为了约束搜索空间，WeTune 考虑了以下有限的约束，这些约束来自于 WeTune 研究现有的重写规则和检查开发人员的手动查询重写的经验。总共有八个约束的举例：RelEq(rel_1, rel_2)、AttrsEq($attrs_1$, $attrs_2$)、PredEq($pred_1$, $pred_2$)、SubAttrs($attrs_1$, $attrs_2$)、RefAttrs(rel_1, $attrs_1$, rel_2, $attrs_2$)、Unique(rel, attrs) 、NotNull(rel, attrs)[13]。

给定一对计划模板<q_{src}, q_{dest}>，约束列举生成集合 C*，其中包含与q_{src}和q_{dest}相关的所有可能的约束。这是通过用q_{src}和q_{dest}中的符号详尽地填入上述约束的参数来实现的。我们在后面寻找有希望的规则时使用 C*。这些主要在代码 constraint 中将这些理论进行具体化，并成功枚举得到约束。

4.1.3 搜寻期望规则

给定一对计划模板<q_{src}, q_{dest}>，约束列举生成集合 C*，其中包含与q_{src}和q_{dest}相关的所有可能的约束。这是通过用q_{src}和q_{dest}中的符号详尽地填入上述约束的参数来实现的。我们在后面寻找有希望的规则时使用 C*。

```
1  EnumerateRules(k):
2    T := EnumerateTemplates(k)
3    R := ∅
4   foreach ⟨q_src, q_dest⟩ ∈ T × T do
5       if q_dest is not simpler than q_src then continue
6       C* := EnumerateConstraints(q_src, q_dest)
7        ℂ := SearchRelaxed(q_src, q_dest, C*)
8        R := R ∪ {⟨q_src, q_dest, C⟩ | C ∈ ℂ }
9   return R
10 SearchRelaxed(q_src, q_dest, C*):
11      if ¬ProveEq(q_src, q_dest, C*) then return ∅
12      ℂ := ∅
13      foreach c ∈ C* do
14          ℂ := ℂ ∪ SearchRelaxed(q_src, q_dest, C* − {c })
15      if ℂ = ∅ then return {C*}
16      else return ℂ
```

图 4.3　搜索期望规则的算法

图表 4.3 显示了搜索有希望的规则的基本算法。它首先列举所有的查询模板，然后，它将列举的模板配对为<q_{src}, q_{dest}>，并保留那些q_{dest}与q_{src}的每种类型的操作符相同或更少

的模板（第 5 行）。对于每一对<q_{src}, q_{dest}>，它通过列举与q_{src}和q_{dest}相关的所有可能的约束，生成约束集C^*。最后，它调用 SearchRelaxed 来递归搜索C^*的子集以形成有希望的规则。

函数 SearchRelaxed 从C^*开始，通过删除一个约束条件（第 14 行）迭代地放松约束集，并验证所产生的规则的正确性（第 11 行）。具体来说，它使用一个底层验证器（第五节）来证明q_{src}和q_{dest}在C中的约束条件下的等价性（第 5 节）。如果验证失败，我们就知道约束集太宽松了，无法暗示等价关系。在这种情况下，我们停止进一步放松和回溯（第 11 行）。如果没有任何约束可以被进一步删除，我们就找到了最宽松的约束集（第 15 行）。请注意，可能有多个最松弛的集合，SearchRelaxed 试图找到所有这些集合。这就是为什么它返回一个集合，而每个成员都是一个最宽松的集合。

为了降低搜索成本，WeTune 引入了以下方法：首先，它从C^*中排除了无用的约束。如果一个约束只涉及q_{dest}中的符号或导致非法的查询计划，则被认为是无用的；第二，它不检查C^*的每个子集，而只检查闭包和非冲突的子集。如果一个子集不能暗示集合中缺少任何约束，那么它就是一个闭包。同时，如果一个子集中没有约束条件相互冲突，那么这个子集就是不冲突的。如果两个约束条件放在一起会引入一个非法的计划，那么它们就有冲突。最后，如果约束集C可以被一个约束集C'所隐含，并且C'已经被证明可以使q_{src}和q_{dest}等价，那么 WeTune 将跳过对该约束集的检查。

搜索期望规则与计划模板枚举器和约束枚举器都在 discover 代码中进行融合，直接生成 success.txt 将得到的证明等价的有期望的规则放入其中。

4.2 WeTune 的内置规则验证器

4.2.1 规则的形式化表示

在高层次上，WeTune 的内置验证器的工作方式是首先将规则<q_{src}, q_{dest}, C>表示为 U-expressions 表达式[11]，然后将该表达式转换为 FOL 公式。最后，使用 SMT 求解器对 FOL 公式进行验证。给定一个重写规则，WeTune 使用 U-expressions 表达式来表示q_{src}和q_{dest}，并使用 FOL 公式来指定约束集C。这一过程主要用来验证规则的正确性，对于减少冗余规则，与筛选有期望的规则占据着重要的地位。

U-expressions：受 UDP[11]的启发，WeTune 也使用 U-expressions 来模拟 bag 语义下的 SQL 查询，它捕捉了关系中元组的多重性。在这种表述下，查询被看作是对自然数的语义的操作[12]。WeTune 采用 UDP[11]中定义的术语，总结有$[\![R]\!](x)$、$[b]$、$\|e\|$、$not(e)$、$\sum_{t\in D} f(t)$

[13]这些关系。

将查询模板转换为 U-expression：WeTune 将每个查询模板q翻译成一个函数$[\![q]\!](t)$：$Tuple \rightarrow \mathbb{N}$，它将一个元组$t$作为输入，并返回其在查询模板输出关系中的多重性。多重性被表示为一个 U-expression. 与 UDP[11]为具体查询执行到 U-expression 的转换不同，WeTune 为符号查询模板进行转换。翻译包括两个步骤：

第一步。翻译查询模板中的符号为三个未解释的函数：$[\![r]\!](t)$：$Tuple \rightarrow \mathbb{N}$、$[\![a]\!](t)$：$Tuple \rightarrow Tuple$、$[\![p]\!](t)$：$Tuple \rightarrow Bool$[13]。

第二步，翻译计划结构。这是在树状结构上通过递归来完成的，如图 4.4 所描述的。函数 ToUExpr 将一个（子）计划模板作为输入。它返回翻译后的表达式和输出关系的一个代表性元组。对于每个运算符，该算法递归计算其子代的表达式，然后在表 4.2 中查找，根据其子代的表达式建立自己的表达式。图 4.5 显示了翻译 U-expression 图 4.1 中的示例。

```
1 ToUExpr(q):
2     ⟨f_l, t_l⟩ := ToUExpr(q.child[0]) //None if no child
3     ⟨f_r, t_r⟩ := ToUExpr(q.child[1]) //None if single child
4 return TranslateByTable3(q, f_l, t_l, f_r, t_r)
```

