在线-离线数据流上复杂事件检测

彭商濂[1)] 李战怀[1)] 陈 群[1)] 李 强[2)]

[1)](西北工业大学计算机学院 西安 710129)

[2)](西北工业大学软件与微电子学院 西安 710072)

摘 要 随着数据采集和处理技术的发展,在物联网对象跟踪、网络监控、金融预测、电信消费模式等领域中进行事件检测显得越发重要.事件检测在一次扫描数据流的假设下完成,数据流在被处理完后丢弃.事实上,很多应用场景中,历史数据流因含有丰富的信息而不能简单丢弃,且一些事件检测查询需要同时在实时和历史数据流上进行.鉴于已有复杂事件检测很少考虑同时在实时-历史数据流上进行模式匹配,作者研究了在线-离线数据流上复杂事件检测的关键问题.主要工作如下:(1)针对滑动窗口内产生的大量模式匹配中间结果,提出利用时态关系和时空关系管理中间结果的方法 TPM 和 STPM. STPM 以中间结果的时态和状态信息为权值对中间结果进行管理,将最近的、最有可能更新状态的中间结果置于内存,极大地减少了中间结果的读取操作代价.(2)给出了基于选择度的在线-离线复杂事件检测优化算法;(3)给出了算法的复杂性分析和代价模型;(4)在基于时空关系的中间结果管理模型下,在一个在线-离线复杂事件检测原型系统中进行实验,对多个参数(子窗口大小,选择度,匹配率,命中率)进行了算法对比分析.实验结果充分验证了所提出的算法的可行性和高效性.

关键词 物联网;复杂事件检测;数据流;非确定有限状态自动机;RFID;无线传感器网络

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2012.00540

Complex Event Processing over Live Archived Data Streams

PENG Shang-Lian[1)] LI Zhan-Huai[1)] CHEN Qun[1)] LI Qiang[2)]

[1)](*School of Computer Science, Northwestern Polytechnical University, Xi'an* 710129)

[2)](*School of Software and Microelectronics, Northwestern Polytechnical University, Xi'an* 710072)

Abstract With the development of data collection and data processing techniques, event detection has become increasingly vital in application areas such as object-tracking in IOT, network monitoring, financial prediction, and telecommunication consumption mode detection, etc. Event processing is supposed to be completed in one-pass of the data streams which are discarded after pattern matching. Actually, historical streams maintain plentiful information which cannot be simply discarded in many scenarios and some event detection queries are always subscribed over both live and archived (historical) streams. Due to the lackness of event processing over live and archived event streams, this paper addresses key issues of live- archived stream complex event processing. Main works are as follows:(1)Due to large numbers of partial matches generated in a sliding window, partial matches management methods named TPM and STPM are proposed. With STPM, spatial and temporal information are kept into partial matches and the most recent and possible updated partial matches are resided in main memory which can reduce pattern match miss ratio and greatly alleviate external partial match loading I/O cost. (2) Optimization of complex event processing algorithm over live-archived streams based on events selectivity is pro-

收稿日期:2011-08-31;最终修改稿收到日期:2011-12-27.本课题得到国家自然科学基金(60803043,60873196,61033007)、国家"八六三"高技术研究发展计划项目基金(2009AA01A404)资助.**彭商濂**,男,1980年生,博士研究生,主要研究方向为RFID数据管理、复杂事件检测等. E-mail: pengshanglian@mail.nwpu.edu.cn. **李战怀**,男,1961年生,教授,博士生导师,主要研究领域为数据库理论、海量数据存储等. E-mail: lizhh@nwpu.edu.cn. **陈 群**,男,1976年生,教授,博士生导师,主要研究领域为XML数据管理、复杂事件检测、数据质量、云计算等. **李 强**,男,1986年生,硕士研究生,主要研究方向为复杂事件检测、RFID数据管理.

posed. (3)Formal cost model of related methods are presented. (4) Based on the proposed partial matches management methods , extensive performance comparison experiments in a prototype CEP system are evaluated(experimental parameters include subwindow size, selectivity, match ratio, hit ratio, etc). Experimental analysis verifies soundness and effectiveness of the proposed methods.

Keywords　Internet of Things; complex event processing; data stream; nondeterministic finite automation (NFA); RFID; wireless senser networks

1　引　言

随着逐渐增多的应用领域(如金融业、网络监控、基于 RFID 的供应链管理、医疗监控、物联网应用等)需要处理大量分布、高速的数据流,复杂事件检测(Complex Event Processing,CEP)变得越发重要[1-2].用户通过复杂事件定义语言描述需要检测的模式(复杂事件),复杂事件检测引擎将用户定义的模式解析为自动机或树模型,然后在事件流上进行事件检测.在当前的复杂事件处理系统中,一般假设事件流以一次扫描的方式被扫描,且处理完后就被抛弃,无需进行归档.

事实上,一次扫描数据流并没有充分发掘数据流的价值,用户将查询定义在数据流上也只能描述一部分现实世界的查询需求.正如下面列举的实例显示,数据流上的复杂事件查询不但需要处理实时数据流,同时也需要访问历史数据流.

示例 1:汽车行驶时间实时计算(如超速监控).高速公路上,汽车上安装有主动 RFID 电子标签用于身份识别和跟踪;高速公路收费点或沿路的监控点都安装了 RFID 阅读器.当汽车经过高速公路收费点时(或途中的监控点时),这些阅读器可以跟踪单辆汽车,并记录其速度信息.为了计算当前点和上一个点的行驶时间,需要从实时数据流上取出事件记录,然后查询历史数据流寻找上一个点的时间信息,判断是否超速行驶.该查询如图 1 所示,是用文献[3]中的基于 SQL 的查询语言描述的.

示例 2:股票趋势预测.股票的趋势预测信息可以辅助投资者进行股票的买进或抛出.投资者可以在股票流上定义如下的复杂事件查询:当在一支股票的实时数据流上检测到一个下降模式("\")时,在历史股票流上检测一个勾模式("√"),该查询如图 2 所示,也用文献[3]中查询语言描述.

这些示例表明,复杂事件检测需要同时集成在

```
Query1:
SELECT CarID_l, Timestamp_l, CarID_a, Timestamp_a
FROM LiveCar MATCH_RECOGNIZE(
PARTITION BY CHECKPOINT_RID
MEASURES CarID AS CarID_l,
     Timestamp AS Timestamp_l
 ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
), ArchivedCar MATCH_RECOGNIZE(
PARTITION BY CarID
MEASURES CarID AS CarID_a,
     Timestamp AS Timestamp_a
ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
)
WHERE CarID_l=CarID_a
AND Timestamp_l-Timestamp_a<30 min;
```

图 1　基于 RFID 的高速公路行驶时间监控事件查询

线数据流与离线数据流的处理.在线-离线复杂事件处理在实际应用中需面对下列挑战:(1)海量高速的流数据.应用场景很容易产生 GB 级到 TB 级的流数据(如在基于 RFID 的电子超市),如何有效地存储这些数据是在线-离线复杂事件处理的重要问题;(2)复杂事件中间结果管理.在很多复杂事件检测应用中,用户定义一个很大的时间窗口 W.随着复杂事件检测的推进,在 W 中的复杂事件的中间结果数量将变得非常巨大.同时在历史数据流上复杂事件检测的中间结果也将随之增大.在复杂事件处理节点内存资源有限的情况下,如何对这些生命周期在 W 内的中间结果进行有效的管理对复杂事件检测的响应时间具有重要影响.(3)数据流中事件分布的不同使得事件检测在实时数据流和历史数据流上的实例数目也不同,导致历史流的访问请求次数的不同和历史流上模式的重复计算,利用数据流中事件类型的选择度和不同实例的历史流访问区间交叉等特性对在线-离线事件检测顺序的进行调度可以有效减少复杂事件的响应时间.

当前的复杂事件检测主要针对实时数据流,很少考虑再次访问历史流.与本文工作最为接近的是 Moirae[4] 和 Dejavu[5-6].Moirae 是一个集成了流数据处理和数据库技术的数据流处理引擎(Stream Processing Engine,SPE).Moirae 支持 4 类查询:事

```
Query2:
SELECT min_tstamp_l, symbol_l, min_price_l,
initial_price_a, min_price_a, max_price_a
FROM LiveStock MATCH_RECOGNIZE(
PARTITION BY Symbol
MEASURES A.Symbol AS symbol_l,
         MIN(B.Tstamp) AS min_tstamp_l,
         MIN(B.Price) AS min_price_l
ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
INCREMENTAL MATCH
PATTERN SEQ(A; B+)
DEFINE
 B AS (B.Price<A.Price AND B.Price<=PREV(B.Price))
), ArchivedStock MATCH_RECOGNIZE(
PARTITION BY Symbol
MEASURES A.Symbol AS symbol_a,
         A.Price AS initial_price_a,
         MIN(B.Price) AS min_price_a,
         LAST(D.Price) AS max_price_a
ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
MAXIMAL MATCH
PATTERN SEQ(A; B+; C*; D+)
DEFINE
  B AS (B.Price<A.Price AND B.Price<=PREV(B.Price))
  C AS (C.Price>=PREV(C.Price) AND C.Price<=A.Price)
  D AS (D.Price>PREV(D.Price) AND D.Price>A.Price)
)
WHERE symbol_l = symbol_a;
```

