软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(11):2715−2730 [doi: 10.13328/j.cnki.jos.004549]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于算子的可扩展复杂事件处理模型*

孟 由[1,2,3], 栾钟治[1,2,3], 谢 明[4], 钱德沛[1,2,3]

[1](中德联合软件研究所(北京航空航天大学),北京 100191)

[2](软件开发环境国家重点实验室(北京航空航天大学),北京 100191)

[3](北京市网络技术重点实验室(北京航空航天大学),北京 100191)

[4](深圳市腾讯计算机系统有限公司,广东 深圳 518057)

通讯作者: 孟由, E-mail: mengyou0304@126.com

摘 要: 随着大数据处理的深入发展,系统单位时间内产生的数据日趋庞大,数据间的关联关系日趋复杂,这使得传统的"存储-查询"或者"发布-订阅"的方式无法很好地满足诸如故障监控、股票分析、医疗及生命保障等对大数据具有实时处理需求的系统.复杂事件处理技术实现的是将用户对特定的事件序列的查询需求映射到特定识别结构上.该结构从多个持续的数据流中分析并提取满足特定模式的事件序列.该技术能够很好地支持对大量数据进行实时在线分析.但由于在数据处理的过程中,系统不可能预置全部的查询语义,许多系统在使用过程中会需要使用新的语义,以查询新产生的模式.因此,一种支持扩展的语义的复杂事件处理模型是非常必要的.同时,现有的复杂事件处理模型仅针对某几类特定的查询进行描述以及优化,对整体模型缺乏统一描述,导致许多模型在多规则复杂查询的情况下效率欠佳.针对上述问题,提出了基于算子的可扩展复杂事件处理模型.该模型能够良好地支持现有的各类查询语义,具有较快的识别速度.基于该模型的形式化描述,对系统在识别过程中的性能消耗进行了详细分析,给出了模型构造最优算法.通过实验验证了算子模型优化方案的正确性.实验结果表明,经过优化后的树结构事件处理速度比开源复杂事件处理引擎 Esper 快 3 倍以上.

关键词: 事件处理;事件算子;实时处理;匹配树

中图法分类号: TP311

中文引用格式: 孟由,栾钟治,谢明,钱德沛.一种基于算子的可扩展复杂事件处理模型.软件学报,2014,25(11):2715−2730. http://www.jos.org.cn/1000-9825/4549.htm

英文引用格式: Meng Y, Luan ZZ, Xie M, Qian DP. Operator-Based extendable complex event processing model. Ruan Jian Xue Bao/Journal of Software, 2014,25(11):2715−2730 (in Chinese). http://www.jos.org.cn/1000-9825/4549.htm

Operator-Based Extendable Complex Event Processing Model

MENG You[1,2,3], LUAN Zhong-Zhi[1,2,3], XIE Ming[4], QIAN De-Pei[1,2,3]

[1](Sino-German Joint Software Institute (BeiHang University), Beijing 100191, China)

[2](State Key Laboratory of Software Development Environment (BeiHang University), Beijing 100191, China)

[3](Beijing Key Laboratory of Network Technology (BeiHang University), Beijing 100191, China)

[4](Tencent Computer System Co., Ltd., Shenzhen 518057, China)

Corresponding author: MENG You, E-mail: mengyou0304@126.com

Abstract: With the development of big-data computing, the system generated data becomes larger and more complex. Yet systems like fault monitoring, stock analyzing and health-care require processing these data in nearly real-time. The original data processing methods such as "save-query" and "publish-scribe" cannot handle the large volume of data in that speed. Complex event processing (CEP) is a data processing scheme that executes the user's real-time queries. It continually takes the high volume of raw data input and produces output

* 基金项目: 国家自然科学基金(61133004); 国家高技术研究发展计划(863)(2011AA01A203)
收稿时间: 2012-11-26; 修改时间: 2013-06-26; 定稿时间: 2013-12-05

for the corresponding data stream according to the queries. However in some practical environments, the data from system may generate many new patterns, and the CEP system cannot prepare for each of them. Consequently, an extendable CEP system is needed. Existing CEP work mainly focus on several special types of queries without a high level overview, therefore cannot easily guarantee the overall performances of the system. Though the NFA model poses a universal semantic, the scalability of the NFA model is still under discussed. To address these defects, an operator-based complex event processing model is proposed to support operator extension. In addition, a detailed analysis is conducted on time consumption of operator-based model and an optimal algorithm is presented. Finally, the correctness of optimization solutions is verified by experiments. Contrast experiments show that the optimized tree model is three times faster than open-source project Esper.

Key words: event processing; event operator; real time processing; match tree

随着大数据处理的深入发展,系统规模日渐扩大,系统需求日趋复杂,其产生的数据量也在不断增加.与此同时,在许多应用,诸如实时监控、股票在线分析以及医疗生命保障等对系统响应速度具有较高要求的系统中,针对大规模复杂数据进行快速处理乃至实时处理的需求更加突出.作为一种针对大量数据进行实时处理的技术,复杂事件处理实现的是将用户对特定的事件序列的查询需求映射到某个一般化算法上,从多个持续事件流中分析并提取满足特定模式的事件序列.

在复杂事件处理中,每一组到达的数据都被认为是一个事件,由用户输入查询条件,系统则根据查询语义,对输入的事件进行持续搜索,找出满足查询语义的事件组合.通过使用复杂事件处理技术,用户定制相应的查询规则,输出相应的事件组合作为查询结果.例如,股票在线分析系统就对数据的实时处理有较高的要求,因为股价波动速度较快,用户需要在极短的时间内针对符合买入、卖出条件的股票进行操作.因此,用户可以通过复杂事件处理技术探测股价波动、交易状况以及相关新闻等多种类型数据之间的关联关系,规定相应的买卖条件,一旦股票满足其规定的条件,即可及时进行相应的买入或卖出操作.此外,在实时系统监控中,将复杂事件处理技术应用于故障探测也具有明显的优势,国内的大数据处理领域较为突出的公司之一,腾讯也在尝试使用复杂事件处理技术对分布在全球各地的服务器进行故障检测.其原有系统通过简单的阈值设置以及逻辑组合的方式实现故障检测及报警,但与之俱来的是经常出现重复报警、无法追溯故障根源、故障信息列表难以维护等诸多问题.在改进后,将底层获取的基本信息(如计算资源监控数据、存储状况监控数据、网络访问状况监控数据)以及定制的基本事件(DOS 攻击事件、用户访问事件、查询操作事件)进行封装,作为事件输入到复杂事件处理的引擎.用户根据已知的故障模型定制查询语句,要求引擎在输入的事件中查找符合故障模型的组合.在故障检测实施过程中,引擎根据用户定制的查询语句构建识别模型,使底层获取的事件持续流入该模型.由该识别结构在这些事件中搜索符合查询条件的集合作为结果,一旦获得结果,就表示满足故障模型的事件已经出现,即,故障已经发生.该公司此前遇到的大多数问题,诸如重复报警、故障根源追溯等,都可以通过复杂事件处理技术中的克林闭包查询、时序查询等方式来解决,其他特殊需求还可以通过复杂事件处理的算子扩展等方式进行处理.作为流处理技术的一个分支,复杂事件处理技术正在更加广泛地应用于各类实时系统中.与此同时,当前的复杂事件处理技术面临如下的挑战:

- 海量输入数据的实时处理.单位时间事件的处理速度,是复杂事件处理系统最根本的性能保障.首先,随着系统规模日渐庞大,单位时间产生的数据规模也日渐增多.因此,足够高的吞吐率才能保证复杂事件处理引擎能够被有效地应用于大数据环境.同时,在实时处理需求较高的环境中,处理时间成为非常重要的衡量标准.如何保证在尽量短的时间内返回符合条件的查询结果,也是复杂事件处理技术的重要性能指标.例如,在云环境的监控中,由于每分钟的计算资源、存储资源以及带宽都被列为计费项目,一旦故障产生而不能及时被探测,将浪费大量的资源.
- 查询条件的可扩展性.在数据处理的过程中,系统不可能预置全部的查询模式,同时,许多系统在演化过程中会产生新的模式,一种支持扩展语义的复杂事件处理模型是非常必要的.例如,在股票在线分析系统中,用户除了需要软件查询已知的如股价 V 字波动以外,还需要根据自己的经验自行定制股价波动模式和相应的查询条件;在系统监控过程中,不同的系统存在许多特定的故障,其数学特征未必明确,探测这些故障也往往需要用户自行定制.因此,语义的可扩展支持是实时复杂事件处理非常迫切的需求

之一.

针对上述问题,本文提出一种基于算子的可扩展复杂事件处理模型.本文主要贡献如下:

(1) 提出了基于算子的事件匹配模型,将事件流的各种复合操作过程映射到一个基本算子上,并通过算子的匹配判定函数、复合计算函数以及输出界定函数描述事件流的复合过程;

(2) 通过建立算子与现有语义的映射,实现用算子的方式描述用户的查询规则,从而保证通过算子构建的复杂事件处理模型在满足查询规则的前提下支持可扩展性;

(3) 对算子以及由算子构建的查询规则树结构进行时间消耗评估,并基于此进行优化,得到算子树的最优构建策略,并在实验环节进行相关验证.