图 4.4　计划表转为 U-expression 的算法

表 4.2　将 SQL 运算符转换为 U-expression 的规则

操作符	表达形式
Input_r	$f(t) := r(t)$
Proj_a	$f(t) := \sum_{tl}(f_l(t_l) \times [t = a(t_l)])$
$\text{Sel}_{p,a}$	$f(t) := f_l(t) \times [p(a(t))]$
InSubSel_a	$f(t) := f_l(t) \times \lvert\lvert f_r(a(t)) \rvert\rvert \times not([IsNull(a(t))])$
$\text{IJoin}_{al,ar}$	$f(t) := \sum_{tl,tr}([t = t_l \cdot t_r] \times f_l(t_l) \times f_r(t_r) \times [a_l(t_l) = a_r(t_r)] \times not([IsNull(a_l(t_l))]))$
$\text{LJoin}_{al,ar}$	$f(t) := (IJoin\ Expr.) + \sum_{tl,tr}([t = t_l \cdot t_r] \times f_l(t_l) \times [IsNull(t_r)] \times not(\sum_{t_r'}(f_r(t_r') \times [a_l(t_l) = a_r(t_r')] \times not([IsNull(a_l(t_l))])))$
$\text{RJoin}_{al,ar}$	$f(t) := (IJoin\ Expr.) + \sum_{tl,tr}([t = t_l \cdot t_r] \times f_r(t_r) \times [IsNull(t_l)] \times not(\sum_{t_l'}(f_l(t_l') \times$

$$[a_r(t_r) = a_l(t_l')] \times not([IsNull(a_r(t_r))]))$$

Dedup	$f(t) := \|\| f_l(t) \|\|$

Example SQL $q5$: `... FROM T WHERE T.x IN (S) AND T.c IN (S)`
$q6$: `... FROM T WHERE T.x IN (S)`

Templates q_{src}: $InSub_a(InSub_a(r_0, r_1), r_1)$
q_{dest}: $InSub_a(r_0, r_1)$

$$[\![q_{src}]\!](t) := r_0(t) \times not([IsNull(a(t))]) \times ||\sum_x r_1(x) \times [x = a(t)]||$$
$$\times not([IsNull(a(t))]) \times ||\sum_x r_1(x) \times [x = a(t)]||$$
$$[\![q_{dest}]\!](t) := r_0(t) \times not([IsNull(a(t))]) \times ||\sum_x r_1(x) \times [x = a(t)]||$$

图 4.5　重写规则的 U 表达式示例

处理 NULL：UDP 对 SQL 查询建模的最大限制之一是它假定关系中没有一个属性是 NULL。因此，UDP 不能支持 OUTER JOIN 操作。根据 WeTune 的研究，从网络应用中收集到的一半以上的 SQL 查询涉及此类操作。为了处理 NULL 和 OUTER JOIN，WeTune 对 U-expression 的翻译考虑到了 NULL 对运算符的影响。如表 4.2 所示。对于运算符 Input$r$，表达式r(NULL)返回输入关系中 NULL 的元组（元组是 NULL 如果所有属性都是 NULL。NULL 属性可以被认为是一个空元组只有一个属性）。

为了模拟 NULL 对表 4.2 中其他运算符的影响，WeTune 为 U-expression 引入了一个新的谓词 IsNull。当x是 NULL 时，IsNull(x)返回真，[IsNull(x)]为 1。详细来说，对于 InSubSel$_a$（IN-subquery），它使用 IsNull 谓词来过滤掉外部查询中的 NULL 元组。对于 INNER JOIN，它使用 IsNull 来过滤掉左或右关系有 NULL 元组的情况。

支持外层连接操作：WeTune 通过使用表 4.2 中基于 NULL 建模的特定规则来支持 OUTER JOIN 操作。与 INNER JOIN 不同的是，OUTER JOIN 保留另一方没有匹配的记录，并且用 NULL 填充空白。例如，“x LEFT JOIN y ON x.a = y.b ”保留左表 x 的所有记录，对于不匹配 x.a = y.b 的任何右边记录的左边记录，NULL 被附加为右边记录。因此，如表 4.2 所示，LEFT JOIN 是两部分的相加：

（1） 对于那些匹配的行与 INNER JOIN 相同

（2） 对于那些不匹配的行是三个术语的乘积。

Example SQL $q7$: `SELECT T.* FROM T LEFT JOIN S ON T.k=S.k'`
$q8$: `SELECT T.* FROM T`
`Integrity Constraint: S.k' is unique key`

Templates q_{src}: $Proj(LJoin_{a_0,a_1}(r_0,\ r_1))$
q_{dest}: $Proj(r_0)$

$$[\![q_{src}]\!](t) \coloneqq \sum_{x,y}([t = x] \times r_0(x) \times r_1(y) \times [a_0(x) = a_1(y)] \times NonNull(a_0(x)))$$
$$+ \sum_{x,y}([t = x] \times r_0(x) \times [IsNull(y)]$$
$$\times not(\sum_{y'} r_1(y') \times [a_0(x) = a_1(y')] \times NonNull(a_0(x))))$$
$$[\![q_{dest}]\!](t) \coloneqq \sum_{x}([t = x] \times r_0(x))$$

图 4.6　翻译 LEFT JOIN 的例子

用 FOL 公式表示约束：根据表 4.3，每个约束条件被直接翻译成 FOL 公式。一组约束 C将转换为其成员的合集：

$$ToFOL(C) \triangleq \bigwedge_{c \in C} ToFOL(c) \tag{4.1}$$

表 4.3　从约束到 FOL 公式的转换表

约束	表达形式
RelEq($r1$, $r2$)	$\forall t.r_1(t) = r_2(t)$
AttrsEq($a1$, $a2$)	$\forall t.a_1(t) = a_2(t)$
PredEq($p1$, $p2$)	$\forall t.p_1(t) = p_2(t)$
SubAttrs($a1$, $a2$)	$\forall t.a_1(t) = a_1(a_2(t))$
RefAttrs($r1$, $a1$, $r2$, $a2$)	$\forall t_1.((r_1(t_1) > 0 \ \wedge \ \neg(\text{IsNull}(a_1(t_1))))$ $\Rightarrow \exists t_2.(r_2(t_2) > 0 \wedge \neg(\text{IsNull}(a_2(t_2))) \wedge [a_1(t_1) = a_2(t_2)]))$
Unique(r, a)	$(\forall t.r(t) \leqslant 1) \ \wedge \ (\forall t, t'.r(t) > 0 \wedge r(t') > 0 \wedge a(t) = a(t')$ $\Rightarrow t = t')$
NotNull(r, a)	$\forall t.r(t) > 0 \Rightarrow \neg(IsNull(a(t)))$

4.2.2　规则的正确验证

在用 U-expressions 形式化查询模板和用 FOL 公式形式化约束后，规则验证器将使用 SMT 求解器检查规则的正确性。

定义一个规则的正确性：为了形式化正确性，WeTune 首先引入解释的概念，它规定了关系、谓词和属性列表符号的含义。WeTune 给定了两个定义：定义 1（解释）、定义 2（重写规则的正确性）即以下公式成立[13]：

$$\forall I.C^I \Rightarrow \forall t.q_{src}^I(t) = q_{dest}^I(t) \tag{4.2}$$

为了证明查询等价，UDP 依赖于将两个 U-expressions 转换为其规范化形式，这对于带有 OUTER JOIN 等操作符的查询来说是无法保证的。即 WeTune 对于带有 OUTER JOIN 等操作符是不支持的，图 4.6 显示了一个例子。