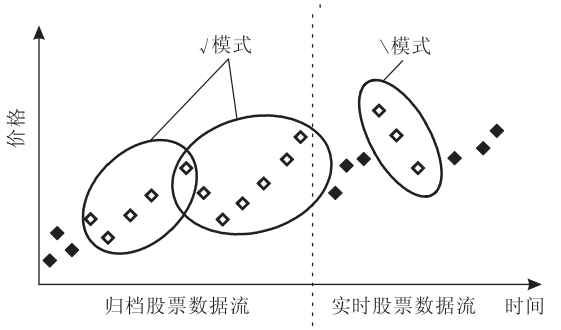

图 2 股票流上事件查询

件查询、标准混合查询、上下文查询、上下文混合查询. Moirae 设计的假设是:查询整个历史日志(数据)太慢以至于不能满足应用的响应要求,所以返回最相关的近似查询结果是合理的. 基于该假设,Moirae 提出了历史流分区存储、最近访问的事件物化、查询执行分解和历史查询调度等设计方法. 可以看出,这些方法都不是全新的,且已经应用到数据库的相关领域. 由于复杂事件检测要求的是准确的查询,所以 Moirae 的设计不能直接用于 CEP. Dejavu 是一个集成了在实时-历史上进行复杂事件检测的 SPE. Dejavu 通过扩展 SQL 语言以便支持复杂事件的定义(如增加模式匹配字句 MATCH_RECOGNIZE[3]),使该事件处理引擎可以定义复杂事件查询,同时通过扩展 MySQL 的读写数据 API,使得该引擎支持快速的数据流归档. 在 Dejavu 中,为避免历史数据流的重复计算,使用了结果缓存的数据结构将历史流上的模式缓存,以便为实时流上后续的复杂事件检测共享使用. 此外,通过利用实时数据流和历史数据流的选择度不同来调整这两个流的连接顺序以优化事件检测查询的执行. 在 Dejavu 中,作者主要考虑股票流上连续模式的在线离线复杂事件检测. 由于离散模式的子事件可能分布在数据流的任意位置,中间结果缓存及相应的优化算法都需要重新考虑. 作者研究了基于离散模式的在线-离线复杂事件检测,文献[7]是作者的初步工作,提出了基于离散模式的在线-离线复杂事件检测的系统结构和基本处理方法. 本文在文献[7]工作的基础上,增加了下列研究内容:(1)针对大滑动窗口内产生的大量模式匹配中间结果,在系统结构中增加了中间结果管理的模块,提出利用时态关系和时空关系管理中间结果的方法 TPM 和 STPM. STPM 以中间结果的时态和状态信息为权值对中间结果进行管理,将最近的最有可能更新状态的中间结果置于内存,极大地减少了中间结果的读取操作代价. (2)增加了基于选择度的在线-离线复杂事件检测优化算法;(3)给出了算法的复杂性分析和代价模型;(4)在基于时空关系的中间结果管理模型下,重新进行了实验设计,进行了相关算法的对比分析.

本文第 2 节介绍相关工作;第 3 节介绍事件模型、问题定义和在线-离线事件检测架构;第 4 节介绍在线-离线 CEP 检测算法及其优化;第 5 节描述中间结果管理策略;第 6 节为实验结果分析;第 7 节总结全文和下一步工作展望.

2 相关工作

复杂事件检测是一种从简单事件集合中关联出更具有语义的复杂事件的数据分析技术,最早在基于 trigger 和 ECA 规则的主动数据库[8-9]中应用,随后在基于关键字或简单谓词的 Publish/Subscribe 系统中应用[10]. 随着流数据处理系统的推广,结合窗口概念的复杂事件检测也被集成到数据流管理系统(Data Stream Management Systems, DSMS)中. 典型的系统有 Aurora[11]、TelegraphCQ[12]和 HiFi[13]等. 这些系统主要针对与数据流的连接(join)、聚集(aggregation)和一些统计(statistics)操作,不适用于定义复杂事件检测的查询.

SASE[1,14-15]、ZStream[16]和 Cayuga[2]是流数据上复杂事件检测的 3 个比较著名的原型系统. 它们各自有事件定义语言(基于类 SQL 或声明型定义语言)和查询模型(如自动机模型、树模型)及相应的优化策略,但缺少集成历史流访问的复杂事件定义和检测的支持.

文献[8,17]研究了在传统数据库上进行模式匹配的相关问题[3]，包括模式定义语言、算子语义和查询模型.这些工作缺乏对流数据上事件检测的考虑，也不适用于流数据上更为复杂的事件查询.

Moirae[4]和 Dejavu[5-6]是两个与本文工作最为接近的实时-历史数据流查询系统.Moirae 主要研究在不可能遍历整个历史流情况下如何为查询返回最相似的结果的问题，精确的查询结果可以通过逐步求精的方式获得.Moirae 通过将已有数据流、数据库和存储等方面的技术集成到一个框架中，支持多种类型的查询.本文工作与 Moirae 的不同之处在于，本文考虑的复杂事件查询要求精确结果且复杂事件有可能分布在较长的时间区间内，考虑如何快速、完整检测数据流上的复杂事件，同时考虑在复杂事件中间结果较多而引起系统负载较大时如何降载的问题.Dejavu 是一个集成实时-历史数据流处理的复杂事件系统.Dejavu 通过声明性的模式匹配语言定义复杂事件.在 Dejavu 中，数据的存储架构基于开源的数据库系统 MySQL.通过在系统中集成 MySQL 的可插入式 API，Dejavu 引入 DStream 和 DArchive 这两类存储结构，并在 DStream 和 DArchive 之间设计了一个最近处理过的事件缓冲区(Recent Buffer)来平衡实时流和历史流归档操作，同时为历史流上的复杂事件检测提供数据源.在 Dejavu 中使用了结果缓存(Results Cache)数据结构存储在历史流上寻找到的模式，以便为后续的模式匹配使用.Dejavu 所处理的在线-离线复杂事件检测主要针对连续模式(如在股票数据流上寻找“勾”)，而 Dejavu 中的离散模式主要是实时处理.此外，Dejavu 中的中间结果缓存基于如下的观察：历史流上检测到的一个实例可能被实时流上的多个实例访问，所以把已经加载的历史流上的所有模式检测出来并缓存有利于减少重复计算和数据加载.但我们观察到，连续模式其实是更为一般的模式——离散模式的特例，离散模式并不严格要求子事件的顺序，即离散模式的中间结果(子事件)可能分散在实时流和历史流的任何可能位置，所以将历史流上的所有中间结果计算出来缓存在内存对离散模式的在线-离线复杂事件检测是不适用的.

Chandrasekaran[18]等研究了在 DSMS 中集成历史数据流查询的问题，主要工作包括存储管理器降载及查询结果精度之间的折中、索引插入和归档查询降载及各种查询在实时流和历史数据流上的无缝执行[19].研究通过位图索引对历史数据流进行存储和更新.他们的工作主要针对实时-历史数据流上的聚集、统计操作，不适用于复杂事件检测.

复杂事件处理的相关工作包括中间件[20-25]、RFID 数据流上的复杂事件检测[1,7,14,26-27]、乱序数据流上的复杂事件检测[28-31]、嵌入序列复杂事件检测[32]、基于不确定模型的事件检测等[33].其中文献[1,14,26]研究在实时数据流进行复杂事件检测，没有考虑实时-历史数据流集成的复杂事件检测.文献[7]是作者对在线-离线复杂事件检测的前期基础研究工作，但没有更为高效的中间结果管理方案和复杂事件检测优化算法.

3　事件模型与问题定义

3.1　事件模型

定义 1.　原始事件.原始事件是指不能再分解成子事件的事件.原始事件是由一些属性组成的元组(*tuple*)，记为 $e(TID, state, Timestamp, \langle attriList\rangle)$，其中 *state* 为事件类型标识，*TID* 为对象标识，*Timestamp* 为事件发生的时间戳，$\langle attriList\rangle$ 为事件的其它属性列表.本文中，事件类型用大写字母 E 表示，E 对应的实例用小写 e 表示.原始事件语义简单，实际应用中需要将原始事件转换为语义更为丰富的复杂事件.

定义 2.　复杂事件.将原始事件/复杂事件通过事件算子，如 SEQ(序列)、AND(与)、OR(或)、NOT(非)等，组合成的事件称为复杂事件.

定义 3.　连续模式.对于一个模式 P，如果构成该模式的子事件要求连续地出现在数据流中，称模式 P 为连续模式.

定义 4.　离散模式.对于一个模式 P，如果构成该模式的子事件可以出现在数据流中任何位置，则称模式 P 为离散模式.可见，离散模式是一种更为通用的模式，包含了连续模式.

定义 5.　在线-离线数据流上事件检测.如果复杂事件检测由在实时流上的模式和离线流(历史流)上的模式组成，则称为在线-离线数据流上事件检测.

3.2　复杂事件定义语言

复杂事件通过事件定义语言描述，目前常用的事件定义语言包括类 SQL 型语言[3]和声明型语言[1].本文对文献[3]的基于 SQL 的复杂事件定义语言进行了适当的扩充以便定义在线-离线复杂事件查询，事件定义语言的结构如下：

```
SELECT 〈selected fields list〉
FROM 〈streams or tables〉MATCH RECOGNIZE (
    [PARTITION BY]
    [ORDER BY 〈field name〉]
    [MEASURES 〈measure list〉]
    [ONE/All ROW PER MATCH]
    [AFTER MATCH SKIP TO NEXT ROW/
    PAST LAST ROW /…]
    PATTERN (pattern description)
    DEFINE 〈events constraints list〉
    [WINDOW 〈window specification〉]
)
```

其中 MATCH_RECOGNIZE 子句表示模式匹配的入口；PARTITION BY 将事件流按某属性进行分区；ORDER BY 将事件流按一定属性排序；MEASURES 子句将事件的一些属性重新命名；ONE/ALL ROW PER MATCH 和 AFTER MATCH SKIP TO NEXT 是模式匹配的事件选取策略；PATTERN 字句中定义要检测的模式；DEFINE 描述复杂事件的约束条件；WINDOW 子句定义了复杂事件的有效时间区间.