1 相关工作

有别于数据库的存储-索引-查询模式[1−4]以及查询订阅模式[5,6],复杂事件处理实现了对流数据的实时处理.在复杂事件流引擎的实际使用过程中,引擎首先根据用户的每条查询中的约束构建识别结构,保证流出该结构的数据符合查询中所有约束;而后,在实际运行过程中,由流数据不断流入识别结构,识别结构根据约束自行选取符合的数据[7].当前主要的识别结构包括 NFA 结构、树形结构、图以及 Petri 网等几类.

NFA 模型是根据时序查询条件生成一个非确定性状态机模型,每当出现符合查询条件的事件后,变更当前存在状态机列表的状态,使状态机到达最终状态的事件集合,即为一组满足查询结果的事件集合.该状态机的构建过程需要分析全部的时序约束之后方能完成.该模型主要存在的问题是构建算法必须考虑到所有的时序组合构建情况.例如,在 NFA 模型中,时序查询中的“否约束”无法被单独定义以及实现,只能通过与序列算子联合定义才具有实际意义;同时,新扩展的时序约束可能导致构建整个 NFA 算法发生改变,如克林约束,因此,该模型不具有良好的可扩展性.Cayuga[8,9]首先提出了 NFA 的代数逻辑以及基于其上的支持复杂事件查询的发布订阅系统.Diao[10−14]等人在其基础上提出了 SASE+系统,改进 NFA 模型为 NFAb[13]模型,使其更好地支持克林语义以及否语义,并在近年来关注于复杂事件处理的乱序问题[15].

树形复杂事件处理的实现则是根据每个查询规则生成一个树节点,数据流从树的叶子节点流入,在节点中复合生成新的事件,并向上流动直至根节点,即为输出结果.树形结构则因为结构灵活、扩展方便等特点而被关注,GEM[2]系统首先提出了树模型,其主要用于系统监控方面的应用.而后,MIT 的 Zstream[16]给出了树形构建的一般策略以及代价模型,但却没有明确给出定量化最优构建算法.因此,对于树结构如何充分利用中间节点找寻最优构建方式以及丰富语义仍有待研究.而后,在国内也有部分相关研究[5,6,17],这些研究工作都着重于系统内部性能的优化,在可扩展性方面没有涉及.由于树结构处理单元之间由数据流连接,可以很容易地将不同的处理单元分布到不同的节点进行处理.相比之下,前人的工作着重于根据查询语义实现识别结构,而并没有太多的规范一般化实现方法.针对具有新增查询语义需求时,往往需要更改整个识别结构.本研究虽然也是基于树结构的识别模型,但着重考虑复杂事件流引擎的一般化实现方法.通过使用每个算子代表一次事件流的复合过程,使算子能够作为构建识别结构的基本单元,并通过定义 F,Q,Z 这 3 个函数实现对算子的完整描述.

除此之外,其他实现工作则主要是在功能上进行了扩展,包括 ZStream[18],TelegraphCQ[19],Aurora[10]等.STREAM 提出了 CQL[18]语义,TelegraphCQ 则根据 PostgreSQL 实现了引擎原型,Aurora 则通过拖拽的方式实现了具有图形化的复杂事件处理引擎.综合而言,在复杂事件处理方面的研究工作已经较为全面,各种结构均有多种实现,但绝大部分都是仅针对特定的语义.NFA^b 模型虽然被证明了其具有完备性,但也没有考虑系统的扩展性问题.在性能优化上,优化算法也局限在性能调优的范畴,没有给出最优化算法.

2 事件模型

复杂事件处理技术最终解决的是将用户对特定的事件序列的实时查询需求映射到某个一般化算法上的过程,因此,本节首先清晰地界定事件、事件流、事件的复合以及事件流的复合等基本概念.

2.1 事件与事件流

事件原本用于描述状态的变化,在处理过程中被泛指为任意一组数据.系统往往把任何一组数据、一个网络上传播的数据包、一个类、一个 xml 文件,都认为是一个事件,用字母 *e* 表示.比如,在系统监控中,监控探针在某一时刻获取了某个机器的硬件当前状态,这个机器当前状态若附加了时间属性即可被认定为一个事件.在监控系统运行的过程中,探针会源源不断地从被监控系统中获取数据.因此,众多与该事件结构相同的数据会按照时间顺序不断到达监控中心,如同水管中的水不断流出.为了更好地表述这一特征,定义同类型的事件按照时间先后顺序排列的集合为一条事件流,用 *E* 表示.每个事件由 3 部分组成:起始时间、结束时间以及该事件所包含的其他数据.因此,定义一个事件 *e* 为

$$e=\langle startTime,endTime,data\rangle \tag{1}$$

其中,*startTime* 与 *endTime* 表征该事件的起始以及结束时间,*data* 表征该事件中其他属性.

而事件流 *E* 则表示为

$$E=\langle type, states\ \{e\}\rangle \tag{2}$$

其中,*type* 表征事件流的类型,为一个字符串,对于任意事件流 E_1,E_2,当且仅当 $E_1.type=E_2.type$ 时,称 E_1,E_2 是两个相同的事件流;$\{e\}$为事件集合,表征事件流 *E* 中的全部事件,该集合往往是一个无穷的集合;*states* 为一组 key-Value 对,用于表征事件流的一些状态特征.以后在不加说明的情况下,事件流均用大写字母表示,该事件流中的事件用与之相同的小写字母表示.

2.2 事件的复合运算

在监控、股票、医疗等领域,系统可获得诸如“CPU 变化”、“股价变动”、“病人心跳”变化等基本事件,但用户希望得到的往往是更加抽象的信息,如,“需要进行维护系统了”、“该在某时刻买入某股票”、“某病人需要呼吸机”等,这些事件往往不是一个可以简单地获取的变量,而是由众多数据或者事件根据特定条件组合的结果.

因此,复杂事件处理技术允许用户编写相应的规则.该规则描述了各种类型的事件如何组合成更加抽象复合的事件.由于这些抽象的事件具有新的含义,因此也往往被认定为新的事件.即,事件的复合是将多个事件按照一定的规则进行组合,生成新事件的过程.例如,“系统被 DDOS 攻击导致服务不可访问”事件是一个相对抽象的事件,实际上,该事件由多个“DDOS 攻击”事件以及“服务不可访问”事件组合形成;同时,“DDOS 攻击”事件同样由“网络监控报告”、“防火墙提醒”、“操作系统日志”等多个基本事件组合而成.图 1 就是将多个类型的事件进行组合以及抽象的过程.通过类似此类事件的层层抽象,系统可以直接向上层用户反映更加直观、具体的信息.用户通过特定的查询语句描述这些事件应当如何进行复合操作,并要求引擎返回符合条件的复合事件.

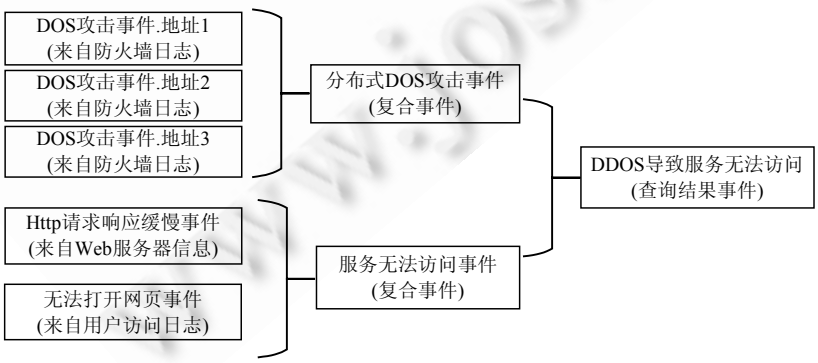


Fig.1 Join process of the events

图 1 事件的复合过程

2.3 事件流的复合运算

事件的复合仅适用于描述一组特定事件的组合操作,但用户的查询需求则是一个持续不断的过程.例如,查

询系统故障需要每当出现相应的特征后就产生警报,即需要持续不断地对输入的事件进行复合操作.因此引入事件流的复合的概念:将 1 个或多个事件流中的事件,持续不断地按照指定的方法进行复合,并持续地输出新事件的过程就是事件流的复合过程.新生成的事件构成了一条新的事件流,即,复合的结果流.

仍以查询"DDOS 攻击导致服务不可访问"为例,由图 1 的事件复合过程可知,用户需要查找一系列事件的组合.但是这些事件分别来自不同的组件,因此,我们可以简单地把每个组件输出的信息封装成一种事件类型,即,每个组件都输出一条事件流.用户的需求是,当出现"DDOS 攻击导致服务不可访问"时就产生报警,因此,引擎需要将流入的事件按照查询规则持续地进行组合,只有满足查询的事件组合才能被输出,作为结果.即,产生了一条新的流.其整个过程如图 2 所示.