基于逻辑的决策过程：不同于 UDP，WeTune 使用一个基于逻辑的决策过程，它将正确性定义（定义 2）转化为 FOL 公式，并用 SMT 求解器进行验证。分两个重要部分。

第一个部分是将 U-expression，$q_{\text{src}}(t) = q_{\text{dest}}(t)$翻译成 FOL 公式。WeTune 根据表 4.4 来执行翻译。该表显示了翻译所使用的基本 U-expressions 和它们相应的 FOL 公式。例如，图 4.7 显示了在证明图 4.1 中两个查询的等价性时翻译的 FOL 公式。偶尔，WeTune 在表 4.4 中找不到任何匹配。在这种情况下，验证者不能证明该规则的正确性。

$$\begin{aligned}[\![q_{src}]\!](t) = r_0(t) \times [\neg(IsNull(a(t))] \times [\exists x.r_1(x) \times [x = a(t)] > 0] \\ \times [\exists x.r_1(x) \times [x = a(t)] > 0] \\ [\![q_{dest}]\!](t) = r_0(t) \times [\neg(IsNull(a(t))] \times [\exists x.r_1(x) \times [x = a(t)] > 0]\end{aligned}$$

图 4.7　翻译 FOL 公式示例

表 4.4　U-expression 到 FOL 公式的转换表

U-expression	FOL 公式
$\boldsymbol{f_1(t) = f_2(t)}$	$Tr(\boldsymbol{f_1(t)}) = Tr(\boldsymbol{f_2(t)})$
$\boldsymbol{f_1(t) + f_2(t)}$	$Tr(\boldsymbol{f_1(t)}) + Tr(\boldsymbol{f_2(t)})$
$\boldsymbol{f_1(t) \times f_2(t)}$	$Tr(\boldsymbol{f_1(t)}) \times Tr(\boldsymbol{f_2(t)})$
$\|\boldsymbol{f(t)}\|$	$ite(Tr(f(t)) > 0, 1, 0)$
$\boldsymbol{not(f(t))}$	$ite(Tr(f(t)) > 0, 0, 1)$
$[\boldsymbol{p}]$	$ite(Tr(p), 1, 0)$
$\|\sum_t \boldsymbol{f(t)}\|$	$ite(\exists t.Tr(f(t)) > 0, 1, 0)$

$\boldsymbol{not}(\sum_t \boldsymbol{f}(\boldsymbol{t}))$	$ite(\exists t.Tr(f(t)) > 0,0,1)$
$\sum_t \boldsymbol{f}(\boldsymbol{t}) = \boldsymbol{0}$	$\forall t.f(t) = 0$
$\sum_t \boldsymbol{f}(\boldsymbol{t}) = \boldsymbol{1}$	$\exists t.(f(t) = 1 \wedge (\forall t'.t' \neq t \Rightarrow f(t') = 0))$
$\sum_t \boldsymbol{r}(\boldsymbol{t}) \times \boldsymbol{f}(\boldsymbol{t}) = \sum_t \boldsymbol{r}(\boldsymbol{t}) \times \boldsymbol{g}(\boldsymbol{t})$	$\forall t.r(t) \times Tr(f(t)) = r(t) \times Tr(g(t))$
$\sum_t \boldsymbol{r}(\boldsymbol{t}) \times \boldsymbol{f}(\boldsymbol{t}) = \sum_{t,s} \boldsymbol{r}(\boldsymbol{t}) \times \boldsymbol{g}(\boldsymbol{t}) \times \boldsymbol{h}(\boldsymbol{t},\boldsymbol{s})$	$\forall t.((r(t) \times Tr(f(t)) \neq r(t) \times Tr(g(t)) \wedge r(t) \times Tr(f(t)) = 0 \wedge Tr(\sum_s h(t,s) = 0)) \vee (r(t) \times Tr(f(t)) = r(t) \times Tr(g(t)) \wedge (r(t) \times Tr(f(t)) = 0 \vee Tr(\sum_s h(t,s) = 1))))$

当把 U-expressions 翻译成 FOL 公式时，最困难的部分是求和的翻译（表 4.4 的最后两行）。具体来说，WeTune 提出定理公式 4.3 和定理公式 4.4，对应于表 4.4 的最后两行。定理公式 4.3 消除了两个相加的变量对齐时的相加。它可以被推广到多个求和变量。

$$\left(\forall I \forall t.r^I(t) \times f^I(t) = r^I(t) \times g^I(t)\right)$$

$$\Leftrightarrow \left(\forall I.\sum_t \left(r^I(t) \times f^I(t)\right) = \sum_t \left(r^I(t) \times g^I(t)\right)\right) \tag{4.3}$$

其中r是表示关系的函数，$f(t)$和$g(t)$是任意表达式。上标I表示符号在I下的解释。

定理 4.4 将定理 4.3 推广到和变量不一致的情况下。

$$\left(\begin{array}{l}\forall I \forall t.\left(r^I(t) \times f^I(t) \neq r^I(t) \times g^I(t) \wedge r^I(t) \times f^I(t) = 0 \wedge \sum_s h^I(t,s) = 0\right) \\ \vee \left(r^I(t) \times f^I(t) = r^I(t) \times g^I(t) \wedge \left(r^I(t) \times f^I(t) = 0 \vee \sum_s h^I(t,s) = 1\right)\right)\end{array}\right)$$

$$\Rightarrow \left(\forall I.\sum_t \left(r^I(t) \times f^I(t)\right) = \sum_{t,s} \left(r^I(t) \times g^I(t) \times h^I(t,s)\right)\right) \tag{4.4}$$

其中 h（t，s）是任意表达式

第二部分是，通用量词可能会使证明变得不可判定，导致 SMT 求解器超时。当证明一个 FOL 公式是同义词时，SMT 求解器需要详尽地检查所有情况。证明一个 FOL 公式是不可满足的（UNSAT）要容易得多，因为 SMT 求解器一旦发现暗示 UNSAT 的矛盾就会停止，这可以避免详尽的推理。

因此，给定的重写规则，WeTune 通过证明$\neg(C \Rightarrow \forall t.q_{\text{src}}(t) = q_{\text{dest}}(t))$是 UNSAT 来验证其正确性，且 WeTune 保守地认为那些导致超时的规则是不正确的。这一部分主要是在 logic

的代码中进行体现，进行内置验证器的设置，验证 WeTune 进行枚举得到的规则是正确的。

4.3 减少冗余规则的方法

WeTune 提出了两个额外的优化策略，以减少多余的规则和消除 SQL 语句中的 ORDER BY。而本文主要是对减少冗余的规则的创新。

减少冗余规则：多个规则可以重写一个查询。例如，考虑一个查询q和三个规则$R1, R2, R3$，我们可以通过：

（1）连续应用$R1, R2$。

（2）应用$R3$ 来改写q后得到相同的查询。

因此，$R3$ 是多余的，可以用$R1$ 和$R2$ 的组合代替。在规则发现过程中，最好能减少这种多余的规则。形式上，给定一个规则集$\mathbb{R}$和一个规则$R \in \mathbb{R}$，R在$\mathbb{R}$下是可减少的。如果

$$\forall \mathrm{q}.\left(\mathrm{Rewritr}(\mathbb{R}, \mathrm{q}) = \mathrm{Rewritr}(\mathbb{R} - \{\mathrm{R}\}, \mathrm{q})\right) \tag{4.5}$$