如图 3 所示，(a)表示的是 Query2 定义的检测在线-离线股票流上的连续模式，(b)中描述的是基于 RFID 的供应链监控中数据流上的离散模式.

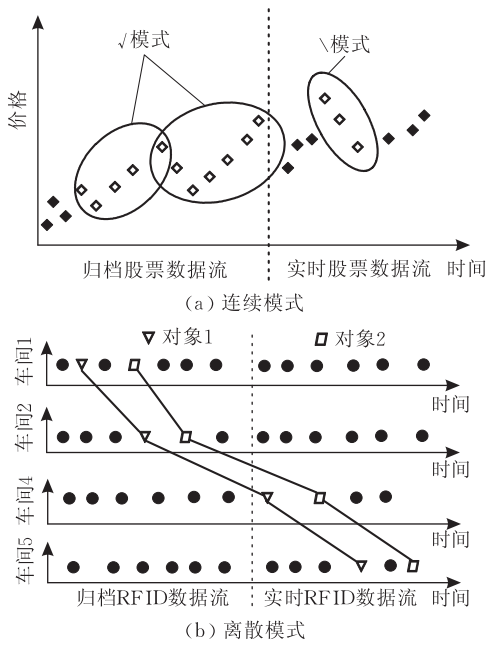

图 3 连续模式与离散模式示例

3.3 在线-离线复杂事件检测架构

在线-离线数据流上复杂事件检测系统结构如图 4 所示. 事件流通过网络接口进入事件流路由器，事件流路由器可以过滤一些不相关的数据流. 数据流以追加的方式存储于时实流缓冲区中，该缓冲区的数据一方面可以为实时流上的复杂事件检测提供源数据，一方面也可以为历史流上的复杂事件检测提供源数据，且数据流的归档可以通过该缓冲区批量完成. 数据流原始数据存储在关系数据库或文件系统中，并可以按事件的 ID 或时间区间进行反复读取. 历史数据流的并发读写通过信号量来控制.

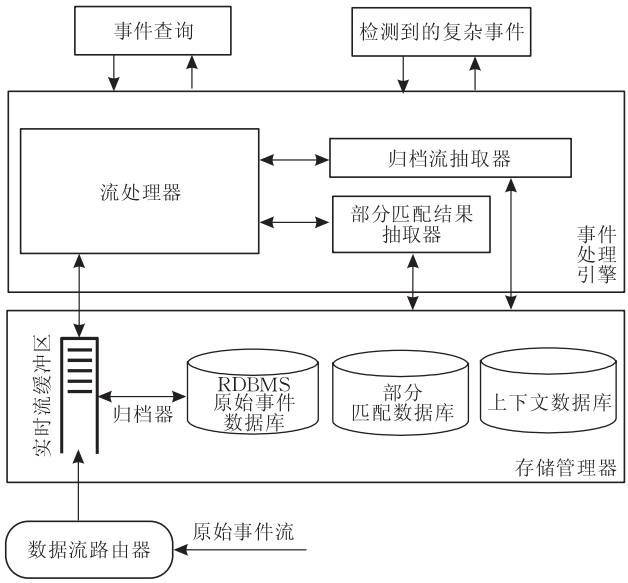

图 4 在线-离线复杂事件检测系统架构

在事件检测过程中，没有达到完全匹配且没有超出滑动窗口的部分模式匹配在系统负载较大的时候被归档到部分匹配数据库中. 数据流处理引擎接收用户定义的复杂事件查询，并将事件查询解析为数据流处理器里对应的查询执行模型(如自动机、树或 Petri 网)，数据流处理器分别从实时流、历史流(通过归档事件抽取器)和部分模式匹配数据库(通过部分匹配抽取器)读取相应的数据进行在线-离线复杂事件模式匹配. 归档事件抽取器中分别以事件的 ID(Hash 索引)和事件归档的时间(B^+ 树索引)建立二维的索引结构，支持单个历史事件和某个时间区间上的历史事件的访问. 部分匹配抽取器中以中间结构的 ID 构建 Hash 索引，以支持快速的中间结果的读取、更新及删除操作.

4 在线-离线复杂事件检测

在线-离线流上复杂事件检测需要在实时数据流和历史数据流上进行，由于实时流和历史流处理模式的不同而导致复杂事件检测响应时间的不同. 本节介绍两种在线-离线复杂事件检测算法并分析它们的代价模型，代价模型的相关标识如表 1 所示.

表 1　代价模型参数表

标记	描述
W_L	实时流上定义的滑动窗口
$Inst$	实例，表示模式的具体对象
R_A	历史(离线)流的访问区间
P_L	实时流上定义的模式
P_A	历史流上定义的模式
PM	实时流上事件检测的中间结果
$Cost_L$	处理实时流上每个事件的平均代价
$Cost_A$	处理离线流上每个事件的平均代价
N_{Inst_L}	实时流上复杂事件实例的个数
N_{Inst_A}	离线流上复杂事件实例的个数
$Cost_{WDB}$	离线数据流的平均代价
$Cost_{WPM}$	离线中间结果的平均代价
$Cost_{RDB}$	读取数据库中归档流的平均代价
$Cost_{P_L \bowtie P_A}$	连接 P_L 和 P_A 的平均代价
N_P	检测出的复杂事件数目

4.1　主动在线-离线事件检测算法

在线-离线数据流上复杂事件检测的一种算法为主动算法(Active Live-Archived Event Detection Algorithm，ALAA). ALAA 的算法描述如算法 1 所示.

算法 1.　主动在线-离线事件检测算法.

输入：P_L，P_A，W_L，R_A

输出：检测到的复杂事件 CE

CoBegin

1. 初始化实例链表 PM_L、PM_L、$Inst_L$ 和 $Inst_A$ 为 Null；
2. 对于实时流上滑动窗口 W_L 内的事件
3.　　进行复杂事件 P_L 的检测；
4.　　将达到最大状态的 P_L 实例放入 $Inst_L$ 中；
5.　　将未达到最大状态的 P_L 实例放入中间结果集合 PM_L 中；
6. 计算最大的历史流访问区间 $R_{\max}$；
7. 对于 $R_{\max}$ 内的事件
8.　　进行复杂事件 P_A 的检测；
9.　　将达到最大状态的 P_A 实例放入 $Inst_A$ 中；
10.　　将未达到最大状态的 P_A 实例放入中间结果集合 PM_A 中；
11.　将 $Inst_L$ 中的实例与 $Inst_A$ 中的实例在 $EventID$ 属性上进行连接操作；
12.　　$CE = Inst_L \bowtie_{EventID \wedge R_{A_i} \wedge P} Inst_A$；
13.　归档历史流数据，处理下一个滑动窗口 W_L；

CoEnd

ALAA 算法对于实时数据流上 P_L 和历史数据流上 P_A 的检测是并行进行的，并通过消息进行通信. 当实时流上检测完一个滑动窗口 W_L 长度的 P_L 后，SPE 通知复杂事件连接器将 $Inst_L$ 中的实例和已检测出的 $Inst_A$ 实例进行事件标识(EventID)或其它属性(如 R_{A_i}，P)的连接操作，生成最后可以输出的复杂事件. ALAA 算法示意如图 5 所示.

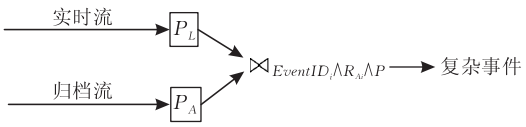

图 5　ALAA 算法执行示意图

ALAA 算法的代价模型. ALAA 算法的代价模型主要由下面几部分组成：(1) 在实时流上进行 P_L 事件检测的代价；(2) 在历史流上进行 P_A 事件检测的代价，这部分代价包括将实时数据流归档的代价和将数据流从外存读取到内存的代价；(3) 将 $Inst_A$ 实例和 $Inst_A$ 实例进行连接操作的代价：

$$Cost_{ALAA} = |W_L| \times Cost_L + |R_{\max}| \times Cost_{RDB} + |R_{\max}| \times Cost_A + Cost_{P_L \bowtie P_A} \times N_{Inst_L} \times N_{Inst_A} \quad (1)$$

从代价模型可以看出，实时流滑动窗口 W_L 的长度和数据流的流速决定了复杂事件的数目，数据流流速越大，窗口长度越长，则实时流上复杂事件检测所需要的计算资源就更多，计算代价也增大. 此外，由于实时数据流是缓冲在一个缓冲区结构中，该缓冲区在为复杂事件检测提供数据源，同时已经处理的数据流也将以批量方式插入数据库，所以缓冲区的大小决定了批量插入的规模，这里需注意的是：缓冲区越大，给 SPE 所能分配的存储资源就越小，同时，批量插入数据库的代价 $Cost_{WDB}$ 也将增加. 最后，实时和历史流上的复杂事件实例连接操作代价 $Cost_{P_L \bowtie P_A}$ 为在所有已检测出来的 $Inst_A$ 和 $Inst_L$ 上进行连接操作的代价.