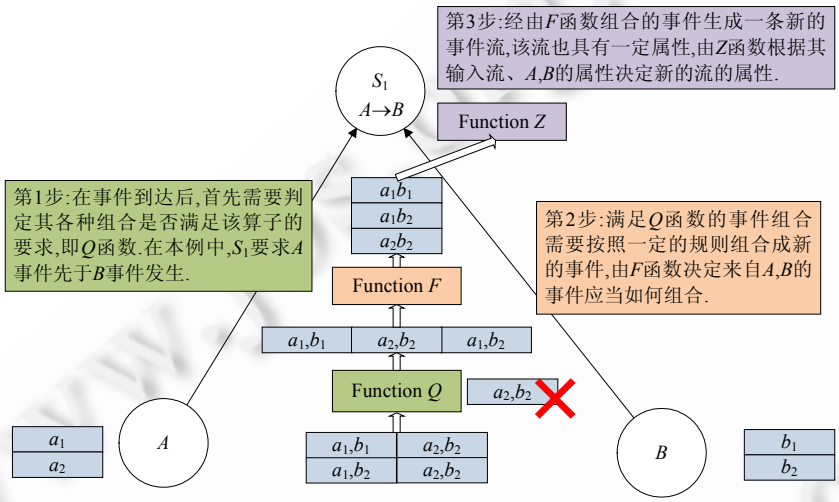


Fig.2 Implements of the operator S by function Q, F, Z

图 2 算子 S 通过函数 Q,F,Z 进行实现的方式

不同于图 1 的是,该系统不是仅仅将 1 组事件组合即可,而是持续不断地将每一组符合条件的事件全部组合.在具体的复合过程中,以一个简单的序列复合为例,查询 $A\rightarrow B$ 表示系统需要输出所有 $a\in A\wedge b\in B\wedge a.starttime<b.starttime$ 的事件.实现该查询需要构建一个能够持续处理的结构.该结构不断输入属于 A,B 的事件,并将其组合.具体而言,该过程需要 3 个函数:

(1) 由于属于 A,B 事件类型的事件不断到达,且要求二者需要满足符号"→"所代表的特定规则:$a.starttime<b.starttime$.因此在每次复合操作之前,都需要先由一个函数判定即将进行组合的这些事件是否满足进行复合的条件,此处即为 $a.starttime<b.starttime$,用事件匹配判定函数 Q 来表示.该函数输出布尔型结果,只有 Q 函数为真,才可以对这组事件组合进行复合操作.

(2) a,b 事件如果通过了事件匹配判定函数,则可以进行事件的复合计算,但二者应当如何组成新的事件?新的事件属性应当如何决定?因此,使用事件复合计算函数 F 描述对满足匹配函数的事件进行事件组合操作过程.

(3) a,b 事件复合之后,输出的事件是一种新的事件,这种事件的集合构成了一条新的事件流,通过使用流状态计算函数 Z 描述了新的事件流的相关属性.

更一般地来说,通过 Q,F,Z 可以描述事件流之间的一次复合过程.例如,$A\rightarrow B$ 在构建 A 与 B 的复合过程中,输入的是复合运算符"→",首先由流状态计算函数 Z 根据该符号确定复合之后新产生的事件流的属性、类型以及状态.在数据到达后,由匹配判定函数 Q 判定一组数据是否满足该符号的需求:若满足,则交由复合计算函数 F 生成新的事件;否则,事件被抛弃.新生成的事件作为一条新的事件流输出,该流的属性则由 Z 决定.为描述方便,我们使用 $S=\langle Z,Q,F\rangle$ 描述一次事件流的复合过程,我们称 S 为一个事件流复合算子,简称算子.

2.4 算子*S*的形式化描述

事件流的复合过程分为两部分,即构建过程和运行过程.构建过程规定了事件流中事件应当进行何种计算过程;而在运行过程中,事件流中的事件则按照规定的运算规则进行计算.假设事件流 $E_1,E_2,\ldots,E_n$ 通过算子 $S=\langle Z,Q,F\rangle$复合生成新的事件流 E_{new},则算子定义如下:

定义 1(算子定义). 若 $E_{new}=Z(E_1,E_2,\ldots,E_n)$,且对任意 $e_{new}\in E_{new}$,存在 $e_1,e_2,\ldots,e_k\in E_1,E_2,\ldots,E_n$,使得 $F(e_1,e_2,\ldots,e_k)=e_{new}$ 且 $Q(e_1,e_2,\ldots,e_k)$=True,称事件流 E_{new} 由 $E_1,E_2,\ldots,E_n$ 复合而成,映射关系 $S=\langle Z,Q,F\rangle$即为事件流的复合函数,或称为一个算子.函数 F 为算子 S 的事件复合计算函数,函数 Z 为算子 S 的流状态计算函数,函数 Q 为算子 S 的匹配判定函数.按照 Q,Z,F 这 3 个函数的具体职能,3 个函数的子函数定义见表 1.

Table 1 Operator S and its three sub functions

表 1 算子 S 及其 3 个子函数

算子的子函数	$Q=Q_{type}\cap Q_T\cap Q_A$	$Z=\langle Z_T,Z_V,Z_S\rangle$	$F=\langle F_S,F_E,F_A\rangle$
Q,Z,F 子函数定义或说明	Q_{type} 定义了关于类型中的匹配判定,即,与 $E.type$ 相关的判定.	$E_{new.type}=Z_T(E_1,E_2,\ldots,E_n)$	$e_{new}.startTime=F_S(e_1,e_2,\ldots,e_n)$
	Q_T 定义了基于时间的约束条件,即,e_i 与 e_j 之间的时间关系	$E_{new.value}=Z_v(E_1,E_2,\ldots,E_n)$	$e_{new}.endTime=F_E(e_1,e_2,\ldots,e_n)$
	Q_A 定义了两个事件流复合过程中,查询语句规定的事件属性应满足的约束条件判定,即谓词约束判定.	$E_{new.stat}=Z_s(E_1,E_2,\ldots,E_n)$	$e_{new}.data=F_A(e_1,e_2,\ldots,e_n)$

3 基于算子的树结构复杂事件处理模型

本节主要介绍如何通过用户的查询条件构建事件识别模型的过程.通过算子与用户查询的映射过程,说明用户的查询均可通过算子的方式进行描述;在构建识别模型的过程中,通过将算子实现成为树节点的方式,构成树形复杂事件处理模型.

3.1 用户查询中约束与其实现

用户的查询以众多约束组合的形式呈现,随着应用需求日趋广泛,查询约束种类繁多,按照查询对构建系统的影响程度,可分为时序约束和谓词约束两类.时序约束是复杂事件处理中特有的一类约束,其包括序列约束、否约束(negative)、或者克林(Kleen)约束以及窗口约束(window)等,这些约束往往对事件的时间有一定要求;谓词约束则与数据库查询类似,包括比较、排序以及各种聚合函数等.

下面以一条系统监控中用于稳定性保障的查询语句为例进行说明.管理员可以通过该句侦测出“由于带宽不足而强行取消用户服务”的情况.其具体查询如下:当 90%的用户操作响应时间超过 100 ms 之后,继而出现平均数据传输带宽小于 128 KB/S 的情况,而后出现用户取消请求,则输出相应事件集合.其描述如下:

select *percentile*(*A.responseTime*,90),*avg*(*B.inTransferRate*)

from *pattern*[(*A*=*ClientPMs*(*appId*=‘*app*1’)→(*B*=*DataTransPMs*(*endpoint*=‘*zeus*−1’))→(*C*=*ClientCancel*(*appId*=‘*app*1’)))].*win*:*time*(5min)

having *percentile*(*A.responseTime*,90)>100 **or** *avg*(*B.inTransRate*)<128

限于篇幅,有关于语法解析方面的内容将不再复述.针对上述查询,其包含的约束可按照如下方式分类:WE_0 为时序约束,其他的 WE_1,WE_2,WE_3 为普通的谓词约束.输出的结果应该是全部满足如上约束的事件集合.下面对两类约束的实现分别进行描述:

输入的事件流:*A*(*ClientPMs*),*B*(*DataTransPMs*),*C*(*ClientCancel*)

约束: WE_0:(*A*→*B*→*C*).*win*:*time*(5min)

WE_1:*A.appId*=‘*app*1’

WE_2:*B.endpoint*=‘*zeus*−1’

WE_3:(*pertile*(*A.responseTime*,90)>100 **or** *avg*(*B.inTransRate*)<128)

3.1.1 时序约束

时序约束用于描述被查询事件在时间上应满足的关系.目前,时序约束的种类有很多,不同的事件流查询引擎实现的约束方式也不尽相同.时序约束除了被单独使用以外,很多查询中还会联合使用多个时序约束.比如在上例中,约束 $WE_0=(A\rightarrow B\rightarrow C).win:time(5\text{min})$是 2 次序列约束以及 1 次时间范围约束的组合.其寻找的是 A,B,C 这 3 种类型的事件先后发生并且其间的时间间隔小于 5 分钟.另外,这些事件同样应该满足其他约束,包括 WE_1, WE_2 和 WE_3.虽然没有明确指出,但时序约束之间都用"与"进行链接,即,要求事件同时满足所有的时序约束.