检查所有的查询是不可能的。相反，WeTune 根据R的源计划模板和约束条件，生成一个具体的探测查询$\hat{q}$和具体的约束。首先，WeTune 根据 SPES 的具体化q_{src} 的步骤，将q_{src} 具体化为$\hat{q}$。其次，WeTune 根据规则中的 NotNull、Unique 和 RefAttrs 约束，为$\hat{q}$添加具体的完整性约束。图 4.8 显示了一个例子。

探测查询$\hat{q}$的具体生成步骤的主要是 SPES 的具体化q_{src} 的步骤，即给定一个重写规则$<q_{\mathrm{src}}, q_{\mathrm{dest}}, C>$，WeTune 需要将其转换成 SPES 所接受的输入。由于 SPES 只接受具体的 SQL 查询，而不承认约束集C，因此 WeTune 根据约束集C将q_{src} 和$q\mathrm{d}_{\mathrm{est}}$ 具体化，如图 4.8 所示的例子，包括以下三个步骤：

首先，WeTune 根据包括 RelEq、AttrsEq、PredEq 和 AggrEq 在内的等价约束为q_{src} 和q_{dest} 中的每个符号分配名称。具体来说，它将等价符号放入同一集合，同一集合中的所有符号将共享一个随机生成的名称。例如，在图 4.1 中，$t2$、$t2'$和$t4$ 可以被分配为“T2 ”的名称。$c0, c0'$和$c1$ 可以被分配为名称为$C1$。

其次，对于每个属性，我们根据 SubAttrs 约束找到它所属的关系。如果一个名称为c的属性属于一个名称为t的关系，那么我们将属性名称从c改为$t.c$。对于图 4.1 中的例子，属性列表$c0$ 的名称将被改变为$T1.C1$。

第三，我们根据关系的属性来构建模式定义。在图 4.1 中，T1 的模式有 1 列$C1$。

$$q_{src}: \mathrm{Proj}_{a0}\ (\mathrm{LJoin}_{a1,a2}\ (\mathrm{Input}_{r0}\ ,\ \mathrm{Input}_{r1}\ ))$$

$$C = \{\mathrm{SubAttrs}(a0,\ ar0\ ),\ \mathrm{SubAttrs}(a1,\ ar0\ ),$$

$$\mathrm{SubAttrs}(a2,\ ar1\ ),\ \mathrm{Unique}(r1,\ a2),\ \cdots\}$$

$$\hat{q}:\ \mathrm{SELECT\ T.a\ AS\ k\ FROM\ T\ LEFT\ JOIN\ S\ ON\ T.b{=}S.c}$$

integrity constraint: S.c is unique key

图 4.8　生成的探测查询的示例

$\hat{q}$必须是R适用的最小模式。也就是说，任何R适用的查询必须包含$\hat{q}$的模式。因此，要决定可归约性，只需检查以下条件是否为真：

$$\mathrm{Rewritr}(\mathbb{R}, \hat{q}) = \mathrm{Rewritr}(\mathbb{R} - \{\mathrm{R}\}, \hat{q}) \tag{4.6}$$

4.4 减少冗余规则的缺点

在 WeTune 进行减少冗余得到最宽松规则集（即执行 reduce 程序）的过程中，使用的算法如图 4.9 类迭代的进行搜寻减少，在搜寻过程中所用到的时间与 reduce 使用的数据集大小有着很大的关系，规则集如果过于庞大，在 reduce 的过程中，会造成时间过长。而 WeTune 中 reduce 的对应的数据集包含 230 多万行，要一一进行循环验证删除，这是使得 reduce 时间过长的主要原因。

```
try (final ProgressBar pb = new ProgressBar("Reduce", bank.size())) {
  final List<Substitution> rules = new ArrayList<>(bank.rules());
  for (int i = 0, bound = rules.size(); i < bound; i++) {
    pb.step();

    final Substitution rule = rules.get(i);
    try {
      if (isImpliedRule(rule)) bank.remove(rule);
      else bank.add(rule);

    } catch (Throwable ex) {
      System.err.println(i + " " + rule);
      //          ex.printStackTrace();
      bank.remove(rule);
      //          throw ex;
    }
  }
}
```

图 4.9　WeTune 减少冗余的主要代码

而在进行 reduce 过程中，逐一减少规则，验证是否满足条件公式（4.6），通过此来减少冗余规则得到最宽松的规则集结果。但是在减少的过程中其 reduce 的规则集文件有 230 多万行的规则，对每一行规则都逐一进行判断，而在判断过程中，规则集中大部分都是无法对具体化的$\hat{q}$进行规则重写的，所以在运行过程中会造成不必要的时间产生，使得对规则 R 是否可归约的判断使用过多不必要的时间。

而且在 reduce 的过程中，不是只进行一次对条件公式（4.6）的判断，在对全部的规则集完成以此判断删除之后，会对得到的规则集再次进行判断是否可归约，直到得到的规则集进行全部判断后没有减少一条规则，则完成 reduce，输出得到的规则集为优化后的规则集。因此数据集数量过大，对于多次的判断删除的运行的时间也有很大的影响。

对比如图 4.10 与图 4.11，可知，运行 60 万行的规则集时 30 分钟只能判断 3194 条规则是否是可归约的，且预测完成 60 万行规则集的第一次 reduce 需要 84 个小时左右才能全部运行完毕，但由第 4.4 小节所说的减少规则冗余的方法，在进行一次 reduce 之后得到的规则集，还需要继续进行判断，直到最终的输出规则及大小与输入的规则集大小完全一致才算最终的结果，因此 60 万行的规则集在进行一次 reduce 之后还需要继续进行 n 次直到最终规则集相等为止，因此完全完成对 60 万行的 reduce 需要比 84 小时更长的时间。而运行 1 万行的规则集时只需要运行 2 分 17 秒就可以完成 1 万行的全部判断，输出最终的规则集，期间进行了 5 次的规则集判断，得到最终相等的规则集。由此可知，规则集大小对减少规则集冗余有很大的影响。

