软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(8):2138−2154 [doi: 10.13328/j.cnki.jos.004733]

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

Mashup 运行时的性能优化方法*

张 鹏[1,2], 刘庆云[1,2], 徐克付[1,2], 林海伦[3], 孙 永[1,2], 谭建龙[1,2]

[1](中国科学院 信息工程研究所,北京 100093)

[2](信息内容安全技术国家工程实验室,北京 100093)

[3](中国科学院大学,北京 100049)

通讯作者: 张鹏, E-mail: pengzhang@iie.ac.cn

摘 要: 近几年来,互联网上出现了一类称为 Mashup 的新型应用,它使最终用户能够个性化地聚合和操作分布在互联网上的数据源.然而,关于 Mashup 在动态环境下运行时的性能研究还比较缺乏.为此,利用缓存技术提出了 Mashup 运行时的性能优化方法——POMO.POMO 具有以下 3 个主要创新点:首先,POMO 通过算子序列的缓存点的成本和收益模型实现了动态缓存点选取;其次,POMO 通过缓存点的 B+树索引实现了缓存点重用;第三,POMO 通过两阶段切换数据传输协议实现了缓存点更新.实验分析结果表明:POMO 减少了 Mashup 在动态环境下的运行成本,提高了 Mashup 运行时的性能.

关键词: Mashup;缓存点;B+树;索引;增量传输

中图法分类号: TP316

中文引用格式: 张鹏,刘庆云,徐克付,林海伦,孙永,谭建龙.Mashup 运行时的性能优化方法.软件学报,2015,26(8):2138−2154. http://www.jos.org.cn/1000-9825/4733.htm

英文引用格式: Zhang P, Liu QY, Xu KF, Lin HL, Sun Y, Tan JL. Approach to performance optimization of Mashup operation. Ruan Jian Xue Bao/Journal of Software, 2015,26(8):2138−2154 (in Chinese). http://www.jos.org.cn/1000-9825/4733.htm

Approach to Performance Optimization of Mashup Operation

ZHANG Peng[1,2], LIU Qing-Yun[1,2], XU Ke-Fu[1,2], LIN Hai-Lun[3], SUN Yong[1,2], TAN Jian-Long[1,2]

[1](Institute of Information Engineering, The Chinese Academy of Sciences, Beijing 100093, China)

[2](National Engineering Laboratory for Information Security Technologies, Beijing 100093, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: In recent years, a new type of applications called “Mashup” has developed on the Internet. These applications can help end-users to collect and operate the distributed data sources on the Internet. However, the research on performance optimization of Mashup operation is lacking. To address this shortfall, this paper proposes a new approach, POMO, for performance optimization of Mashup operation. POMO engages progress in three areas. Firstly, it implements dynamic cache point selection through the tradeoff between operation cost and caching benefit. Secondly, it implements cache point reusing by B+ tree index. Thirdly, it implements cache point update through incremental transmission. Experimental results show that POMO reduces the cost of Mashup operation and improves the performance of Mashup operation in dynamic environment.

Key words: Mashup; cache point; B+ tree; index; incremental transmission

随着互联网技术的发展以及应用的深化,特别是面向服务体系结构(service oriented architecture,简称 SOA)、软件即服务(software as a service,简称 SaaS)、Web 2.0 等新兴互联网应用架构、模式与技术的发展,应用软件逐渐转移到互联网这一开放、动态、难控的协同计算平台上进行开发和运行,并体现出分布、开放、动

* 基金项目: 国家自然科学基金(61402464); 中国博士后基金(2013M541076)
收稿时间: 2013-04-15; 修改时间: 2014-01-21; 定稿时间: 2014-09-28

态、快速演变、个性化等特征[1].在上述发展趋势下,如表1所示,互联网上的资源越来越多地以Web服务形式对外提供.通过对网络上封装各类资源的Web服务的共享和集成来构造和支撑应用软件,正逐渐成为一种新兴的、主要的方式.