3.1.2 谓词约束

类似于数据库处理中的 where 部分查询,谓词约束同样也是用户对事件的筛选条件.谓词约束往往被表示为由多个 and 以及 or 连接而成的布尔表达式,并使用事件中的多个属性进行聚合、比较等操作.我们将整个查询条件转化为多个 *WE* 通过与(and)关系连接,其目的是方便后面的数据处理.

我们定义谓词约束为如下形式:

$$WhereQuery::=WE\ \{\text{and}\ WE\}^* \tag{3}$$

$$WE::=BoolExpression\ \{\text{or}\ BoolExpression\}^* \tag{4}$$

其中,*BoolExpression* 是一个最基本的布尔表达式,其可以是几个简单的比较表达式,如约束 WE_1:*ppId*='*app*1',也可以是由聚合函数组成的比较表达式,例如 WE_3:*avg*(*DataTransPerfMs.inTransRate*)<128.

3.1.3 用户查询的算子描述

时序约束与谓词约束事实上都是对事件属性的限定条件,但在实现上却有极大的区别:

- 首先,谓词约束是对事件除时间外的属性的约束,针对谓词约束的实现,早在数据库理论中就有诸多研究工作,算法实现效率高,能够进一步优化之处较少;时序约束是对一组事件的时间进行限定,而事件则基本都是按照时间先后顺序到达,因此,算法有较大的优化空间.例如,序列约束 $A\rightarrow B$ 要求 *A* 类型事件先于 *B* 类型事件先到达,因此,系统只需在接收到每个 *A* 类型事件后,将其与所有在其后到达的 *B* 类型事件组合作为结果即可.而不像谓词约束的 Join 操作,需要对所有事件进行全匹配.因此,本部分重点研究时序约束的实现,而针对谓词约束,则采用公认的高效实现算法.
- 其次,时序约束会将原有的事件集合组合成新的事件,因此在计算顺序上存在一定的先后关系,但谓词约束中往往没有此类限定条件.例如上例中的时序约束 $WE_0=(A\rightarrow B\rightarrow C).win:time(5\text{min})$,其中,序列约束 $A\rightarrow B\rightarrow C$ 可被看作事件流 *A* 与 *B* 复合后形成的新的事件流再与 *C* 流复合,即$(A\rightarrow B)\rightarrow C$;也可以看作是 *B* 与 *C* 复合后再与 *A* 复合的过程,即 $A\rightarrow(B\rightarrow C)$.虽然这两种计算方式最终都会获得同样的结果,但是不能存在除此之外的其他计算方式.且其最终结果都要通过事件范围约束 *win*:*time*(5min),该约束限定的是 $A\rightarrow B\rightarrow C$ 的结果,即 *ABC* 流复合的结果流.由于时序约束进行的是事件流的复合过程,因此考虑使用算子对时序约束进行描述.

图 2 是通过算子 *S* 的 3 个子函数,即事件匹配判定函数 *Q*、复合计算函数 *F* 以及流状态计算函数 *Z* 来实现时序约束的过程.通过更改算子 *S* 的 3 个子函数,可以对现有的时序约束进行充分的描述.为了验证其表述能力,表 1 列出了使用算子 *S* 的方式描述 NFA^b 中所使用的时序约束.由于 YanleiDiao 等人已经证明 NFA^b 模型的语义完备性[13],可推得通过算子描述的时序模型同样具有语义完备性,即,算子描述模型可支持任何现有时序语义.由于时序约束可以统一地通过算子 *S* 进行描述,上例中的时序约束 WE_0 用中缀表达式可表示为

$$S_{win}(S_{\rightarrow 2}(S_{\rightarrow 1}(A,B),C)).$$

对谓词约束而言,谓词约束仅仅包含对事件中属性的要求,因此,谓词约束的实现并非新添加一个算子,而直接将谓词约束添加到某个已经存在的算子中即可.由于算子 *S* 的复合匹配函数 *Q* 用于验证数据是否满足算子的要求,所以可直接更改其 *Q* 函数为 $Q_{new}=Q_{old}\wedge WE$,保证了通过该算子的数据都是满足了谓词约束的数据.以第 3.1 节的用户查询为例,对事件流 *A*,*B* 的复合过程 $A\rightarrow B$ 而言,系统需要构建一个算子 S_1 完成该过程.谓词约束 WE_1,WE_2,WE_3 均为对 *A*,*B* 流中的事件的约束,因此,系统生成一个算子 S_1 实现"$A\rightarrow B$"时赋予 $S_1.Q=A.endtime<B.starttime$,用于保障每个 *B* 事件都发生在 *A* 事件之后.在实现谓词约束时,需要添加约束 WE_1,WE_2,WE_3 到算子

S_1 中.按照前述方法,算子更改后的匹配函数改为 $S_1.Q_{new}=A.endtime<B.starttime\wedge WE_1\wedge WE_2\wedge WE_3$.此时,$A,B$ 事件流中的事件如果需要通过算子 S,则必须满足 $S_1.Q_{new}$,从而保障了输出的事件组合既满足时序约束 S_1,也满足谓词约束 $WE_1\wedge WE_2\wedge WE_3$.

3.1.4 算子的扩展支持

本节开始时列举了部分时序约束,但实际使用的时序约束并非只包括如上这些查询.在使用过程中,很多查询可能与这些查询类似,却在细微位置有部分更改.如下列出两例进行说明:

例 1:在去除重复告警事件的过程中,可使用扩展克林约束对完全相同的告警事件进行过滤,但由于告警事件的数量不确定,不能直接使用扩展克林约束中的参数"n",而应该要求该约束匹配任意数量连续地报警.此外,扩展克林约束会组合全部的同类型事件之后才组成一个新的事件并输出,即,告警信息会在所有重复告警全部到达后才被输出,此时与第 1 个告警信息到达时间相比已经延后了很长时间,在实际使用中,应该在第 1 个告警事件到达时就进行报警,因此需要在扩展克林的原有语义上进行修改或扩展.

例 2:在一般的序列约束 $A\rightarrow B$ 中,A,B 事件会构成新的事件,且新事件以 A 事件的起始时间为开始时间、以 B 事件的结束时间为结束时间.但在股票价格分析过程中,其数学模型均以单个时间点作为事件触发条件,不存在时间段的概念,因此在金融领域,探测双峰波动等事件时,不能使用原有的序列约束,需要对序列约束的部分实现进行扩展或调整.

以上两例描述的仅仅是两个较小的改动需求.事实上,虽然第 3.1.1 节描述的几个约束组合已经被证明具有完备性,但由于实际使用中存在的差异,用户对时序约束的需要也在不断变化,需要对原有约束进行扩展和调整.而算子 S 的 3 个子函数 Q,F,Z 可对时序约束进行充分描述,因此,用户可直接更改 S 的 3 个子函数以实现新的时序约束.例 1 中的需求可通过在原有克林闭包算子的基础上修改其匹配判定函数 Q 进行实现.例 2 中对序列约束的需求可通过修改其复合计算函数 F 进行实现.这两个新实现的算子对应于两个新的时序约束.通过更改算子 S 的 3 个子函数,可对算子 S 表述的语义进行扩展.

3.2 树结构的算子模型

通过单个算子 S 可实现单个的时序约束,但由于用户查询往往由多个时序约束组合而成,因此需要一个统一结构对事件进行处理.本节描述如何将实现单个时序约束的算子构建出统一的树形识别结构,注意,此时对于单个算子而言,其对树结构的影响只有其 3 个子函数 Q,F,Z,因此无论是系统内置的时序算子还是用户自行添加的算子,在构建树结构的过程中都是完全相同的.

3.2.1 树模型运行样例分析

由于复杂事件处理技术最终将多个基本事件组合成 1 个复合事件,是一个自底向上的汇聚过程.每个实现了时序约束的算子作为树的分支节点,而流入该引擎的事件类型名作为树的叶子节点.如图 3 所示为针对查询 $E_{query}=S_{win}(S_{\rightarrow 2}(S_{\rightarrow 1}(A,B),C))$构建的树形结构,输入系统的事件类型名 A,B,C 全部作为树的叶子节点,算子 S_{win},$S_{\rightarrow 2},S_{\rightarrow 1}$ 作为树结构的分支节点.节点间通过边相连,若某个节点 M 以另一个节点 N 的输出事件流作为输入,则有一条从 M 节点到 N 节点的边.在运行阶段,分别属于 A,B,C 类型的事件 a,b,c 事件持续从 A,B,C 节点持续流入,沿树形结构的边向上流入到分支节点进行数据处理.流入的数据首先由匹配判定函数 Q 检测是否需要进行复合操作,Q 函数判定无误后交由复合计算函数 F 将流入的事件复合成新事件,新事件的属性则有 Z 函数根据流入事件的属性确定.最终,从根节点流出的事件即为查询的结果.图 3 演示了在运行阶段数据的流向过程,分别属于 A,B 的事件经由自己所属类型的叶节点流入引擎,沿树结构的边流入节点 S_1.该节点由序列约束实现,因此要求 a 事件发生在 b 事件之前.输入的事件 a_1,a_2 与 b_1,b_2 进行全匹配之后,生成 4 种组合,并由 S_1 的 Q 函数判定.由图 3 可知,事件发生的先后顺序为 a_1,b_1,c_1,c_2,a_2,b_2,因为 a_2 发生在 b_1 之后,不满足该条件,因此 a_2b_1 的组合无法通过 Q 函数,而其他 3 组数据则通过该函数,按照 S_1 的事件复合计算函数 F 生成 3 个新的事件组合:a_1b_1,a_1b_2,a_2b_2.这 3 组事件组合成新的事件并向节点上方流动,到达 S_2.S_2 同样是序列算子,这 3 个新的事件与 c_1,c_2 进行全匹配后交由 S_2 的 Q,F,Z 这 3 个函数进行处理.输出的结果同样按照树结构的边流向下一算子,如此往复.最终仅有 1 组事件符合全部的查询约束 $a_1b_1c_1$,即为查询结果.