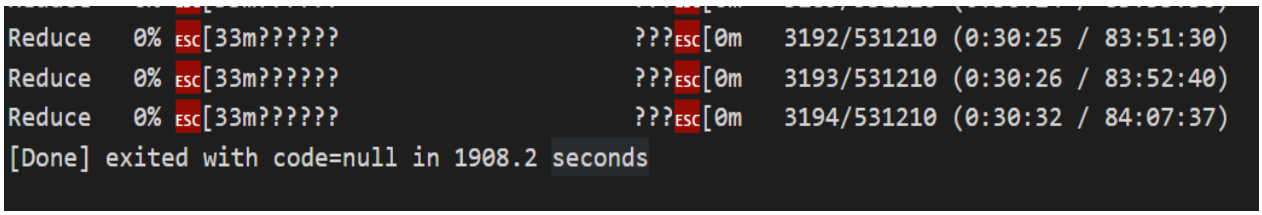
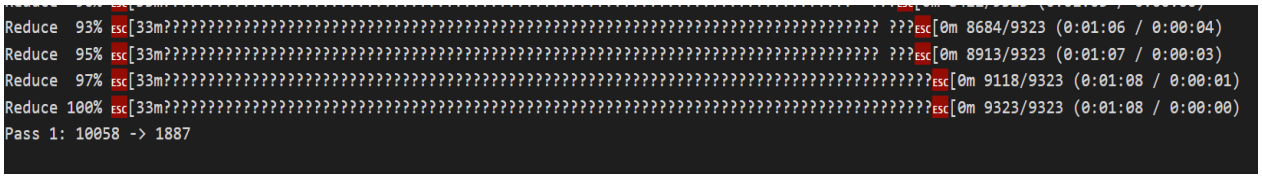
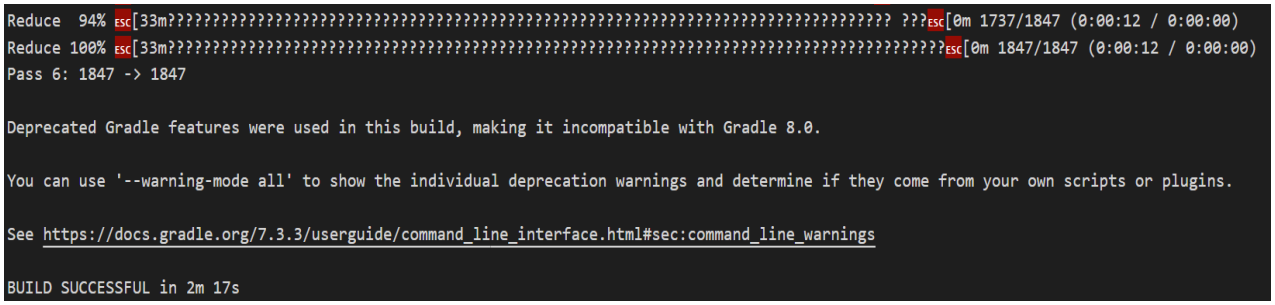


图 4.10　60 万行规则集 reduce 举例



（a）1 万行进行第一次 reduce 的时间



（b）1 万行最终完成 reduce 的时间

图 4.11　1 万行规则集进行 reduce 举例

4.5 对搜减少冗余规则时间缺点的改进

对减少冗余规则算法缺点的改进，就从其引起缺点的原因进行改进，即所说的 reduce 运行时间与规则集大小息息相关，要减少运行时间，则需要减少数据集的大小，让相似规则作为一个数据集进行多次迭代运行，减少不必要的时间运行。且为了保证得到最终规则集的正确性，需要对分割所得的所有规则集 reduce 得到的结果进行合并再 reduce 判断，这样的方法与 reduce 本身进行多次规则集判断是一样的道理。这样可以保证，最终得到的规则集，其中所有的规则都已经进行了 reduce 判断，不会存在可归约的规则，使得最终得到的结果是和目标规则集一起进行 reduce 的结果是一致的。

而经过多次不同大小规则集的 reduce 运行时间的对比与所得到结果的判断，本文认为 1 万个规则集及其 reduce 所需要的时间与得到的规则集的大小，都对之后的合并再进行 reduce 有很好的效果，1 万行的规则进行 reduce 之后得到的规则集大约在 2 千行左右，这样之后在进行合并时，会对 230 个分割为 1 万行的规则集运行 reduce 之后得到的规则集，按照每 10 个规则集进行合并再判断，得到每个合集的规则数量大致再 2 万行左右，这样每个合集的运行时间大约需要 20 分钟，之后对得到的 23 个规则集进行合并在判断，得到最终想要的规则集。

所做出的改进具体操作如下：

(1) 对目标规则集进行排序后分割为 230 个 1 万行的规则集，如图 4.12。

(2) 对分割后的所有规则集按照 1 万行为一个分割点进行 reduce，如图 4.13。

(3) 对所有规则集 reduce 后所得到的规则集，按照顺序每十个进行合并再次 reduce，如图 4.14。

(4) 重复此操作直到得到最终的规则集。

进行创新后运行全部规则得到的减少冗余后规则只需要 3 天左右，对比创新前，则需要运行 1 个月左右，再 reduce 的时间上有了显著的提升。

```
#分割
# split -l 10000 /root/wetune/wtune_data/rules/rules.0407133816.txt -d -a 3 rules_
```

图 4.12　分割目标规则集

```
for (( i = 000; i <= 230; i++)); do
if [ $i -le 9 ] ; then
    t=00$i
elif [ $i -le 99 ] ;  then
    t=0$i
else
    t=$i
fi
 out="${rules_dir}/rules$t.txt"
 in="${rules_dir}/rules_$t"
```

图 4.13　分割运行所有规则集

```
for ((i = 1 ; i <= 9; i++)) do
 echo rules00$i.txt;
done | xargs -i cat {} >> a_rules1.txt
```

图 4.14　按照顺序进行规则集合并

4.6 验证创新改进的正确性

对所做创新是否能够保证最终得到规则集的正确性，进行一个实验验证，保证理论与实践结果是符合的。

对目标规则集进行片段的截取，作为一个实验规则集，验证创新改进的正确性，对实验规则集进行如上小节所说的分割 reduce 直到找到最终结果规则集，所得到的规则集与直接对实验规则集进行 reduce 得到的规则集进行等价性判断，如果是等价的，则表明创新改进的正确性。对最终得到的两个规则集进行等价判断的主要代码如图 4.15 所示，图

4.15 中（a）对两个规则集进行排序后，逐行判断，如果等价则写入 thesame 这个文件里，最终只需要进行 thesame 与两个规则集是否有规则数量上的差距，如果不存在差异则证明是等价的。图 4.15 中（b）则对两个规则集进行分别排序后直接进行等价判断，输出文件编码，一致则表示两个文件内容的一致性。

图 4.15（b）中的实验规则表有 1 万行，对其总体进行 reduce，得到的规则表排序后存储到 uniq1，之后对此实验规则表进行 5 千行的拆分，分别 reduce 后再对两个结果规则集进行合并 reduce，最终得到的规则集排序后存储到 uniq2，md5sum 后得到的两个文件编码相等，且 thesame 文件中的规则集数量与 uniq1 和 uniq2 中规则集数量是一致的。证明两个规则集是等价的。为了验证并非是偶然情况，本文还将实验规则表的行数分别进行了 5 千行、2 万行、3 万行的实验，分别进行如上的分割合并 reduce 得到规则集的判断，得到的结果与图 4.15（b）中的结果一致，全部证明相等。并每一个都进行了 3 次以上，规避偶然性，验证了创新改进的正确性。

```
for Text1 in `cat  uniq1`
do
        for Text2 in `cat uniq2`
        do
                if [ "$Text1" =  "$Text2" ]
                then
                echo $Text1 >> thesame
                fi
        done
done
```

（a） 对两个规则集排序后进行逐行判断

```
root@5714bd077564:~/wetune/wtune_data/rules# md5sum uniq1 uniq2
ed6abdd66d18aad4e4a6251abf8b1891  uniq1
ed6abdd66d18aad4e4a6251abf8b1891  uniq2
```