ALAA 算法的不足在于它需要检测历史流中所有存在的 P_A 模式实例，无论该实例是否参加后续的模式匹配. 可以看出，ALAA 算法是一种直接但比较盲目的算法，它没有利用到数据流中各个事件类型的选择度特性来减少复杂事件检测的计算代价. 因此，我们介绍另外一种在线-离线复杂事件检测算法.

4.2　被动在线-离线事件检测算法

在线-离线数据流上复杂事件检测的另外一种算法为被动算法(Lazy Live-Archived Event Detection Algorithm，LLAA). 与主动算法不同，被动算法只有在其中一个数据流上检测出达到最大状态的实例后才触发另外一个数据流上的复杂事件检测. LLAA 算法的执行是一个顺序交替的过程，如算法 2 所示.

算法 2.　被动在线-离线事件检测算法.

输入：P_L，P_A，W_L，R_A

输出：检测到的复杂事件 CE

1. 对于实时流 W_L 内的每个事件 e_i

2. 检测是否存在对应的 P_L 实例 $InstL_i$
3. 如存在,但将 e_i 的状态更新至 $InstL_i$ 后没达到最大状态处理 W_L 中的下一事件;
4. 如存在,且将 e_i 的状态更新至 $InstL_i$ 后达到最大状态
5. 计算 $InstL_i$ 的历史流访问区间 R_{Ai};
6. 将 R_{Ai} 对应的数据流读取到内存,检测历史流 R_{Ai} 上的 P_A 实例并放入 $InstA$ 中;
7. 在事件 ID 属性上对实例 $InstL_i$ 和 $InstA_j$ 进行连接操作
$CE_i = InstL_i \bowtie_{EventID_i \wedge R_{Ai} \wedge P} InstA_j$;
8. 输出 CE_i;
9. 处理 W_L 中的下一事件 e.

LLAA 算法的执行示意图如图 6 所示. 算法的执行可以从实时流或历史流上的事件检测的任意一边开始. 算法 2 中是从实时流开始. 当实时流上检测到一个 P_L 实例后,就触发一次归档流上的 P_A 的复杂事件检测. 为了减少数据流的重复扫描和计算,LLAA 算法一次把 R_{Ai} 对应的历史流加载进内存并将 R_{Ai} 上的所有的 P_A 实例都检测出来,此操作是基于对实际应用场景中的观察:相似的模式总有聚集出现的现象,如在一个传送带上,监控物品经过某几个点的路径总是聚集在数据流的一个事件区间上出现. 当 R_{Ai} 上的所有的 P_A 实例都检测出来后,将实时流上的 $InstL_i$ 实例与它们进行一些约束关系(如 $EventID_i$,R_A 等)的连接操作,生成可输出的复杂事件,然后进行下一个实时流事件的处理.

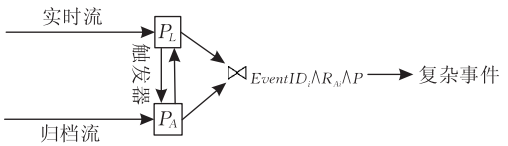

图 6 LLAA 算法执行示意图

LLAA 算法的代价模型. LLAA 算法的代价模型包括如下几部分:(1)处理实时流 W_L 中的每个事件的代价;(2)读取 $InstL_i$ 对应的历史流到内存的代价;(3)在 $InstL_i$ 对应的历史流上进行 P_A 检测的代价;(4) $InstL$ 与 $InstA$ 进行连接操作的代价:

$$Cost_{\mathrm{LLAA}} = |W_L| \times Cost_L + N_{Inst_L} \times Cost_{RDB} + \sum_{i=1}^{N_{InstL}} Cost_A \times |R_{Ai}| + Cost_{P_L \bowtie P_A} \times N_P \quad (2)$$

在该代价模型中,$N_{Inst_L} \times Cost_{RDB}$ 为所有达到最大状态的实时流上的实例触发的历史流访问的代价,$\sum_{i=1}^{N_{InstL}} Cost_A \times |R_{Ai}|$ 是计算 $InstL_i$ 对应的历史流 R_{Ai} 上的复杂事件的代价,$Cost_{P_L \bowtie P_A} \times N_P$ 为实时流和归档流上达到最大状态的实例连接操作的代价. LLAA 算法的复杂度为 $O(|W_L| \times |R_A|)$.

LLAA 算法是一种基于实时流上单实例驱动的算法,即只有实时(历史)数据流上存在一个最大状态的实例时,才触发另外一个数据流上的复杂事件检测. 实际上,基于单个实例的 LLAA 事件检测的方法没有利用到实时流上临近的实例访问历史流区间重叠,导致多次外存数据的请求和重复的计算,因此在下节我们介绍如何利用数据流的选择度和模式的聚集特点优化 LLAA 算法. 需要注意的是,本文提出的 LLAA 算法和文献[6]中的 Lazy Pattern Processing 都是基于触发式的查询处理方法,即先在实时流上循环处理,待检测到一个实时流的实例后,再作历史流的处理. 在文献[6]中,由于针对的是连续模式的查询,第一次历史流访问后的所有实例按时间顺序存储,并在新的实时流实例被检测到时删除不可能存在匹配的历史流上的实例. 由于本文处理的是离散模式,触发历史流访问的实例可能存在于实时流的任何位置,因此需要缓存很多实时流的实例,且每个实时流实例访问历史流的时机不一样,因此历史流上计算出来的实例也不能简单的删除. 所以,本文的 LLAA 虽在结构上和文献[6]相似,但因为处理的问题的差异,所需要的存储和计算代价是不一样的. 另外,本文使用 B^+ 树结构存储原始事件和中间结果,可以更为快捷地进行状态更新、批量数据库插入、批量过期事件的删除等操作.

4.3 优化 LLAA 算法

定义 6. 选择度. 给定一个模式 P,事件流 S,P 选择度 Sel_P 定义为

$$Sel_P = \frac{N_P}{|S|} \quad (3)$$

即数据流上匹配模式 P 的实例的数目占事件流大小的比例. 可见 Sel_P 越大,则表明事件流 S 中匹配 P 的实例数目越多; Sel_P 越小,S 中匹配 P 的实例数目越小. Sel_P 常来确定最佳的查询执行计划(如连接操作中,用较小 Sel_P 的对象集合连接较大 Sel_P 集合可大大减少连接操作的代价).

(1)基于 Sel_P 的 LLAA 算法优化

在 LLAA 算法中,Sel_P 可以用来选择实时流和历史流的连接顺序,如 $Sel_{P_L} < Sel_{P_A}$,则 $Inst_L \bowtie Inst_A$ 可以减少历史流的加载和重复计算;如果 $Sel_{P_A} < Sel_{P_L}$,$Inst_A \bowtie Inst_L$ 可以减少实时流上不会产生最终输出的模式匹配的计算. 由于历史流存储在静态的文件或数据库上,其选择度可以通过数据库管理

系统的工具得到；对于实时流，选择度可以通过应用场景的先验知识或进行统计抽样等技术计算. 基于选择度的实时流-历史流连接顺序选择如图 7 所示.

图 7 中，实时流的滑动窗口 W_L 中有 3 个 P_L 实例，而 W_L 对应的历史流访问区间 R_A 中有 6 个 P_A 实例，$Sel_{P_L} < Sel_{P_A}$，所以在当前滑动窗口中选择实时流连接历史流以减少历史流的加载与计算.

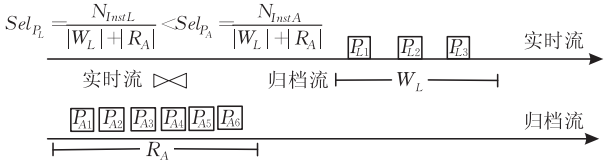

图 7　LLAA 算法数据流连接顺序选择示意图

(2) 基于子窗口的 LLAA 算法优化

虽然通过选择度可以减少部分历史流的加载与计算，但当 W_L 存在多个 P_L 实例，且每个实例都对应自己的归档流访问请求，如果采用检测到一个 P_L 实例触发一次历史流的访问与计算，将会导致多次数据流的请求，且没有利用到临近 P_L 实例访问的历史流具有交叉的性质. 如图 8 所示，实时流上存在 7 个 P_L 实例 $P_{L1}\sim P_{L7}$，如果按照算法 2 的复杂事件检测方法，需要进行 7 次历史流访问，虽然可以通过结果缓存共享一次加载的历史流计算结果，但是两个历史流未交叉的部分还需要加载和计算. 如图 8 中的历史流访问区间 R_{A2}-R_{A3}，R_{A4}-R_{A5}-R_{A6}，它们之间都存在交叉，此时如果将 R_{A2}-R_{A3}，R_{A4}-R_{A5}-R_{A6} 分别作为一个处理单元进行加载和计算，可以减少数据库的访问请求操作和重复计算.

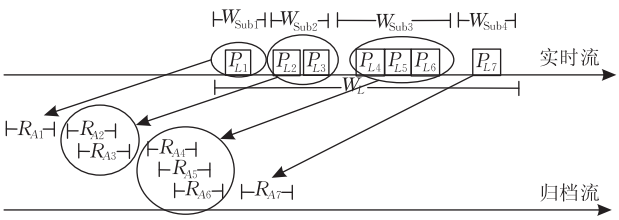

图 8　基于子窗口的 LLAA 算法执行示意图

因此，我们利用实例间的聚集特性将滑动窗口划分为多个子窗口，以子窗口为单位进行在线-离线复杂事件检测.

如何选择合适大小的子窗口与复杂事件输出的延迟直接相关：如果子窗口选择较小，则可能导致多次的历史流访问请求和计算，增加响应时间；如果子窗口选择较大，则可能导致较长的复杂事件输出的延迟.