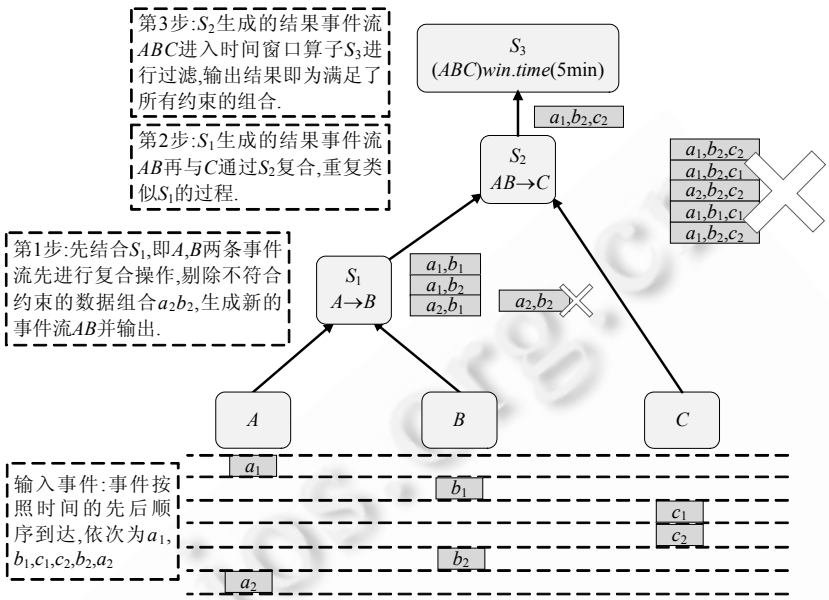


Fig.3 Operator-Based tree model implements

图 3 基于算子的树结构模型实现

3.2.2 树模型构建一般化方法

上节通过查询 $WE_0=(A\rightarrow B\rightarrow C).win{:}time$(5min)演示了树形识别结构的构造方法以及运行过程.更一般地来说,对于任意用户查询,最终通过语法解析转化为算子的表示形式 $E_{query}=S_1S_2S_3\ldots S_m(E_1,E_2,\ldots,E_n)$,其中,$E_1$~$E_n$ 是输入系统的事件流,S_1~S_m 是通过算子方式描述的时序约束.这些约束可能是现有的时序约束,也可以是用户自行扩展的时序约束.给定算子计算的先后顺序 $SP=S_{i_1}S_{i_2}\ldots S_{i_m}$ $(i_1,i_2\ldots i_m\in[1,m])$,存在唯一的识别树结构与之映射.具体构造算法如下:

算法 1. Construction and runtime algorithm of factor-based tree model.

1. Date Streams $E\leftarrow\{E_1,E_2,E_3,\ldots,E_{n-1}\}$
2. Factors $S\leftarrow\{S_1,S_2,S_3,\ldots,S_n\}$
3. User Query $E_C=S_1S_2S_3\ldots S_m(E_1,E_2,\ldots,E_n)$
4. Factor Compute Priority $SPattern=S_{i_1}S_{i_2}\ldots S_{i_{n-1}}$; $i_1,i_2,\ldots,i_{n-1}\in[1,n-1]$

   //Tree building process
5. **for** j **in** $\{i_1,i_2,\ldots,i_{n-1}\}$
6. $E_{new}\leftarrow$make tree node use factor S_j and related event stream E_j, E_{j+1}
7. $E_{new}.property=S_j.Z(E_j,E_{j+1})$
8. $E_j\leftarrow E_{new}$
9. $E_{j+1}\leftarrow E_{new}$
10. //Runtime for each event processing in tree model
11. **if** S is father node of E **and** $e\in E$
12. **if** $S.Q(e)$==true
13. $e_{new}=S.F(e)$

4 复杂事件处理模型性能分析与优化

在复杂事件处理中,作为实时数据处理引擎,衡量效率的最重要标准为吞吐率,即,单位时间能够处理的事

件数量.因此,本节着重于针对处理事件的速度进行分析,研究不同的算子组合结构对系统处理事件的速度造成的影响,并提出了最优的树形构造模型.

4.1 单个算子内部的时间消耗

整个树结构都由多个算子构成,因此,首先需要衡量各个算子在处理事件的时间消耗,用字母 C 表示.因为考虑时间消耗,因此单位为 ms.下面分别讨论算子 S 的 3 个子函数:Q,Z 以及 F 对应的消耗函数 C_Q,C_Z,C_F.

4.1.1 基本单位定义

首先定义几种基本操作的时间消耗,推导获得时间消耗以及部分匹配函数的通过概率定义,见表 2.

Table 2 Meaning of the basic variables

表 2 基本变量定义

数据类型	数据含义	消耗的时间的字符表示
时间消耗基本单位	进行一次二元比较的时间消耗	N
	获取事件的一个属性时间消耗	u
	生成一个新的事件的时间消耗	j
概率相关数据	通过类型判定函数 Q_{type} 的概率	P_{Q_t}
	通过时间判定函数 Q_T 的概率	P_{Q_T}
	通过属性判定函数 Q_A 的概率	P_{Q_A}

结合表 1 中算子 3 个子函数的定义,每个算子处理一组数据的时间消耗应该为其 3 个子函数的时间消耗之和.因此,如果用 C_Z,C_Q,C_F 表示算子 S 的 3 个子函数对一组数据的时间消耗,因为 Q 函数和 Z 函数对通过的每组数据都适用,但只有通过 Q 函数的数据才能进入 F 函数进行组合计算,假设通过 Q 函数的概率为 P_Q,则总消耗 C 可表示为

$$C=C_Q+P_QC_F+C_Z \tag{5}$$

下面分别计算 3 个函数的时间消耗,限于篇幅,直接给出 3 个函数时间消耗如下:

Q 函数的总时间消耗为 $C_Q=C_{Q_t}+P_{Q_t}C_{Q_T}+P_{Q_t}P_{Q_T}C_{Q_A}$,$Q$ 函数的通过概率 $P_Q=P_{Q_t}P_{Q_T}P_{Q_A}$,其中,$Q_{type}$ 为类型判定函数.因为类型验证在识别树构建过程中完成,一个算子的输入一定满足其输入要求判定.因此,运行时此判定被忽略 $C_{Q_t}=0,P_{Q_t}=1$. Q_T 为时间判定函数,除了 Conjunction 以及 Disjunction 之外,其他算子都需要进行一次时间的比较,$C_{Q_T}=n+2u,P_{Q_T}=1/2$. Q_A 为属性判定函数,即,加载算子上的谓词约束.每个算子的 Q_A 中谓词约束条件各不相同,假设某算子有 n 个谓词约束条件,$Q_A=WE_1$ and WE_2 and …. and WE_n,通过的概率分别为 P_1,P_2,P_3,…,P_n,每个的时间消耗为 C_1,C_2,C_3,…,C_n,则数据通过全部谓词约束的概率为

$$P_{Q_A}=\prod_{i=1}^{n}P_i \tag{6}$$

在约束匹配过程中,谓词约束仅需要对数据进行属性判断,因此谓词约束的时间消耗基本相同,即 $C_W=C_1=C_2=\ldots=C_n$,其占用时间的期望最小值为

$$C_Q=C_{Q_t}+P_{Q_t}C_{Q_T}+P_{Q_t}P_{Q_T}C_{Q_A}=n+2u+C_w\left(1+\sum_{i=2}^{n}\prod_{j=1}^{i-1}P_j\right) \tag{7}$$

通过匹配函数的概率,即,通过概算子的概率为

$$P=P_Q=P_{Q_t}P_{Q_T}P_{Q_A}=1\times1/2\times\prod_{j=1}^{n}P_j \tag{8}$$

特殊情况下,当概算子没有谓词约束条件时,即 $P_i=1$,$C_W=0$,带入得:$C_Q=n+2u$,$P_Q=1/2$.复合计算函数 F 的运行是在匹配函数全部运算完毕且全部通过之后,根据表 1,F 由 3 个子函数 F_S,F_E,F_A 构成,但因为 3 个函数较为简单,故直接给出 $C_F=C_{F_S}+C_{F_E}+C_{F_A}=2n+j$.