（b） 对两个规则集排序后直接进行判断等价性

图 4.15　验证规则集的等价性

5 实验结果分析

5.1 模板枚举示例

所有最大尺寸为 4 的模板（剪枝后）将被打印出来，共 3113 个。

```
InnerJoin(Proj*(InnerJoin(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(LeftJoin(InnerJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(LeftJoin(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(InnerJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(LeftJoin(Input,Input),Input)),Input)
InnerJoin(Proj*(InSubFilter(InSubFilter(Input,Input),Input)),Input)
LeftJoin(Input,Proj(InnerJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InnerJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(LeftJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(LeftJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj(InSubFilter(InSubFilter(Input,Input),Input)))
LeftJoin(Input,Proj*(InnerJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InnerJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(LeftJoin(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(LeftJoin(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(InnerJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(LeftJoin(Input,Input),Input)))
LeftJoin(Input,Proj*(InSubFilter(InSubFilter(Input,Input),Input)))
```

（a）枚举模板操作符为 4 的例子

```
InnerJoin(InnerJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
InnerJoin(InnerJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
InnerJoin(InnerJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
InnerJoin(LeftJoin(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
InnerJoin(LeftJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
InnerJoin(LeftJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
InnerJoin(LeftJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
LeftJoin(InnerJoin(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
LeftJoin(InnerJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
LeftJoin(InnerJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
LeftJoin(InnerJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
LeftJoin(LeftJoin(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
LeftJoin(LeftJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
LeftJoin(LeftJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
LeftJoin(LeftJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InnerJoin(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(InnerJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InnerJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(InnerJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(LeftJoin(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(LeftJoin(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(LeftJoin(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(LeftJoin(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(InnerJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(InnerJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(LeftJoin(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(LeftJoin(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(InSubFilter(InnerJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(InSubFilter(LeftJoin(Input,Input),Input),Input),Input)
InSubFilter(InSubFilter(InSubFilter(InSubFilter(Input,Input),Input),Input),Input)
Total template at max size 4: 2261
```

（b）所举例子模板最大操作符为 4 的例子总共共有 2261 条

图 5.1　模板枚举示例

WeTune 最多只能枚举 4 个操作符，Input 不算入其中，具体的原理主要在第 4.2 小节所说的对于模板的枚举原理介绍，如图 5.1 所示就是 WeTune 进行模板枚举最多 4 个操作符的树的例子，得到的模板枚举，在之后进行一一配对，组成<q_{src}, q_{dest},>，并为之后的约束枚举提供条件。

5.2 查询重写示例

WeTune 得到的规则应用于实际的查询语句，对于每个规则，将打印以下项目：

（1）规则本身。如图 5.2 中的 1。

（2）一个查询 q0，该规则可以应用于此。

（3）查询 q1，对 q0 应用该规则后的重写结果。如图 5.2 中的 2。

对于可以通过 WeTune 内置的验证器来证明的规则，会打印额外的项目。

（4）一对 U-Expression，从规则中翻译出来。如图 5.2 中的 3。

（5）一个或多个 Z3 片段，提交给 Z3 的公式。进行 UNSAT 验证。如图 5.2 中的 4。

```
1. Rule String
  Filter<p0 b0>(Proj<a0 s0>(Input<t0>))|Proj<a1 s1>(Filter<p1 b1>(Input<t1>))|AttrsSub(b0,s0);AttrsSub(a0,t0);TableEq(t1,t0);AttrsEq(b1,b0);AttrsEq(a1,a0);PredicateEq(p1,p0);SchemaEq(s1
,s0)

2. Example Query
  SELECT * FROM (SELECT `t0`.`c0` AS `c0` FROM `r0` AS `t0`) AS `q0` WHERE P0(`q0`.`c0`)
  SELECT `t0`.`c0` AS `c0` FROM `r0` AS `t0` WHERE P0(`t0`.`c0`)

3. U-Expression
  [[q0]](x1) := ???{x0}([p0(b0(x1))] * [x1 = a0(x0)] * t0(x0))
  [[q1]](x1) := ???{x0}([x1 = a0(x0)] * t0(x0) * [p0(b0(x0))])

4. First-Order Formulas (Z3 Script)
==== Begin of Snippet-1 ====
(declare-sort Tuple 0)
(declare-fun t0 (Tuple) Int)
(declare-fun Null () Tuple)
(declare-fun a0 (Int Tuple) Tuple)
(declare-fun b0 (Int Tuple) Tuple)
(declare-fun x1 () Tuple)
(declare-fun x0 () Tuple)
(declare-fun p0 (Tuple) Bool)
(assert (forall ((x Tuple)) (>= (t0 x) 0)))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (a0 s x) Null))))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (b0 s x) Null))))
(assert (forall ((x Tuple) (s Int)) (= (b0 3 (a0 s x)) (b0 s x))))
(assert (= x1 x1))
(assert (not (= (ite (and (p0 (b0 3 x1)) (= x1 (a0 1 x0))) (t0 x0) 0)
       (ite (and (= x1 (a0 1 x0)) (p0 (b0 1 x0))) (t0 x0) 0))))

(check-sat)
==== End of Snippet-1 ====
==> Result: UNSATISFIABLE

Rule-1: EQ
```

图 5.2　查询重写示例

```
1. Rule String
  Proj*<a0 s0>(Input<t0>)|Proj<a1 s1>(Input<t1>)|AttrsSub(a0,t0);Unique(t0,a0);TableEq(t1,t0);AttrsEq(a1,a0);SchemaEq(s1,s0)

2. Example Query
  SELECT DISTINCT `t0`.`c0` AS `c0` FROM `r0` AS `t0`
  SELECT `t0`.`c0` AS `c0` FROM `r0` AS `t0`

3. U-Expression
  [[q0]](x1) := ???{x0}([x1 = a0(x0)] * t0(x0))
  [[q1]](x1) := ???{x0}([x1 = a0(x0)] * t0(x0))

4. First-Order Formulas (Z3 Script)
==== Begin of Snippet-1 ====
(declare-sort Tuple 0)
(declare-fun t0 (Tuple) Int)
(declare-fun Null () Tuple)
(declare-fun a0 (Int Tuple) Tuple)
(declare-fun x1 () Tuple)
(declare-fun x0 () Tuple)
(assert (forall ((x Tuple)) (>= (t0 x) 0)))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (a0 s x) Null))))
(assert (forall ((x Tuple)) (<= (t0 x) 1)))
(assert (forall ((x Tuple) (y Tuple))
  (=> (and (> (t0 x) 0) (> (t0 y) 0) (= (a0 1 x) (a0 1 y))) (= x y))))
(assert (= x1 x1))
(assert (not (= (ite (= x1 (a0 1 x0)) (t0 x0) 0) (ite (= x1 (a0 1 x0)) (t0 x0) 0))))

(check-sat)
==== End of Snippet-1 ====
==> Result: UNSATISFIABLE

Rule-2: EQ
```