引理 1.　给定一个在线-离线复杂事件查询，一个用户定义的容忍延迟时间 T_L，对于滑动窗口 W 内达到最大状态的复杂事件实例序列 I_1，I_2，…，I_n，实例 I_i，I_{i+1}，…，I_{i+j} 被划分到第 i 个子窗口 W_{Sub_i} 当且仅当

$$|\min\{I_k.e_{\text{start}}.time\}-I_{i+j}.e_{\text{end}}.time|\leq T_L$$

且　$|\min\{I_k.e_{\text{start}}.time\}-I_{i+j+1}.e_{\text{end}}.time|\geq T_L$.

其中，$i\leq k\leq(i+j)$，$I_i.e_{\text{start}}.time$ 表示第 i 个达到最大状态的实例的最后一个子事件的时间戳，$I_i.e_{\text{end}}.time$ 表示第 i 个达到最大状态的实例的最后一个子事件的时间戳.

引理 1 表明 W_{Sub_i} 的长度为该窗口中具有最大 $e_{\text{end}}.time$ 的实例与具有最小 $e_{\text{start}}.time$ 的实例的时间差.

基于子窗口的 LLAA 如算法 3 所示.

算法 3.　基于子窗口的 LLAA 算法(SLLAA)

输入：P_L，P_A，W_L，R_A，T_L

输出：检测到的复杂事件 CE

1. 对W_L中的事件进行复杂事件 P_L 的检测；
　　将达到最大状态的 P_L 实例放入 $Inst_{Li}$ 中，未达到最大状态的置于中间结果缓冲区；
2. 根据引理 1 计算子窗口 W_{Sub} 的大小；
3. 计算 W_{Sub} 中实例访问历史流的最大区间 $R_{Ai}=[P_{A1}.R_A.Start, P_{An}.R_A.End]$
4.　　对于 R_{Ai} 内的事件
5.　　进行复杂事件 P_A 的检测；
6.　　将达到最大状态的 P_A 实例放入 $Inst_{Aj}$ 中，将未达到最大状态的 P_A 实例放入中间结果集合 PM_A 中；
7.　　如果 R_{Ai} 中的事件已处理完
8.　　将 $Inst_{Li}$ 中的实例与 $Inst_{Aj}$ 中的实例在 $EventID$ 属性和其它属性上做连接操作 $CE=Inst_{Li}\bowtie_{EventID\wedge R_{Ai}\wedge P}Inst_{Ai}$；
9.　　输出满足约束的复杂事件 CE；
10.　处理滑动窗口 W_L 内的其它事件；
11.　如当前 W_L 一处理完，删除失效的事件和中间结果，向前滑动.

在算法 3 中，滑动窗口 W_L 按实例聚集被划分为若干小的子窗口 W_{Sub}，离线复杂事件检测以子窗口为单位进行，子窗口的大小决定历史流的访问和计算频率：如果 $|W_{\text{Sub}}|=1$，复杂事件检测就等价于 LLAA 算法；如果子窗口过大，如 $|W_{\text{Sub}}|=|W_L|$，即将整个窗口的实时流处理完后再进行历史流的处理，则复杂事件检测的平均响应时间将增加.

SLLAA 算法的代价模型如下：

$$Cost_{\text{LLAA}}=|W_L|\times Cost_L+n\times Cost_{RDB}+\sum_{i=1}^{n}Cost_A\times|R_{Ai}|+Cost_{P_L\bowtie P_A}\times N_P+Cost_{WPM}+Cost_{WDB}\quad(4)$$

其中 $n\times Cost_{RDB}$ 为以 n 个 W_{Sub} 单位读取数据库的代价，$\sum_{i=1}^{n} Cost_A \times |R_{Ai}|$ 为在每个 W_{Sub} 上进行 P_A 模式匹配的代价. *SLLAA* 的算法复杂度为 $O\left(\frac{(|W_L|)}{n}\times \max_{i\in[1,n]}\{|R_{Ai}|\}\right)$，可以看到，只要选择了合适的 n，SLLAA 算法的代价比 LLAA 的要小很多.

以实时流上达到最大状态的实例集合为单位进行历史流访问时，为了确保事件检测的完整性(防止复杂事件的漏检)和一致性，需要将实时流和历史流上未达到最大状态，但生命周期还在滑动窗口 W_L 内的部分匹配结果缓存起来，以便后续的事件检测进行相应的状态更新.

5 中间结果管理

随着数据流的流速的变化和复杂事件检测的推进，复杂事件检测将产生大量未到达最大状态的部分模式匹配结果(partial pattern match results 简称中间结果，PM). 例如在一个基于 RFID 的实时物品监控场景中，用户定义一个检测 10 h 内经过 RFID 阅读器 A，B，C 和 D 的物品，该复杂事件可以描述为一个序列(SEQ)模式 $P=SEQ(A;B;C;D)$[10 h]，假设滑动窗口为 *tumbling* 类型(处理完一整个窗口后向前滑动)，模式匹配缓冲区大小是固定的. 则在缓冲区中存在着的中间结果集 $PM=\{A, SEQ(A;B), SEQ(A;B;C)\}$ 的实例，这些实例随着复杂事件的推进可能溢出缓冲区，为了保证复杂事件检测的完整性和一致性，需要对这些没有超出滑动窗口区间的中间结果进行归档，并在复杂事件检测过程中不断更新它们的状态. 在线-离线复杂事件检测同样存在中间结果管理的问题，我们主要针对实时流介绍相关管理算法，历史流的中间结果管理类似.

5.1 基于时态的中间结果管理

基于时态的中间结果管理(Temporal Partial Match Management，TPM)方法如图 9 所示. 中间结果按产生的时间顺序被放入中间结果缓冲区，缓冲区中存储的是中间结果的全部信息(例如事件标识符 TID、事件的状态 $State$、中间结果产生的时间 T_S 和最后更新的时间 T_E)，通过一个 B^+ 树数据结构来辅助中间结果的状态更新操作和插入，其中 B^+ 树的键值为(TID，$State$)，树的叶节点指向对应的中间结果实体，其记录一直存储在内存中，直至对应的中间结果输出复杂事件或该中间结果无效才被删除. 中间结果缓冲区有一个归档的界限，当达到这个界限时就通过 PMR 将对应区间上的中间结果写入到外存文件系统. 当数据流中的事件触发中间结果状态更新时，如该中间结果不在缓冲区，则通过 PMR 将其读入，并进行状态更新操作.

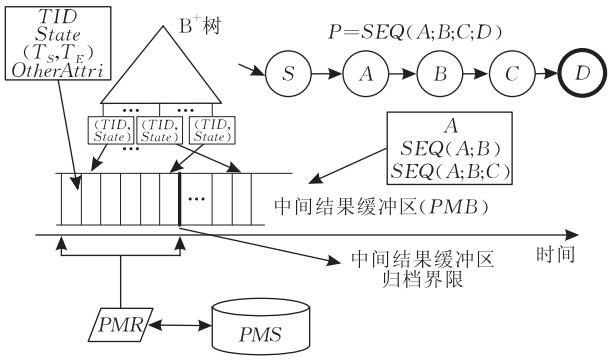

图 9 基于时态的中间结果管理

TPM 是一种利用中间的时态关系管理中间结果的方法，可以看到，当数据流的流速增加时，TPM 会频繁地将产生的中间结果归档到外存，而当实时流中的事件触发中间结果状态更新时，由于其对应的中间结果有可能不在内存中，TPM 需要到中间结果存储区(PMS)中去读取，这就导致了中间结果频繁的被读写操作. TPM 这种中间结果管理方法适用于中间结果更新少的应用，批量的中间结果归档很高效. 如图 3(a)中的股票流上连续模式的在线-离线复杂事件检测，历史流上的“√”模式被检测出来后就不需要进行更新，而实时流上的“\”模式只与后续的股票事件关联，不需要去访问历史的中间结果. 但对于图 3(b)中的离散模式，TPM 就不适用了. 因为离散模式的子事件分布在数据流的任意时间点，在未达到复杂事件的输出状态且滑动窗口未滑动没有滑动前，中间结果还处于在其生命周期中，数据流上新的事件随时都可能触发中间结果的状态更新操作，所以 TPM 只利用时态关系管理中间结果是不够的，还需要考虑中间结果的空间信息.

5.2 基于时空关系的中间结果管理

针对 TPM 管理中间结果的不足，提出基于时空关系的中间结果管理(Spatial Temporal Partial Match Management，STPM)方法，如图 10 所示.

STPM 方法中，中间结果的时态关系以优先级的方式体现，优先级定义如下.