4.1.2 综合计算

对于一组用于匹配的数据,消耗的时间对于匹配最复杂的节点为

$$C = C_Q + P_Q C_F + C_Z = n + 2u + C_w\left(1 + \sum_{i=2}^{n}\prod_{j=1}^{i-1} P_j\right) + n\prod_{i=1}^{n} P_i + j\prod_{i=1}^{n} P_i / 2 \tag{9}$$

通过的概率与公式(8)相同,其中,u,j,C_W均为恒定值,P_j为各谓词约束通过的概率.当且仅当$P_1<P_2<P_3<\ldots<P_n$时,公式(9)中的第 2 项达到最小.由此得出:

优化策略 1. 当一个算子存在多个约束条件时,应当先判断通过概率低的约束条件.

4.2 匹配树COST值计算

我们已经获得单个算子进行一次匹配所消耗的时间 C,以及单个节点通过概率 P.在整个识别树的运行过程中,每个算子构成一个中间节点,大量数据从叶节点流入,匹配成功后向上层父节点发送.每个中间节点S_i耗的时间总量为匹配数据量Num_i与每次匹配消耗的时间C_i之积,即$Num_i \times C_i$,节点的通过概率为P_i,该节点输出的数据量为该节点输入的数据量与通过概率之积$Num_i \times P$.对于模式匹配查询$E_C=E_1S_1E_2S_2E_3\ldots S_{n-1}E_n$,当算子表达式$SP = S_{K_1}S_{K_2}\ldots S_{K_{n-1}}$时,模型下总时间消耗为各个节点消耗时间之和.

$$Cost = \sum_{i=1}^{n-1}\left(Num_i\prod_{j=1}^{i}(P_{ij}) \times Num_{i+1}\prod_{j=1}^{i}(P_j) \times S_i \times C_i\right) \tag{10}$$

其中,若存在$Q_1,Q_2,\ldots,Q_i \in [K_1,K_i]$且分别为$i,i+1,i+2,\ldots,j-1$或$j,j+1,j+2,\ldots,i-1$,则$P_{ij}=P_j \times Num_i$;否则,$P_{S_{ij}}=1$.由定义知,$P$函数恒小于 1,且 C 函数值基本恒定不变可知,显然,只有当$P_{i_0} < P_{i_1} < \ldots < P_{i_n}$时,$cost$值达到最小.因此有:

优化策略 2. 在分配谓词约束时,应当尽可能地分配到先结合的算子上.

4.3 基于COST模型的优化树构建策略

由于即使对于同一个算子语义表达式$E_C=E_1S_1E_2S_2E_3\ldots S_{n-1}E_n$,根据算子计算式 $SPattern$ 的不同,其构造的树结构也各不相同.根据算子结合序列 SP 的定义,其构建方式共有$(n-1)!$种,现给出基于算子的匹配树最优构建策略如下.由于基于 SP 的树构建策略已经在第 3 节给出,因此此处仅给出如何搜索得到最优 SP 的策略:

算法 2. Construction of most efficiency algorithm of factor-based tree model.

1. Date Streams $E \leftarrow \{E_1,E_2,E_3,\ldots,E_{n-1}\}$
2. Factors $S \leftarrow \{S_1,S_2,S_3,\ldots,S_n\}$
3. User Query $E_C=E_1S_1E_2S_2E_3\ldots S_{n-1}E_n$
4. $WhereQuery=W_1$ and W_2 and W_3 and … and W_k
5. //Tree building process
6. for W_j in $WhereQuery$
7. for E_i in E
8. if W related to E
9. $Q_i = Q_i \times Q_{w_j}$
10. for S_i in $\{S\}$ order by Q_i ASC
11. $SPattern+=S_i$
12. Make Tree use $SPattern$

5 性能评测

本节将通过实验验证在第 4 节模型性能分析中获得的几个优化结论以及最优算法的正确性,而后通过自行开发的引擎 Cesar 与 NFA 结构的开源项目 Esper 进行性能比较.我们通过实验论证影响 Cesar 性能的主要因素,并优化其运行效率.所有实验全部使用同一套实验设备:IBM 刀片服务器 HS21,8 核 Intel(R) Xeon(R) CPU E5405@2.00 GHz CPU,8 G 内存,132 GB SCSI 硬盘,操作系统 CentOS5.2,内核为 2.6.18-164.11.1.el5xen #1 SMP,JDK 版本为 1.6.0_01-b06.实验用例则采用分布式系统监控场景,事件中的数值分布采用中国国家网格 CNGridEye 中两个主节点(中国科学院计算机网络信息中心以及上海超级计算中心)2011 年系统监控数据,其中

包括集群内部网络使用情况、队列使用情况、主机运行性能、作业完成情况、请求应答信息以及应用运行状况这 6 类数据,依照如上分类封装为 6 类事件,总计约 600 G 条,21 TB.

5.1 性能优化对比

由于采用统一的系统,因此计算量消耗与消耗时间基本成正比.在性能优化当中,我们主要对比不同的策略下,系统的总时间消耗.具体而言,时间消耗主要包括生成事件所用时间(t_1)、获取属性所用时间(t_2)、谓词判断所用时间(t_3)这 3 部分.

5.1.1 数据量对优化策略的影响

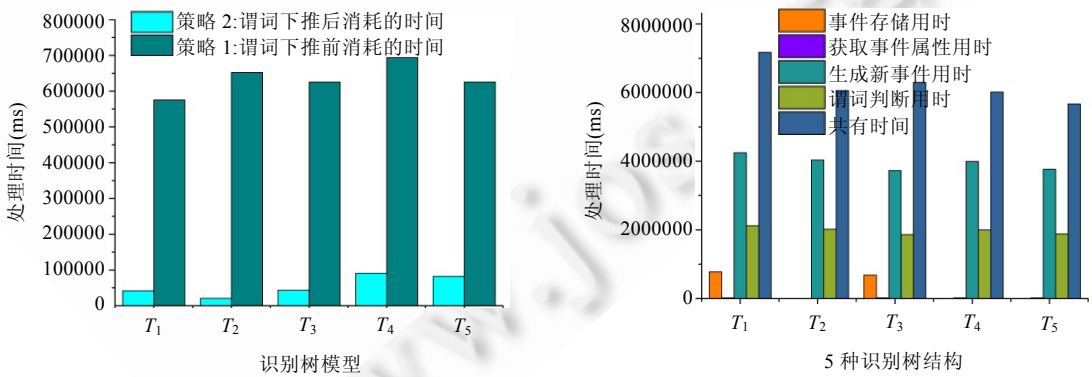
本实验验证输入事件的比例不同是否会对识别效率造成影响.首先验证在不存在谓词约束的情况下,树结构的构建策略.值得说明的是,在实际使用中,都会使用窗口算子对数据的到达事件进行约束,因此输入事件集合不会上升至无限大,因此,本实验中,A,B,C,D 都输入 1 000 个事件,结果事件集在千万级.针对时序约束 $A\rightarrow!B\rightarrow C^*\rightarrow D$,其查询的是 $A\{C\}^nD$ 这种模式,并且要求 A 与 C 之间不能存在任何事件 B.由于否算子“!”以及克林算子“*”为一元算子,所以必须先进行计算.基于算子的识别树有 5 种(共 6 种,但其中有两种是等同的)构建方式,定义其名称为 T_1~T_5,见表 3.并且,A,B,C,D 这 4 种事件数据量相同,均为 10 000 条,在不添加其他谓词约束的情况下,系统将产生 10000^4 组中间结果.

Table 3 Names of the match trees with different construction form

表 3 不同构造结构的识别树结构命名

选择的谓词约束	名称
$((A\rightarrow!B)\rightarrow C^*)\rightarrow$D	T_1
$(A\rightarrow!B)\rightarrow(C^*\rightarrow D)$	T_2
$(A\rightarrow(!B\rightarrow C^*))\rightarrow D$	T_3
$A\rightarrow((!B\rightarrow C^*)\rightarrow D)$	T_4
$A\rightarrow(!B\rightarrow(C^*\rightarrow D))$	T_5

T_1~T_5 这 5 种时间消耗如图 4(b)所示,由于使用数据量较大且没有谓词约束,所以匹配事件较长,在实际使用情况中不会出现此类情况.



(a) 谓词下推对识别效率差生的影响 (b) 没有谓词约束时,不同构造方式的识别效率差异

Fig.4 Match efficiency influenced by different predicate constraints

图 4 不同的谓词约束对识别效率的影响

由表 1 可知,带有克林算子的事件通过概率也为序列算子的 1/n(在此,根据概率计算,n 约为 5),以上问题即演化为 $PattenQuery=A\rightarrow B\rightarrow C\rightarrow D$,但 A,B,C,D 这 4 类事件的输入比例各不相同的问题.

根据第 4 节的结论,应当先计算通过概率低的算子.因为没有其他的谓词约束条件,4 种算子同为序列算子,通过概率相同.因此,T_1~T_5 这 5 种识别模式的事件消耗相近,最高效率与最低效率相差为 21.7%.具体而言,消耗

时间最多的是生成新事件消耗的时间,其次为谓词判断所用时间.在此,虽然已经没有人为添加谓词约束,但事实上,因为序列算子 $A\rightarrow B$ 本身就要求 $A.endtime<B.starttime$,所以在事件复合的过程中,仍然需要进行一定的属性比较,这一部分耗时也加入到谓词判断当中.