Table 1 Service resources on the Internet

表1 互联网上的服务资源

资源类型	数量	接口形态	相关标准
Web数据库服务	>450 000个[2]	通过网站查询接口提交查询,返回动态生成的HTML网页	HTTP, HTML
RSS/Atom种子资源	>1亿个[3]	HTTP请求返回结构化的RSS,Atom内容	HTTP, RSS, Atom
Open API	>1 000个[4]	通过HTTP协议提供REST风格或RPC风格的Web访问接口	HTTP, OAuth, OpenID, Javascript, OpenSocial
SOAP Web services	>400个[5]	通过SOAP协议提供的具有WSDL描述的访问接口	SOAP, WSDL, WS-*

Mashup作为由最终用户创建并且使用的Web服务,由于能够提供给最终用户高度的个性化使其变得越来越流行.据ProgrammableWeb统计表明,每天新增的Mashup有6.29个[4].Mashup之所以能够提供高度的个性化,是因为Mashup由最终用户开发,而不像传统的Web服务是由专业人员开发.Mashup的主要功能是从分布在互联网上的多个数据源中采集数据,并且对其进行聚合和过滤,最后得到最终用户需要的结果.目前的Mashup平台包括雅虎、微软、谷歌、IBM以及SIGSIT推出的Yahoo Pipes[6],Microsoft Popfly,Google Mashup Editor,IBM Damia[7]和Mashroom[3],其特征主要体现在最终用户高度的个性化和参与度上.然而,正是这些特征给Mashup运行时的性能带来了挑战.

首先,由于用户可以自主开发属于他们自己的Mashup,导致Mashup平台托管大量的Mashup,相比传统的Web服务平台,这种方式对Mashup运行时的性能提出了更高的要求;其次,Mashup从互联网上分布的多个数据源中获取数据,这些数据源在结构特征和数据特征上存在差异,增加了内部的处理开销;最后,Mashup一般由非专业的用户开发,因此它们很可能没有从性能上进行优化,当数据源动态变化时,Mashup运行时的性能问题变得更加突出.缓存已经被证明是一种有效提高Web应用性能的技术,其中包括一些专门为Web服务设计的缓存技术[8−10].例如,Li提出的SigsitAccelerator[9]中间件能够通过缓存减少服务调用过程中的数据传输时间;此外,Papageorgiou等人讨论了针对移动客户端的Web服务的缓存机制[10].然而,这些技术不适合Mashup,原因包括以下3点:

① 尽管大部分Mashup的请求频率不是很高,但是由于托管的Mashup的数量很多,Mashup平台的总请求频率可能会很高.因此,Mashup平台产生的数据远远多于Web服务门户网站[11].由于Web服务缓存技术只能在固定点进行缓存,因此,Mashup平台中的缓存命中率会很低;

② 由于传统的Web服务由专业人员开发,它们的性能往往经过优化,并且在开发时由于遵守规范的设计原则,专业人员或许能够为Web服务选取合适的缓存点.相反,Mashup是由具有不同技术经验的最终用户所创建,它们的拓扑结构差异很大,因此很难事先选好合适的缓存点;

③ 大部分Mashup需要访问外部数据源,使得Mashup运行时的性能主要取决于外部条件,尤其是数据源的频繁更新会限制Mashup运行时的性能.

正是由于这种差异,传统的Web服务缓存技术不适合Mashup,因此,Mashup平台需要一种专门的缓存机制,其中包括:① 支持Mashup算子序列运算的中间结果能够被缓存,中间结果是否缓存要依据动态的成本和收益的分析,该分析需要考虑外部条件;② 缓存的中间结果能够被其他的Mashup快速发现和重用;③ 当数据源更新时,缓存的中间结果能够快速更新.因此,Mashup的缓存必须要重新设计,其中不仅要考虑Mashup的拓扑结构,而且还要考虑Mashup的动态环境.

然而,目前的Mashup性能优化方法主要考虑构造时的算子重新排列,例如Hassan和Lin等人提出的算子合并和重排方法[11,12],该方法通过发现Mashup应用中存在的公共算子序列来避免重复的计算,其中定义了算子的重排规则,提高了发现公共算子序列的概率.即便如此,每次Mashup请求,公共算子序列至少需要执行一次重新计算,然而这些工作没有考虑公共算子序列的计算结果是否可以通过缓存以进一步减少重新计算.同时,我们

也提出了基于数据服务的缓存点选取方法[13],但是该方法假定影响缓存点选取的参数是静态的,并且没有考虑如何重用缓存点,因此无法适合动态变化环境.为此,本文提出了 Mashup 运行时算子运算的性能优化方法——POMO,包含以下 3 个技术创新点:

(1) 动态缓存点选取技术.该技术通过对 Mashup 中不同算子序列的缓存点的成本和收益进行计算来选择最大化净收益的缓存点集合.

(2) 支持近似匹配的 B+树索引技术.该技术通过 B+树索引快速发现 Mashup 可重用的缓存点,并且通过支持近似匹配进一步提高了缓存点命中率.

(3) 两阶段切换数据传输技术.该技术能够适应不同更新频率的数据源,减少缓存点更新的数据传输时间.