定义 7. 中间结果优先级. 假设给定一个模式 $P=SEQ(E_1,\cdots,E_i,E_{(i+1)},\cdots,E_n)$，事件类型 E_i 和 $E_{(i+1)}$ 的事件发生的时间间隔最大值为 Δt_i，$p(E_{(i+1)}|E_i)$ 为 E_i 类型事件发生后 $E_{(i+1)}$ 类型事件发生的条

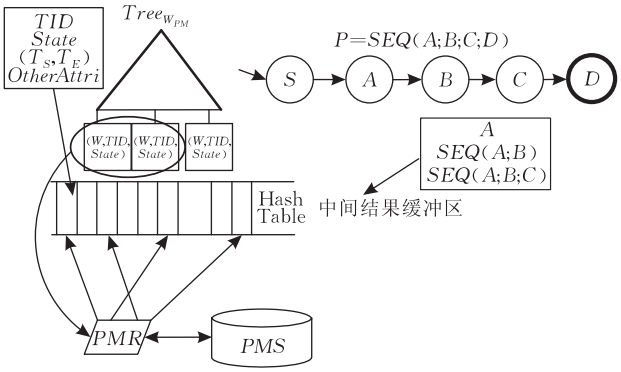

图 10　基于时空状态的中间结果管理

件概率. 对于数据流上的一个 E_i 类型($i<n$)的事件 e,它产生的中间结果 PM_e 的优先级 W_{PM_e} 定义为

$$W_{PM_e} = e.t + \Delta t_i \times p(E_{(i+1)} \mid E_i) \qquad (5)$$

即中间结果优先级定义为简单事件当前的时间与下个类型事件可能发生的时间间隔之和. W_{PM_e} 体现了中间结果的时空信息. 基于 W_{PM_e} 的中间结果管理如过程 1 所示.

过程 1.　STPM.

1. 对于一个事件 e
2.　如在 PMB 存在它的 PM_e 实体
3.　　$tmpState = PM_e.State$;
4.　　$PM_e.State = PM_e.State \cup e.State$;
5.　　如果 $tmpState \neq PM_e.State$;
6.　　//在 $Tree_{W_{PM}}$ 插入一条记录
7.　　$W_{PM_e} = e.t + \Delta t_i \times p(E_{(i+1)} \mid E_i)$;
8. $Tree_{W_{PM}}.Insert(W_{PM_e}, e.TID, e.State)$
9. 如果 PMB 达到归档上限 ΔR
10.　　$PMR.Write(\Delta R, Tree_{W_{PM}}, PMB)$;
11. 如在 PMB 不存在它的 PM_e 实体,但存在其 $Hash$ 键值
12.　　$PMR.Write(e.TID, PMS)$ $PMR.Write(e.TID, PMS)$;
13. 处理下一个事件.

当事件 e 进入复杂事件处理引擎后,首先查询 PMB 中是否存在它的 PM_e 实体. 如存在,则需要将 e 的状态更新到 PM_e 上,如 PM_e 的状态未更新(说明该事件已出现过,如 RFID 数据流中的重复读事件),则不需更新优先级树 $Tree_{W_{PM}}$,否则根据式(5)计算 PM_e 的 W_{PM_e},并在 $Tree_{W_{PM}}$ 插入一个新的记录. 需要注意的是,同一 TID 的事件在 $Tree_{W_{PM}}$ 上可能对应多条记录,权值越小的记录越靠近树的左边,在归档过程中可以通过与中间结果的状态比较将不是最新状态的 $Tree_{W_{PM}}$ 记录删除,减少 $Tree_{W_{PM}}$ 的操作.

中间结果的归档是通过一个计数器 ΔR,当中间结果中的记录达到 ΔR 时就进行一次归档操作:在 $Tree_{W_{PM}}$ 上按优先级从小到大取 ΔR 长的中间结果进行归档. 归档时要比较 $Tree_{W_{PM}}$ 上记录的状态和中间结果的最新状态,将不是最新状态的 $Tree_{W_{PM}}$ 上的记录删除,将是最新状态的记录归档到 PMS.

对于实体不在内存的中间结果,需要通过 $PMR.Write(e.TID, PMS)$ 将其读入,更新其状态和 $Tree_{W_{PM}}$.

可以看到,STPM 是一种启发式方法,它始终将时态信息和状态信息最新的中间结果保留在中间结果缓冲区,以便为复杂事件检测提供更高的命中率,减少复杂事件的输出的响应时间. STPM 的不足在于进行中间结果归档时,基于 Hash 的方法无法按中间结果时间属性直接进行批量插入操作,但可以通过对将归档的中间结果排序来完成,所以 STPM 稍作调整后也适用于连续模式的中间结果管理. 此外,STPM 的 $Tree_{W_{PM}}$ 虽然可能存储了多个 TID 相同的记录,但随着复杂事件检测的推进,这些记录会被不断地删除掉,所以 STPM 耗费的空间代价与 TPM 只有微弱的差异.

6　实验结果与分析

为了测试文中提出的方法,我们设计了一个在线-离线复杂事件检测原型系统. 该复杂事件检测原型系统用 C++ 实现,编译环境为 Visual Studio 2008,使用的数据库是开源数据库 MySQL Server 5.2,系统配置:CPU 为 Pentium(R) E5200 Dual Core 2.52 GHz,内存为 2 GB RAM,操作系统为 Windows XP.

由于本文主要考虑离散模式的在线-离线复杂事件检测,为了测试本文算法,我们设计了一个数据模拟器,模拟类似图 3(b)所示离散模式的在线-离线复杂事件检测. 实验分析使用的主要参数如表 2 所示,其中一些参数为数据生成器的参数. 历史数据流被加载到 SPE 的一个缓冲区,当缓冲区内不能存储所请求的历史流时,通过按时间顺序批量加载.

表 2　实验参数

参数	说明
N_E	数据流中事件类型的数目
M_{Ratio}	数据流中复杂事件的比例,[0,1]
$p(B \mid A, \Delta t)$	A 类型事件在 Δt 出现在 B 的概率
R_{Buffer}	实时流缓冲区
$\lvert W_{\text{Sub}} \rvert$	子窗口大小
N_S	数据流长度
P	模式,序列模式(SEQ)
ΔR	实时流实例访问历史流的区间

复杂事件定义被解析为自动机模型进行检测.实验的性能指标为在线-离线复杂事件检测的 CPU 耗费(处理完实时流的时间)和未命中次数(missed hit times).

实验 1. ALAA 与 LLAA 的 CPU 性能比较.

图 11 是两种不同的在线-离线执行算法的性能比较.由于 ALAA 算法在实时流和历史流整个数据集上运行,而 LLAA 算法只在其需要的历史流上运行,所以为了性能比较的公平,我们限定了实时流和历史流的大小,并用均匀分布生成数据流,匹配率设为 0.5, ΔR 为固定值.

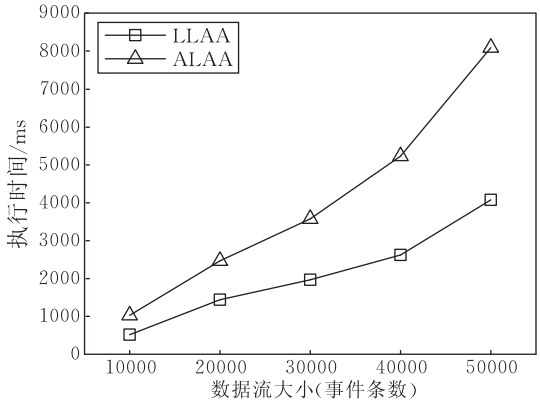

图 11 ALAA 与 LLAA 的 CPU 性能比较
($M_{\text{Ratio}}=0.5$, ΔR 为固定值)

从图 11 可以看到,ALAA 的 CPU 耗费始终比 LLAA 高,其主要原因是 ALAA 是一种盲目的在线-离线检测算法,它要加载整个历史流并检测其上的 P_A,而 LLAA 是一种触发式的在线-离线复杂事件检测方法,只有在存在达到最大状态的 P_L 时才触发一次归档流的访问和计算.由于 ALAA 比 LLAA 算法性能相差较大,所以后续实验的性能测试我们主要关注 LLAA.

实验 2. 基本的 LLAA 算法与基于选择性的 LLAA 算法的 CPU 比较.

基于选择性的 LLAA 算法(Optimized LLAA, OLLAA)性能主要测试不同选择性的时候,在线-离线事件检测选择的数据流顺序的变化对 CPU 耗费的影响.该组实验在大小为 1～5 万的数据集上进行,该组实验我们固定了实时流的选择度,调整历史流的选择度,基本的 LLAA 算法用实时流去连接历史流,而 OLLAA 算法则是历史流连接实时流.从图 12 可以看出,OLLAA 在 CPU 耗费上更少,其原因是使用基于选择度的复杂事件检测减少了不必要的 P_L 实例和 P_A 实例的计算和连接操作.但随着历史流的选择度逐渐增大,LLAA 与 OLLAA 的算法性能差异变小,主要原因是因为两个数据集的可选择度接近时,连接操作的次数与连接操作的顺序关系不是很大.在实际处理中,历史流的选择度可以通过数据库的统计工具获得,而实时数据流的选择度则可以通过一些抽样技术获得.

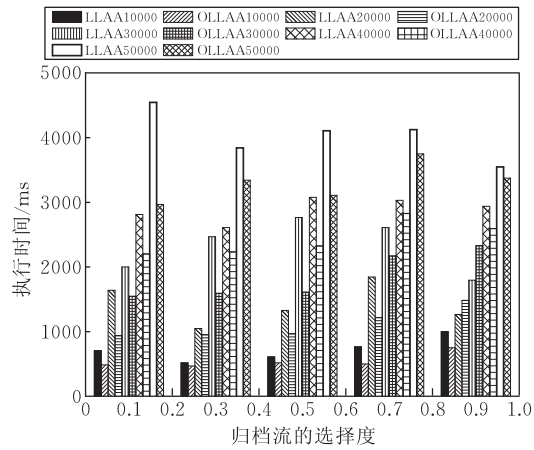

图 12 调整历史流的选择度 LLAA 与 OLLAA 的 CPU 性能比较($|W_{\text{Sub}}|=2000$, N_S:1～5 万,实时流的 selectivity 为 1.0)

实验 3. 基本的 LLAA 算法与 SLLAA 算法性能比较.