5.1.2 谓词约束放置位置对效率的影响

为了只验证谓词约束放置的不同位置给系统带来的效率差异,我们采用最普通的查询模式 $A\rightarrow B\rightarrow C\rightarrow D$.此时,加入谓词约束 WE_1~WE_6,其具体信息见表 4,相关事件类型表示该约束使用的事件.如 $WE_1=A.a_1>B.b_1$,由于在生成 a_1 以及 b_1 的过程中应用同种构造函数,因此,WE_1 实现的概率为 1/2.其他约束条件同样具有类似的性质.

Table 4 Features of the basic predicate constraints

表 4 基本谓词约束特征

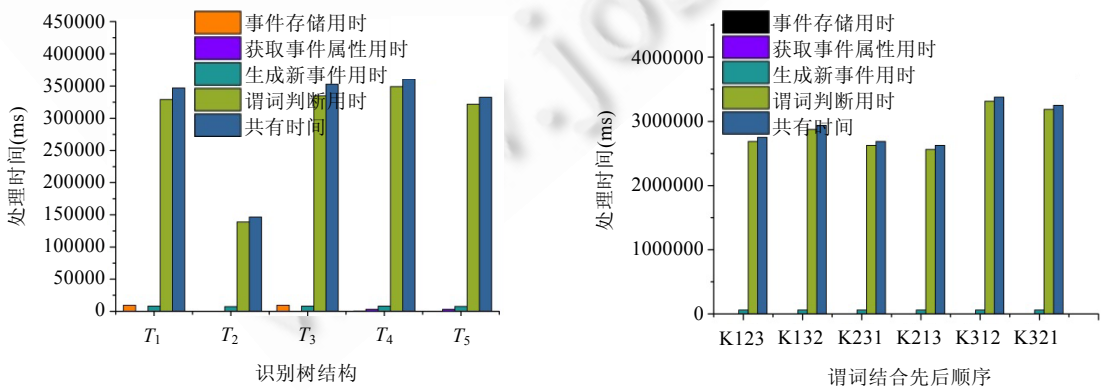
谓词约束名称	WE_1	WE_2	WE_3	WE_4	WE_5	WE_6
相关事件类型	*A*, *B*	*A*, *B*	*A*, *B*	*B*, *C*	*C*, *D*	*B*, *D*
通过概率	1/4	1/8	1/2	80%	1/4	1/16

- 加载谓词约束的算子选择

谓词约束放置的位置将在很大程度上影响识别效率,因此,本实验验证谓词约束选择的不同位置的算子给识别效率带来的影响.任意添加谓词约束 WE_3,WE_4,WE_5,WE_6 中的 3 种,按照第 4.3 节给出的最优构建算法,对于 $PattenQuery=A\rightarrow B\rightarrow C\rightarrow D=A,S_1,B,S_2,C,S_3,D$,无论使用哪种构造模式,在最简单的情况下,我们可以将全部约束记载在最后计算的算子上,因为概算子已经聚合了所有的事件,必然能够判断所有的约束,记为策略 1.同时,也可以尽可能地将算子放置在较低的位置,即,一旦事件中包括了谓词约束的全部事件后,即对谓词约束进行判断,如,因为 WE_3 仅包含事件 AB,因此 S_1 算子一旦进行计算,则必然包含 AB 事件,因此可以加载谓词约束 WE_3;同理,S_2 加载 WE_4,S_2 加载 WE_5,WE_6,记为策略 2.

对比两种策略下各个算子的时间消耗,如图 4(b)所示,策略 2 消耗的总时间明显要小于策略 1.具体而言,如图 4(b)所示,在识别树 T_3 模式下,策略 1 在 S_1,S_2 算子上消耗时间很少,由于前两个算子上没有约束条件,数据将全部两两匹配,达到 N^3 的程度,由于谓词约束全部加载在 S_3 上,因此要进行 N^3 的匹配.相比之下,策略 2 在开始的两算子即进行数据筛选,在开始就由部分数据被筛选,因此虽然前两个算子时间消耗较多,但极大地减小了最后一个算子的数据量,从而达到整体最优.

更一般地来说,我们将 T_1~T_5 这 5 种构建模型在谓词约束 WE_1~WE_6 下分别进行测试,所获得的效率如图 5(a)所示,使用谓词约束下推都获得了明显的效率提升.提升效率最不明显的 T_4 提高效率也在 7.6 倍左右,最高的 T_2 缩短处理时间达到 31 倍.当然,该明显效果也与较多的谓词约束以及谓词约束通过概率较低有较大的影响.



(a) 一般情况下的最优构建策略 (b) 谓词判断的先后对识别效率的影响

Fig.5 Verification of best strategy on building the tree

图 5 最优树结构建策略验证

- 约束判断的先后顺序

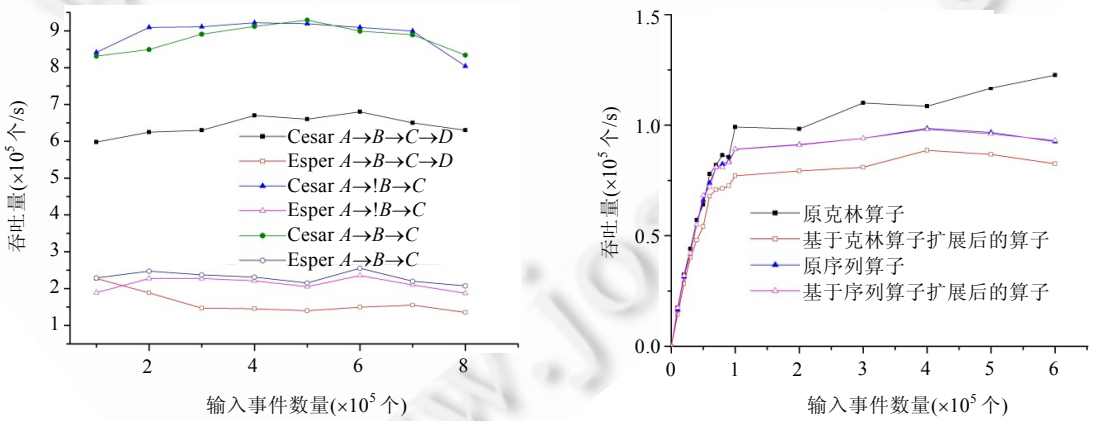
在 1 个算子上仍然有可能存在多个谓词约束.本实验验证如何排列 1 个算子中多个谓词约束判断的计算先后才能达到高效的识别速度.考虑到约束条件中,WE_1~WE_3 均为针对 AB 的约束条件采用 AB 事件各 1 000 000 个.为了避免其他操作耗时对系统带来的影响,仅针对 $PattenQuery=A\rightarrow B$ 以及约束条件,WE_1~WE_3.约束条件有 6 种放置方式,用 K123 表示依次判断 WE_1,WE_2 以及 WE_3,则这 6 种方式为 K123,K132,K231,K213,K312,K321. 6 种方式的识别效率如图 5(b)所示,K213 获得最少的处理时间,与模型分析的结论相同:当 1 个算子存在多个约束条件时,应当先判断通过概率低的约束条件.该问题仍然可以从数据量的角度进行解释,假设每次谓词约束判断所消耗的计算量相仿,则系统识别过程消耗的时间完全取决于数据量的多少.显然,只有先判断通过概率低的谓词约束,才能尽早地降低数据量,从而减少整体判断的次数,进而获得最小的时间消耗.

5.2 外部比较

由于 Cesar 的可扩展性已得到验证,所以本实验主要对 Cesar 的处理速度进行评价,即在数据大量到达的前提下,单位时间内能够处理的输入事件数量,也称为吞吐率.其计算方法为:输入数据总量/处理时间.其中,该处理时间已经是去除了数据读入以及类型转化的时间.本实验将优化后的 Cesar 系统与其他复杂事件流处理引擎进行对比,选取的是同样由 Java 语言开发的,目前被广泛使用的开源事件流引擎 Esper 作为对比基准,二者使用同样的输入数据以及查询语句.

5.2.1 多算子识别结构性能评价

在复杂事件处理引擎的使用过程中,更多的时候是使用多个约束相结合的查询条件.因此,本节对复杂的查询条件的识别效率进行验证,采用的输入查询条件选自另一篇使用复杂事件流引擎进行系统监控的工作[17]. Cesar 和 Esper 在相同输入情况下的识别速度如图 6(a)所示,Cesar 的识别效率均高于基于 NFA 实现的 Esper.但考虑到 Esper 因为是开源实现,其内部对内置函数的支持较为丰富,且需要对各种类型输入进行兼容,因此实现起来可能比 Cesar 更为臃肿.