图 5.3　查询重写示例

```
1. Rule String
  Proj<a2 s0>(InnerJoin<a0 a1>(Input<t0>,Input<t1>))|Proj<a3 s1>(Input<t2>)|AttrsSub(a0,t0);AttrsSub(a1,t1);AttrsSub(a2,t0);Unique(t1,a1);NotNull(t0,a0);Reference(t0,a0,t1,a1);TableEq(t
2,t0);AttrsEq(a3,a2);SchemaEq(s1,s0)

2. Example Query
  SELECT `t0`.`c2` AS `c2` FROM `r0` AS `t0` INNER JOIN `r1` AS `t1` ON `t0`.`c0` = `t1`.`c1`
  SELECT `t0`.`c2` AS `c2` FROM `r0` AS `t0`

3. U-Expression
  [[q0]](x2) := ???{x0,x1}([x2 = a2(x0)] * t0(x0) * [a0(x0) = a1(x1)] * not([IsNull(a1(x1))]) * t1(x1))
  [[q1]](x2) := ???{x0}([x2 = a2(x0)] * t0(x0))

4. First-Order Formulas (Z3 Script)
==== Begin of Snippet-1 ====
(declare-sort Tuple 0)
(declare-fun t0 (Tuple) Int)
(declare-fun t1 (Tuple) Int)
(declare-fun Null () Tuple)
(declare-fun a0 (Int Tuple) Tuple)
(declare-fun a1 (Int Tuple) Tuple)
(declare-fun a2 (Int Tuple) Tuple)
(declare-fun x2 () Tuple)
(declare-fun x0 () Tuple)
(assert (forall ((x Tuple)) (>= (t0 x) 0)))
(assert (forall ((x Tuple)) (>= (t1 x) 0)))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (a0 s x) Null))))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (a1 s x) Null))))
(assert (forall ((x Tuple) (s Int)) (=> (= x Null) (= (a2 s x) Null))))
(assert (forall ((x Tuple)) (= (> (t0 x) 0) (not (= (a0 1 x) Null)))))
(assert (forall ((x Tuple)) (<= (t1 x) 1)))
(assert (forall ((x Tuple) (y Tuple))
  (=> (and (> (t1 x) 0) (> (t1 y) 0) (= (a1 2 x) (a1 2 y))) (= x y))))
(assert (forall ((x Tuple))
```

（a）多个 z3 片段的规则等价证明示例

```
    (and (= (a0 1 x0) (a1 2 x1)) (not (= (a1 2 x1) Null)) (> (t1 x1) 0)))))
(assert (not (= x1 x1_)))
(assert (= (ite (= x2 (a2 1 x0)) (t0 x0) 0) (ite (= x2 (a2 1 x0)) (t0 x0) 0)))
(assert (and (= x2 (a2 1 x0)) (> (t0 x0) 0)))
(assert (Z x1))
(assert (Z x1_))

(check-sat)
==== End of Snippet-5 ====
==> Result: UNSATISFIABLE

Rule-7: EQ
```

（b）验证查询重写的规则的正确性

图 5.4　查询重写示例

5.3 约束枚举示例

总共展示了 5 对计划模板之间的约束枚举找到最终规则的示例，对于约束枚举的原理主要在第 4.2 小节进行了解释说明，示例将列举给定规则的计划模板之间的约束，并搜索最宽松的约束集。每个被检查的约束集及其验证结果都将被打印出来。找到的规则和度量将在列举完成后被添加。最终找到的符合条件的期望规则会被打印出来。

```
Going to verify: AttrsEq(a0,b0);AttrsSub(a0,t0);AttrsSub(b0,s0);Unique(t0,a0);NotNull(t0,a0);TableEq(t1,t0);AttrsEq(b1,b0);AttrsEq(a1,b0);PredicateEq(p1,p0);SchemaEq(s1,s0);
  => Answer from verifier: EQ, 10ms
  => Current EQ cache size: 4
Going to verify: AttrsEq(a0,b0);AttrsSub(a0,t0);AttrsSub(b0,s0);Unique(t0,a0);TableEq(t1,t0);AttrsEq(b1,b0);AttrsEq(a1,b0);PredicateEq(p1,p0);SchemaEq(s1,s0);
  => Answer from verifier: EQ, 9ms
  => Relax. Current EQ cache size: 4
Going to verify: AttrsEq(a0,b0);AttrsSub(a0,t0);AttrsSub(b0,s0);NotNull(t0,a0);TableEq(t1,t0);AttrsEq(b1,b0);AttrsEq(a1,b0);PredicateEq(p1,p0);SchemaEq(s1,s0);
  => Answer from verifier: EQ, 9ms
  => Current EQ cache size: 5
Going to verify: AttrsEq(a0,b0);AttrsSub(a0,t0);AttrsSub(b0,s0);TableEq(t1,t0);AttrsEq(b1,b0);AttrsEq(a1,b0);PredicateEq(p1,p0);SchemaEq(s1,s0);
  => Answer from verifier: EQ, 10ms
  => Relax. Current EQ cache size: 4

Found Rules (4 in total)
  Filter<p0 a1>(Proj<a0 s0>(Input<t0>))|Proj<a3 s1>(Filter<p1 a2>(Input<t1>))|AttrsEq(a0,a1);AttrsSub(a0,t0);AttrsSub(a1,s0);TableEq(t1,t0);AttrsEq(a2,a0);AttrsEq(a3,a0);PredicateEq(p1,
p0);SchemaEq(s1,s0)
  Filter<p0 a1>(Proj<a0 s0>(Input<t0>))|Proj<a3 s1>(Filter<p1 a2>(Input<t1>))|AttrsEq(a0,a1);AttrsSub(a0,t0);AttrsSub(a1,s0);TableEq(t1,t0);AttrsEq(a2,a0);AttrsEq(a3,a1);PredicateEq(p1,
p0);SchemaEq(s1,s0)
  Filter<p0 a1>(Proj<a0 s0>(Input<t0>))|Proj<a3 s1>(Filter<p1 a2>(Input<t1>))|AttrsSub(a0,t0);AttrsSub(a1,s0);TableEq(t1,t0);AttrsEq(a2,a1);AttrsEq(a3,a0);PredicateEq(p1,p0);SchemaEq(s1
,s0)
  Filter<p0 a1>(Proj<a0 s0>(Input<t0>))|Proj<a3 s1>(Filter<p1 a2>(Input<t1>))|AttrsEq(a0,a1);AttrsSub(a0,t0);AttrsSub(a1,s0);TableEq(t1,t0);AttrsEq(a2,a1);AttrsEq(a3,a1);PredicateEq(p1,
p0);SchemaEq(s1,s0)

Metrics:
  C* size: 12
  # of enumerated constraint sets: 20
  # of EQ constraint sets: 16
  # of NEQ constraint sets: 0
  # of UNKNOWN constraint sets: 4
  # of built-in verifier invocations: 20
  # of EQ cache hit: 0
  # of NEQ cache hit: 0
  # of Relaxing: 0
```

图 5.5　约束枚举的示例

```
Found Rules (1 in total)
  Proj*<a0 s0>(Input<t0>)|Proj<a1 s1>(Input<t1>)|AttrsSub(a0,t0);Unique(t0,a0);TableEq(t1,t0);AttrsEq(a1,a0);SchemaEq(s1,s0)

Metrics:
  C* size: 6
  # of enumerated constraint sets: 3
  # of EQ constraint sets: 2
  # of NEQ constraint sets: 1
  # of UNKNOWN constraint sets: 0
  # of built-in verifier invocations: 3
  # of EQ cache hit: 0
  # of NEQ cache hit: 0
  # of Relaxing: 0
```