该组实验测试基本的 LLAA 算法和 SLLAA 算法的 CPU 性能比较.SLLAA 的子窗口的选取值分别为 500,1000,1500,2000,2500 和 3000.为了说明子窗口效果,我们将数据流的复杂事件的 M_{Ratio} 设置为 1.从图 13 可以看出,SLLAA 算法比 LLAA 算法的 CPU 耗费都要少,主要原因是SLLAA算法通过批量的历史流加载和计算,减少了对历史流的请求次数(读写操作)和具有交叉历史流访问区间的重复计算,从而使得在线-离线复杂事件检测的平均响应时间减少.图 13 还表明,子窗口的选取并不是

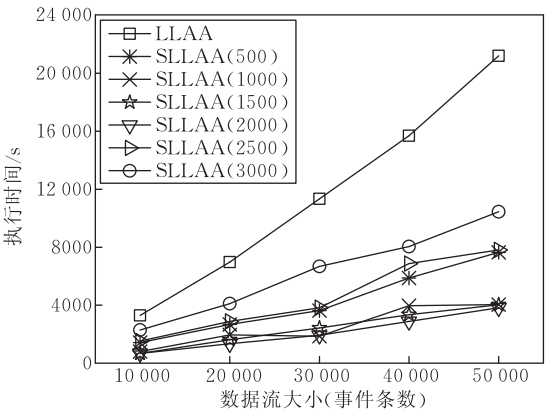

图 13 LLAA 与 SLLAA 的 CPU 性能比较
($R_{\text{Buffer}}=5000$, $M_{\text{Ratio}}=1$)

越大或越小，SLLAA 的性能就越好. 相反的是，子窗口的取值在一个取值区间的中间部分(如本实验的[500,3000])时，SLLAA 可达到 CPU 性能最优. 该组实验可以为用户设定合适的容忍时间上限提供很好的推荐：用户可以根据系统的历史处理信息，如窗口大小、数据流的分布、模式特点、子窗口的取值区域等，结合系统当前的特点，设定比较合适的响应时间，达到系统资源的合理利用.

实验 4.　调整 LLAA 的 R_{Buffer} 性能比较

在前面的章节介绍过，R_{Buffer} 是一个数据缓冲区，可以缓冲实时的数据量，为实时复杂事件检测提供数据源，同时也为历史流上的复杂事件检测提供数据源. R_{Buffer} 不能太小，因为当数据流高速进入事件处理引擎时，R_{Buffer} 由于可能存储不了数据流而频繁地将 Buffer 中的数据归档到外存，同时历史流上的归档操作也会因为 R_{Buffer} 没有其需要的数据而频繁地访问外存，这就导致系统的高负载. 由于内存是一个公共资源，当 R_{Buffer} 的空间增大时，系统分配给模式匹配的空间自然减少，而模式匹配要存储原始事件的所有信息，因此，如果 R_{Buffer} 过于大，则系统的性能会下降. 图 14 表明，较小的 R_{Buffer} 导致较多的数据库读写操作，而较大的 R_{Buffer} 则可以减少数据库的读写操作，但 R_{Buffer} 不能无限大.

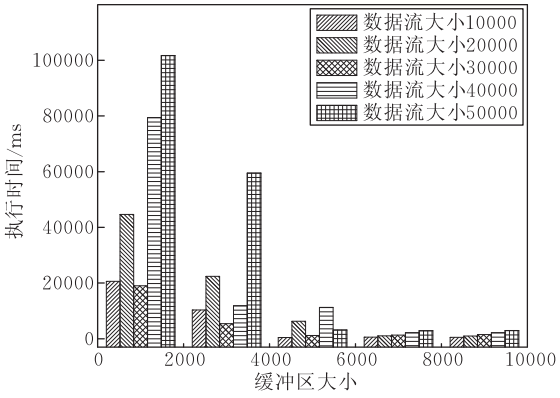

图 14　LLAA 运行于不同的 R_{Buffer} 和数据集时的 CPU 性能($M_{\text{Ratio}}=0.5$)

实验 5.　TPM 与 STPM 的 CPU 性能比较.

LLAA 在 STPM 与 TPM 两种中间结果管理方法上的性能测试如图 15 所示，为了能明显看到中间结果管理的效果，实验中设置 $|W_{\text{Sub}}|=5000$，$R_{\text{Buffer}}=50\,000$. 从图 15 可看出，STPM 的 CPU 性能明显优于 TPM，其主要原因在上节也介绍过，是由于 TPM 没有利用到中间结果的空间信息，盲目地将中间结果归档，导致事件检测过程的中间结果命中率急剧下降，命中率下降意味着需要多次的数据库读取操作，数据库的频繁操作是非常耗时的. 而 TPM 利用中间结果的时空关系(虽然空间关系时态化了)，将最有可能更新状态的中间结果放于内存，提高了复杂事件的命中率，减少了复杂事件输出的时间.

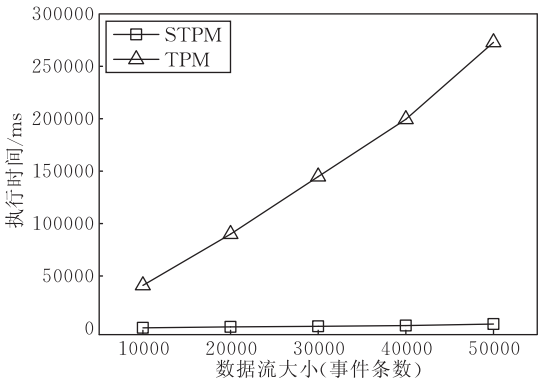

图 15　STPM 与 TPM 的 CPU 性能比较
($|W_{\text{Sub}}|=5000$，$R_{\text{Buffer}}=50\,000$，$M_{\text{Ratio}}=0.5$)

实验 6.　调整 LLAA 的 M_{Ratio} 时 TPM 与 STPM 的 CPU 性能和 Miss 次数比较

我们调整数据集的大小和匹配率 M_{Ratio}，测试了 STPM 和 TPM 的 CPU 性能和处理完相同数据集后的 Miss 次数统计. STPM 和 TPM 的 CPU 性能比较如图 16 所示，可以看出，在数据集和 M_{Ratio} 变化时，STPM 的 CPU 耗费稳定且都远小于 TPM，进一步说明了中间结果管理需要更多的查询的特性才能使复杂事件检测响应时间更短. STPM 和 TPM 的 CPU 性能差异其实来自于模式匹配时中间结果的 Miss 次数的差异，如图 17 所示.

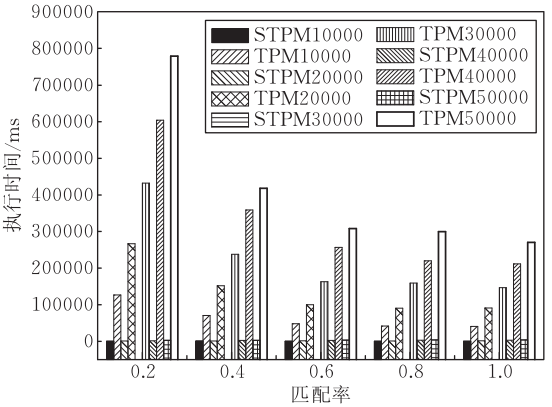

图 16　STPM 与 TPM 的 CPU 性能比较
($|W_{\text{Sub}}|=5000$，$R_{\text{Buffer}}=50\,000$)

可以看到，随着 M_{Ratio} 的变大，TPM 搜寻中间结果缓冲区的失效次数逐步减小，而 STPM 的中间结果缓冲区搜索失效次数保持在一个很低且稳定的区域(10～20 次左右)，即 STPM 能在缓冲区内保存更

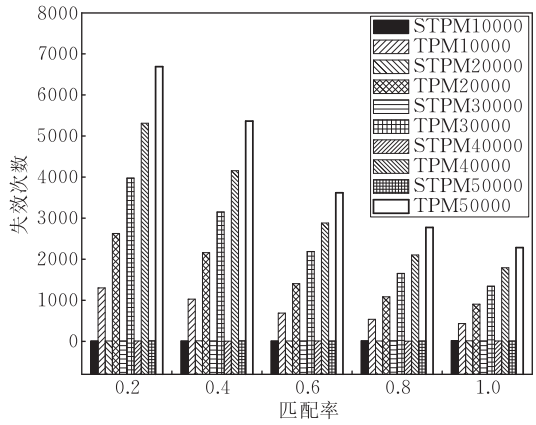

图 17 STPM 与 TPM 的中间结果 Miss 次数比较 ($|W_{Sub}|=5000$, $R_{Buffer}=50000$)

为活跃的中间结果,大大减少了归档中间结果的访问代价.