(a) 多个时序约束时系统的吞吐率对比 (b) 扩展后算子与原有算子的吞吐率对比

Fig.6 Throughput comparisons between Esper with Cesar when face multiple time constraints

图 6 当存在多个时序约束时,Ceser 与 Esper 吞吐率的对比

但即使去除实现复杂度的影响,在一些特殊算子的识别过程中,Cesar 的实现效率仍然明显高于 Esper,例如加入了否约束“!”的查询.对比两条查询语句“$A\rightarrow B\rightarrow C$”以及“$A\rightarrow !B\rightarrow C$”的实现效率可以看出:在 Esper 的实现中,带有否算子的结构识别效率要略低于没有否算子的实现;而在 Cesar 中,二者的识别效率基本相同,且带有否算子的识别结构“$A\rightarrow !B\rightarrow C$”吞吐率还高于没带有否算子的查询“$A\rightarrow B\rightarrow C$”.因为 Esper 基于 NFA 实现,NFA 模型对否算子和克林算子的实现时间复杂度较高,因此带有此类约束的实现效率较低.而在树模型中,此类约束同

样被实现为一个具有 Q,F,Z 函数的算子,因此吞吐率基本上与其他查询约束类似.同时,由图 5(b)也可以看出:随着时序查询语句的增加,系统的吞吐率也在逐渐下降.且在时序约束增加的开始,吞吐率下降明显;而在时序约束已经具有一定数量后,再进一步增加时序约束则不会使吞吐率明显下降.其主要原因是:在时序约束较少时,符合查询的事件组合较多,而增加时序约束需要对所有符合查询条件的事件做进一步的处理,因此增加了处理时间,较大幅度地增加了吞吐率.而在已经具有一定数量的时序约束之后,符合查询条件的数据已经很少,增加约束也只需对较少的数据做进一步的筛选,因此仅增加极少的数据处理时间,对吞吐率影响减少.

5.2.2 附带自扩展算子的识别结构性能评价

本节对用户自行扩展的算子识别效率进行评价,主要采用第 3.1.4 节提出的两个时序约束进行了扩展,并将其与原有算子的识别速度进行比较.例 1 中,在原有的克林算子上进行了扩展,更改其匹配判定函数 Q 以及符合计算函数 F,因此在实验过程中主要用此算子的识别效率与原有的克林算子进行比较,结果如图 6(b)所示:随着数据量的增加,新扩展的算子吞吐率明显略低于原有的克林算子,但二者吞吐率随输入数据变化的趋势基本保持一致.其主要因是,新算子修正了原有克林算子的 Q 函数以及 F 函数,与其原有的两函数相比,其计算量增大,因此每个到达的事件都会比原有克林算子进行更多的计算,导致吞吐率下降是正常的,但因为二者处理方式基本相同,因此二者波动变化类似.

图 6(b)同时也对比了第 3.1.4 节例 2 扩展的序列算子以及原有序列算子,因为扩展的序列算子仅修改了复合计算函数 F 为 $F_e=a.starttime$,这种更改仅仅改变了生成新事件的结束时间,并没有增加任何计算量,因此可以看到:新扩展的算子与原有算子的计算量应该是完全相同的,二者吞吐率完全相同.由此可见:使用扩展的算子虽然有可能在一定程度上降低识别速度,但其识别效率还是由其定义的 Q,F,Z 这 3 函数的计算量决定.并不会因为增加扩展算子而显著影响整个识别结构的效率.

6 结论及下一步工作

本文通过形式化定义事件以及事件流之间的复合关系,明确提出了算子的概念,并将复杂事件处理中的时序约束、谓词约束映射到算子相应的组件,将算子作为事件复合的基本单元,使算子能够在支持现有语义的同时,可以完善地支持新增语义.本工作通过算子的方式构建树结构事件识别模型,并对该模型的识别效率进行了评估和优化,提出了一套定量化的优化方案.最后,在性能评测中验证了评估模型中各条优化结论,与当前使用最广泛的开源事件流引擎 Esper 进行了对比.在下一步工作中,将进一步完善事件模型,在多种事件筛选策略、乱序事件处理等方面继续进行研究.

References:

[1] Eugster PTH, Felber PA, Guerraoui R, Kermarrec AMG. The many faces of publish/subscribe. ACM Computing Surveys (CSUR), 2003,35(2):114−131. [doi: 10.1145/857076.857078]

[2] Mansouri-Samani M, Sloman M. GEM: A generalized event monitoring language for distributed systems. Distributed Systems Engineering, 1997,4(2):96−108. [doi: 10.1088/0967-1846/4/2/004]

[3] Gruber RE, Krishnamurthy B, Panagos E. The architecture of the READY event notification service. In: Proc. of the 19th IEEE Int'l Conf. on Distributed Computing Systems Workshops on Electronic Commerce and Web-Based Applications/Middleware. IEEE, 1999. [doi: 10.1109/ECMDD.1999.776423]

[4] Hong MS, Demers A, Gehrke J, Koch C, Riedewald M, White W. Massively multi-query join processing in publish/subscribe systems. In: Proc. of the SIGMOD 2007. 2007. [doi: 10.1145/1247480.1247564]

[5] Hu JF, Jin BH, Zhuo W, Chen HB, Zhang LF. Spatial event detection and optimization. Ruan Jian Xue Bao/Journal of Software, 2011,22:147−156 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/11035.htm

[6] Fu Y, Zhou MQ, Wang XS, Luan H. On-Line event detection from Web news stream. Ruan Jian Xue Bao/Journal of Software, 2010,21:363−372 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/10037.htm

[7] Balis B, Kowalewski B, Bubak M. Real-Time grid monitoring based on complex event processing. Future Generation Computer Systems, 2011,27(8):1103−1112. [doi: 10.1016/j.future.2011.04.005]

[8] Demerset A, Gehrke J, Panda B, Riedewald M, Sharma V, White W. Cayuga: A general purpose event monitoring system. In: Proc. of the CIDR 2007. 2007. 412−422. [doi: 10.1145/1247480.1247620]

[9] Dayal U, Blaustein B, Buchmann A, Chakravarthy U, Hsu M, Ledin R, McCarthy D, Rosenthal A, Sarin S, Carey MJ, Livny M, Jauhari R. The hipac project: Combining activedatabases and timing constraints. SIGMOD RECORD, 1988,17(1):51−70. [doi: 10.1145/44203.44208]

[10] Abbadi D, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S. Aurora: A new model and architecture for data stream management. In: Proc. of the Brown Computer Science (CS-02-10). 2002. [doi: 10.1007/s00778-003-0095-z]

[11] Gyllstrom D, Wu E, Chae HJ, Diao YL, Stahlberg P, Anderson G. SASE: Complex event processing over streams. arXiv preprint cs/0612128, 2006.

[12] Diao YL, Immerman N, Gyllstrom D. SASE+: An agile language for kleene closure over event streams. Technical Report, Department of Computer Science, University of Massachusetts Amherst, 2007.

[13] Agrawal J, Diao YL, Gyllstrom D, Immerman N. Efficient pattern matching over event streams. In: Proc. of the SIGMOD 2008. 2008. [doi: 10.1145/1376616.1376634]

[14] Wu E, Diao YL, Rizvi S. High-Performance complex event processing over streams. In: Proc. of the SIGMOD 2006. 2006. [doi: 10.1145/1142473.1142520]

[15] Chandramouli B, Goldstein J. High-Performance dynamic pattern matching over disordered streams. Proc. of the VLDB Endowment, 2010,3(1-2):220−231. [doi: 10.14778/1920841.1920873]

[16] Yuan M, Madden S. ZStream: A cost-based query processor for adaptively detecting composite events. In: Proc. of the SIGMOD 2009. 2009. [doi: 10.1145/1559845.1559867]

[17] Cheng SJ, Wang YJ, Meng Y, Cheng ZD, Luan ZZ, Qian DP. PMTree: An efficient pattern matching method for event stream processing. Journal of Computer Research and Development, 2012,49(11):2481−2493 (in Chinese with English abstract).

[18] Mei Y, Madden S. Zstream: A cost-based query processor for adaptively detecting composite events. In: Proc. of the 2009 ACM SIGMOD Int'l Conf. on Management of Data. ACM Press, 2009. [doi: 10.1145/1559845.1559867]

[19] Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Reiss F, Shah MA. TelegraphCQ: Continuous dataflow processing. In: Proc. of the ACM SIGMOD Conf. 2003. [doi: 10.1145/872757.872857]

附中文参考文献:

[5] 胡佳锋,金蓓弘,禚伟,陈海彪,张利锋.空间事件的检测及优化策略.软件学报,2011,22:147−156 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/11035.htm

[6] 付艳,周明全,王学松,栾华.面向互联网新闻的在线事件检测.软件学报,2010,21:363−372 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/10037.htm

[17] 程苏珺,王永剑,孟由,程振东,栾钟治,钱德沛.PMTree:一种高效的事件流模式匹配方法.计算机研究与发展,2012,49(11):2481−2493.

孟由(1986－),男,河北保定人,博士生,主要研究领域为监控,故障管理,复杂事件处理.
E-mail: mengyou0304@126.com

谢明(1978－),男,博士,工程师,主要研究领域为分布式存储.
E-mail: reganxie@tencent.com

栾钟治(1971－),男,博士,副教授,CCF 高级会员,主要研究领域为分布式计算,高性能计算,服务计算.
E-mail: rick710055@263.net

钱德沛(1952－),男,教授,博士生导师,CCF 会士,主要研究领域为计算机系统结构,高性能计算,分布式计算.
E-mail: depeiq@buaa.edu.cn