图 5.6　约束枚举的示例

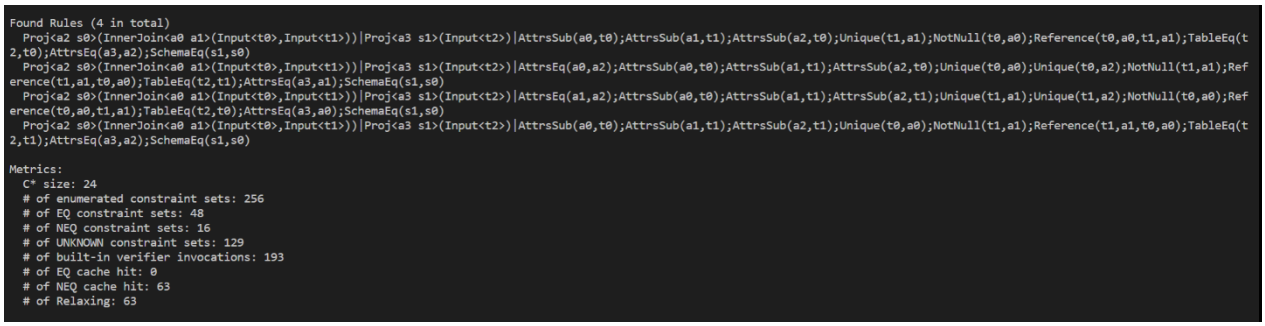


图 5.7　约束枚举的示例

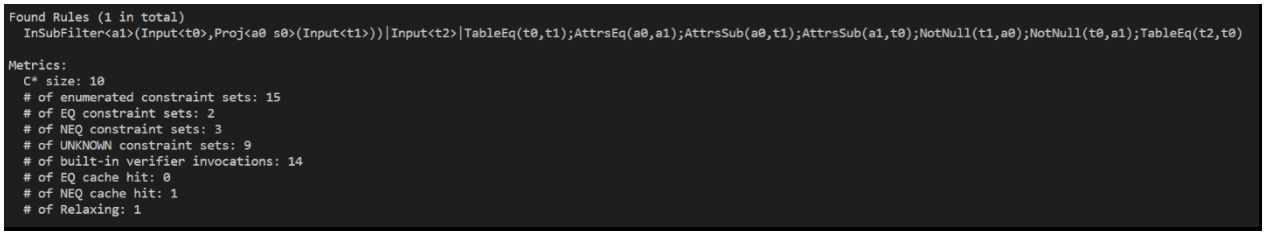


图 5.8　约束枚举的示例

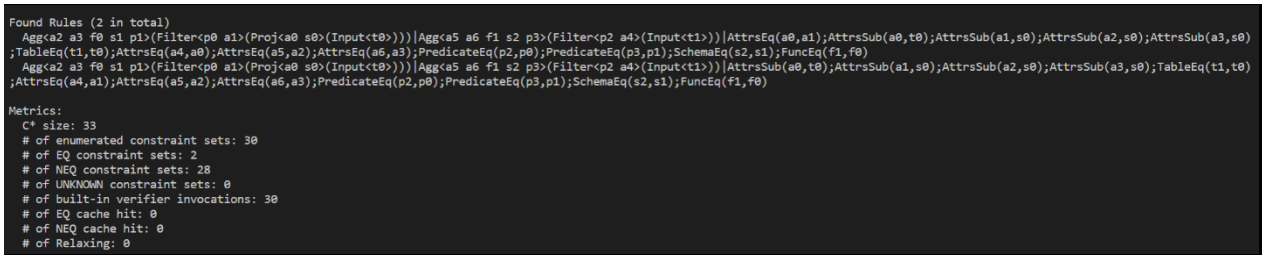


图 5.9　约束枚举的示例

6　总结与展望

6.1 总结

本文主要介绍了 WeTune，它可以自动发现 SQL 查询的重写规则。它列举了所有有效的逻辑查询计划达到一定的大小，以发现等价的计划的基础上，一个新的基于 SMT 的验证。本文将 WeTune 所得到的规则对查询语句进行重写，发现可以得到很大程度的查询重写优化，从而大幅提高了性能。

WeTune 可以自动的发现重写规则，在进行对规则正确性的验证后，应用到实际中也确实可以对数据库的查询语句进行重写，优化查询语句，减少查询时间，对 WeTune 减少冗余规则的创新改进，也对于之后 WeTune 增加聚合和 UNION 的查询功能后，再次进行减少冗余操作会大幅度减少所用的时间，加快找到有用的规则。

6.2 展望

（1）对于 WeTune 还有一些不足的地方，一个不足是内置验证器的不完全性。首先，由于∑运算符的无界性，U-expression 从根本上超过了 FOL 的表达能力。目前，只有部分情况可以被翻译成 FOL 并由 SMT 求解器自动验证。如何将任何 U-expression 自动转化为 FOL 公式是可以期待解决的。

（2）WeTune 翻译后的公式并不总是落入 SMT 求解器的可解码片段；因此可能导致超时，从而错过有用的规则。

（3）WeTune 不支持的 SQL 特性。WeTune 不支持递归查询。有些功能是不被支持的，比如 UNION。有些功能被部分支持，比如 NULL。WeTune 目前只考虑了 NULL 对部分操作符的影响。支持更多的特性是可以改进的。

参考文献

[1] Edmon Begoli,Jesús Camacho-Rodríguez,Julian Hyde,Michael J. Mior,Daniel Lemire. Apache Calcite[P]. Management of Data,2018.

[2] 郭丽英. 数据库中查询重写及基于遗传算法的多连接查询优化研究[D].沈阳:东北大学,2008.

[3] Zheng XiaoQing,Chen HuaJun,Wu ZhaoHui,Mao YuXin. Dynamic Query Optimization Approach for Semantic Database Grid[J]. Journal of Computer Science and Technology,2006,21(4).

[4] 王力,王成良.基于免疫遗传算法的关系型数据库查询优化技术[J].计算机系统应用,2008(01):72-75.

[5] G.Graefe, RL.Cole. Optimization of Dynamic Query Execution Plans [C]. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapllis, Minnesota, USA, May. 1994, 150-160.

[6] R.S.G.Lanzelotte, P.Valduriez. On the Effectiveness of Optimization Search Strategies [C]. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. Dublin, Ireland, 1993.24-27.

[7] Y.E.Ioannidis, E.Wong. Query Optimization by Simulated Annealing [C]. In Proc.of the 1987 ACM SIGMOD Conference on the Management of Data, San Francisco, CA , and May 1987

[8] Sorav Bansal, Alex Aiken. Automatic generation of peephole superoptimizers [J]. ACM SIGARCH Computer Architecture News, 2006, 34(5): 394-403.

[9] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. SPES A Two-Stage Query Equivalence Verifier [J]. arXiv preprint arXiv 2004.00481 (2020).

[10] GitLab[EB/OL]. GitLab. https://gitlab.com/gitlab-org/gitlab. 2021.

[11] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries [J]. arXiv preprint arXiv:1802.02229 (2018).

[12] Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semir-ings [C]. In Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 2007. 31-40.

[13] Wang Z, Zhou Z, Yang Y, et al. WeTune: Automatic Discovery and Verification of Query Rewrite Rules[C]//Proceedings of the 2022 International Conference on Management of Data. 2022: 94-107.