7 结 论

本文研究了在线-离线(历史)数据流上的复杂事件检测问题.通过使用最近处理过的事件缓冲区缓存最近处理过的实时数据流,可以减少历史流的请求代价.提出了在线-离线复杂事件检测算法 ALAA 和 LLAA,并通过数据流的事件分布特性提出了 LLAA 的优化算法.针对复杂事件处理大窗口和高流速时产生的大量中间结果,提出了基于时态和时空关系的中间结果管理方法 STPM 和 TPM.通过利用中间结果的时态和空间状态信息,STPM 减少了状态需要更新的中间结果被交换到外存的可能性,提高了复杂事件检测的中间结果命中率,减少了复杂事件检测的响应时间.本文目前研究的是离散模式的在线-离线复杂事件检测,系统中只有一个事件查询.实际应用中,可能存在多个用户同时在实时流和历史流上查询,每个查询由于其语义(模式,窗口等)的不同而对应不同的查询执行模型(不同的自动机或 Tree),如何在一次扫描数据流的时候或一次扫描多个查询对应的自动机的情况下进行查询调度,是多个查询优化的重要问题.此外,当系统中存在多个在线-离线复杂事件查询时,由于每个查询对应不同的时间窗口约束和历史流访问请求区间,如何利用这些查询的共同特点和数据的特点进行查询优化,如何在系统资源有限的情况下,对各种状态的中间结果进行高效管理,给系统降载,都是需要重点考虑的问题.此外,由于复杂事件检测的应用场景一般为分布式环境,数据在汇集到中心节点时,由于网络延迟或其它不确定因素,在数据流中可能存在数据乱序的情况,如何在保证在线-离线复杂事件检测完整性和一致性的前提下去除乱序数据的影响,也是需要考虑的问题.

参 考 文 献

[1] Wu E, Diao Y, Rizvi S. High-performance complex event processing over streams//Proceedings of the 2006 ACM SIGMOD international conference on Management of data. Chicago, USA, 2006: 407-418

[2] Brenna L, Alan D, Johannes G, Mingsheng H, Joel O, Biswanath P, Mirek R, Mohit T, Walker W. Cayuga: A high-performance event processing engine//Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. Beijing, China, 2007: 1100-1102

[3] Zemke F, Witkowski A, Cherniak M, Colby L. Pattern matching in sequences of rows. ANSI Standard Proposal: Technical Report, 2007

[4] Balazinska M, Kwon Y, Kuchta N, Lee D. Moirae: History-enhanced monitoring//Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research. Asilomar, USA, 2007: 375-386

[5] Nihal D, Baris Gc, Patrick L, Asli O, Merve S, Nesime T. DejaVu: Declarative pattern matching over live and archived streams of events//Proceedings of the ACM SIGMOD International Conference on Management of Data. Providence, USA, 2009: 1023-1026

[6] Nihal D, Peter M. F, Merve S, Nesime T. Efficiently correlating complex events over live and archived data streams//Proceedings of the 5th ACM International Conference on Distributed Event-Based System. New York, USA, 2011: 243-254

[7] Peng S, Li Z, Li Q. Event detection over live and archived streams//Proceedings of the 12th International Conference on Web-Age Information Management (WAIM 2011). Wuhan, China, 2011: 566-577

[8] Zimmer D. On the semantics of complex events in active database management systems//Proceedings of the 15th International Conference on Data Engineering. Sydney, Australia, 1999: 392-399

[9] Adi A, Etzion O. Amit- the situation manager. The VLDB Journal, 2004, 13(2): 177-203

[10] Li G, Jacobsen H-A. Composite subscriptions in content-based publish/subscribe systems//Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware. Grenoble, France, 2005: 249-269

[11] Abadi D J, Carney D et al. Aurora: A new model and architecture for data stream management. The VLDB Journal, 2003, 12(2): 120-139

[12] Chandrasekaran S, Cooper O, Deshpande A, Franklin M J, Hellerstein J M, Hong W, Krishnamurthy S, Madden S R,

Reiss F, Shah M A. TelegraphCQ: Continuous dataflow processing//Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. San Diego, USA, 2003: 668-668

[13] Hussein EA-S, Abdel-Wahab. HiFi: A new monitoring architecture for distributed systems management//Proceedings of the 19th IEEE International Conference on Distributed Computing Systems. Austin, USA, 1999: 171-178

[14] Agrawal J, Diao Y, Gyllstrom D, Immerman N. Efficient pattern matching over event streams//Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. Vancouver, Canada, 2008: 147-160

[15] Zhang H, Diao Y, Immerman N. Recognizing patterns in streams with imprecise timestamps. Proceedings of the VLDB Endowment, 2010, 3(1-2): 244-255

[16] Mei Y, Madden S. ZStream: A cost-based query processor for adaptively detecting composite events//Proceedings of the 35th SIGMOD International Conference on Management of Data. Providence, USA, 2009: 193-206

[17] Gatziu S, Geppert A, Dittrich KR. Integrating active concepts into an object-oriented database system//Proceedings of the 3rd International Workshop on Database Programming Languages. Nafplion, Greece, 1992: 399-415

[18] Chandrasekaran S. Query processing over live and archived data streams [Ph. D. dissertation]. University of California, Berkeley, 2005

[19] Reiss F, Stockinger K, Wu K, Shoshani A, Hellerstein J M. Enabling real-time querying of live and historical stream data//Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007). Banff, Canada, 2007: 28-37

[20] Wang W, Sung J, Kim D. Complex event processing in EPC sensor network middleware for both RFID and WSN//Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing. Orlando, USA, 2008: 165-169

[21] Strom R, Dorai C, Buttner G, Ying L. SMILE: Distributed middleware for event stream processing//Proceedings of the 6th International Conference on Information Processing in Sensor Networks. Cambridge, USA, 2007: 553-554

[22] Eyers D M, Roberts B, Bacon J, Papagiannis I, Migliavacca M, Pietzuch P, Shand B. Event-processing middleware with information flow control//Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. Urbanna, USA, 2009: 1-2

[23] Eyers D M, Vargas L, Singh J, Moody K, B Jean. Relational database support for event-based middleware functionality//Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems. Cambridge, United Kingdom, 2010: 160-171

[24] Magid Y, Sharon G, Arcushin S, Ben-Harrush I, Rabinovich E. Industry experience with the IBM Active Middleware Technology (AMiT) Complex Event Processing engine//Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems. Cambridge, United Kingdom, 2010: 140-149

[25] Wang F, Liu S, Liu P. Complex RFID event processing. The VLDB Journal, 2009, 18(4): 913-931

[26] Peng S, Li Z, Li Q. Efficient multiple objects-oriented event detection over RFID data streams//Proceedings of the 11th International Conference on Web-Age Information Management (WAIM 2010). Jiuzhaigou, China, 2010: 97-102

[27] Wei M, Liu M, Li M, Golovnya D, Rundensteiner E A, Claypool K. Supporting a spectrum of out-of-order event processing technologies: From aggressive to conservative methodologies//Proceedings of the 35th SIGMOD International Conference on Management of Data. Providence, USA, 2009: 1031-1034

[28] Chandramouli B, Goldstein J, Maier D. High-performance dynamic pattern matching over disordered streams. Proceedings of the VLDB Endowment, 2010, 3(1-2): 220-231

[29] Johnson T, Muthukrishnan S, Rozenbaum I. Monitoring regular expressions on out-of-order streams//Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007). Istanbul, Turkey, 2007: 1315-1319

[30] Liu M, Li M, Golovnya D, Rundensteiner E A, Claypool K. Sequence pattern query processing over out-of-order event streams//Proceedings of the 2009 IEEE International Conference on Data Engineering. Shanghai, China, 2009: 784-795

[31] Liu M, Ray M, Rundensteiner E A, Dougherty D J, Chetan G, Wang S, Ari I, Mehta A. Processing nested complex sequence pattern queries over event streams//Proceedings of the 7th International Workshop on Data Management for Sensor Networks. Singapore, 2010: 14-19

[32] Wasserkrug S, Gal A, Etzion O, Turchin Y. Complex event processing over uncertain data//Proceedings of the 2nd International Conference on Distributed Event-Based Systems. Rome, Italy, 2008: 253-264

PENG Shang-Lian, PENG Shanglian, born in 1980, Ph. D. candidate. His main research interests include RFID data management and complex event processing etc.

LI Zhan-Huai, LI Zhan Huai, born in 1961, professor, Ph. D. supervisor. His research interests include database theory, massive data storage etc.

CHEN Qun, born in 1976, professor, Ph. D. supervisor. His research interests include XML data management, complex event processing, data quality and cloud computing etc.

LI Qiang, born in 1986, master candidate. His main research interests include complex event processing and RFID data management.

Background

Complex event processing (CEP) is a state-of-art data processing technique which has been widely utilized in scenarios such as network monitoring, financial trend prediction, RFID based object tracking, supply chain management, telecommunication monitoring, etc. Both enterprises and academia have been paying great attention to CEP. With the emergence of Internet of Things(IOT), large amount of data will be generate in a distributed environment enabling more adaptation of online processing algorithms(such as CEP).

Generally, CEP is running on data streams with one-pass scan of the event stream, data streams are discarded after CEP. However, as described in the introduction section of this paper, many scenarios need to archive historical data streams and CEP queries can be subscribed over both live and archived data streams. So efficient integration of historical stream access into CEP becomes increasingly important. In this paper, we consider overall issues of CEP over live-archived streams (LA-CEP) including stream buffer mechanism, event detection algorithms and optimizations and partial pattern match management. With stream buffer mechanism, recent buffer is introduced which caches the latest events in memory before the events are archived. This mechanism provides the archived stream processor with freshest data so data base access cost is reduced, further, streams can be archived into database in batch size. Active and lazy event detection algorithms for LA-CEP are proposed. Selectivity of event types is used to optimize event stream join order and the sliding window is divided into subwindows thus live stream pattern match is carried out in subwindow unit, both of the optimization techniques can reduce response time of CEP efficiently. Spatial-temporal information based partial match management method is proposed, with this method the latest and the most probably updated partial match results are kept in memory which can reduce data base access cost largely and enable fast complex event generation.

This work is supported by the National Natural Science Foundation of China (NSFC) (grant Nos. 60970070, 60803043, 60873196, and 61033007) and the National High Technology Research and Development Program of China (863 Program) under grant No. 2009AA01A404.