我们已经通过一系列实验对 POMO 进行评价,实验分析表明,POMO 减少了 Mashup 在动态环境下运行的时间成本,提高了 Mashup 运行时的性能.

本文第 1 节讨论 POMO 的研究动机和基本原理.第 2 节重点介绍缓存点的成本和收益模型及动态缓存点选取算法.第 3 节给出缓存点重用的 B+树索引技术以及提高缓存点命中率的近似匹配技术.第 4 节重点介绍两阶段切换数据传输技术.第 5 节通过实验详细分析 POMO 的有效性.

1 POMO 的原理

本节首先给出 Mashup 的形式化表示作为分析其性能的基础,然后给出 POMO 的整体架构和原理.

1.1 Mashup模型

一个 Mashup 平台是一个包含一组 Mashup 的系统,其中,$MpSet=\{Mp_0,Mp_1,\ldots,Mp_{N-1}\}$表示在给定时刻 Mashup 平台中所有的 Mashup.Mashup 平台提供一组数据处理算子,例如 Mashroom 中的 filter,sort,join,truncate,unnest,nest,deleteColumn,addColumn 等算子.此外,Mashroom 还引入两类特殊的算子——import 算子和 sink 算子:import 算子通过调用数据服务获取外部数据源的数据,而 sink 算子则把 Mashup 的结果可视化呈现给最终用户.数据服务作为网络信息资源的统一抽象,降低网络信息资源的访问复杂性[14].$OpSet=\{op_0,op_1,\ldots,op_{M-1}\}$表示在 Mashup 平台中使用的算子集合.不失通用性,算子 op_{M-2} 和 OP_{M-1} 分别对应 import 算子(表示为 io)和 sink 算子类型(表示为 so).*OpSet* 的其余元素都是数据处理算子,它们接收满足指定条件的输入并且产生输出.一个 Mashup 由一组从 *OpSet* 中选取的算子实例所组成,其中包含一个或者多个 import 算子和 sink 算子的实例.特别地,一个 Mashup 可以被建模成一棵树,其中每个节点对应一个算子实例,该算子实例的输出是其父节点对应的算子实例的一个输入.此外,树的根节点对应 sink 算子实例,树的叶子节点对应 import 算子实例.$\{nd_0^l,nd_1^l,\ldots,nd_{Q-1}^l\}$ 表示 Mashup 标识符是 Mp_1 的树的节点集合,其中每个节点对应 *OpSet* 中的一个算子实例.当 Mashup 被建模为一棵树时,多个 Mashup 可能由于共享数据源而形成一个有向无环图.

OpSet 中的每个算子实例都有一个执行成本函数 $C^{op_j}(s_0,s_1,\ldots,s_{q-1})$,该函数表示执行算子实例 op_j 的计算成本.$s_0,s_1,\ldots,s_{q-1}$ 表示算子 op_j 的输入参数.执行成本函数可以通过不同的方式度量,例如通过算子执行的计算次数或者算子执行的计算时间.本文通过算子执行的计算次数表示一个算子实例的执行成本.Mp_1 的总成本是它的所有算子实例的成本总和.下面为了便于说明,我们把算子实例也称为算子.

1.2 POMO的原理

图 1 显示了 Mashup 平台的整体架构,该架构建立在前期工作 Mashroom 的基础上,其中植入了 POMO.图 1 左边的 POMO 包含 6 个模块,其中,统计信息模块监控 Mashup 的执行以收集相关的统计信息,例如 Mashup 的请求频率、更新频率以及算子的计算次数.其中,数据服务中触发缓存内容更新的触发器会通过两阶段切换数据传输协议主动向 POMO 发送更新数据,POMO 以此获得更新频率;Mashup 的请求频率和算子的计算次数则可以通过 Mashup 平台的数据库获取.然后,缓存点选取模块对 Mashup 树的所有节点进行缓存点的成本和收益的

分析,选择能够产生最大净收益的节点进行缓存.POMO 通过和 Mashup 编辑器进行交互,获得新建的 Mashup.对于每个新建的 Mashup,缓存点发现模块利用 B+树索引查找匹配的缓存点.如果找到了匹配的缓存点,那么缓存点重用模块会修改 Mashup 的拓扑结构,并将修改后的 Mashup 提交给 Mashup 平台来运行,使其在运行时可以重用缓存点.在 Mashup 运行时,如果数据源发生更新,则两阶段切换数据传输模块会更新缓存点.由于缓存点选取、缓存点重用和缓存点更新是主要模块,下面重点介绍它们所涉及到的技术.

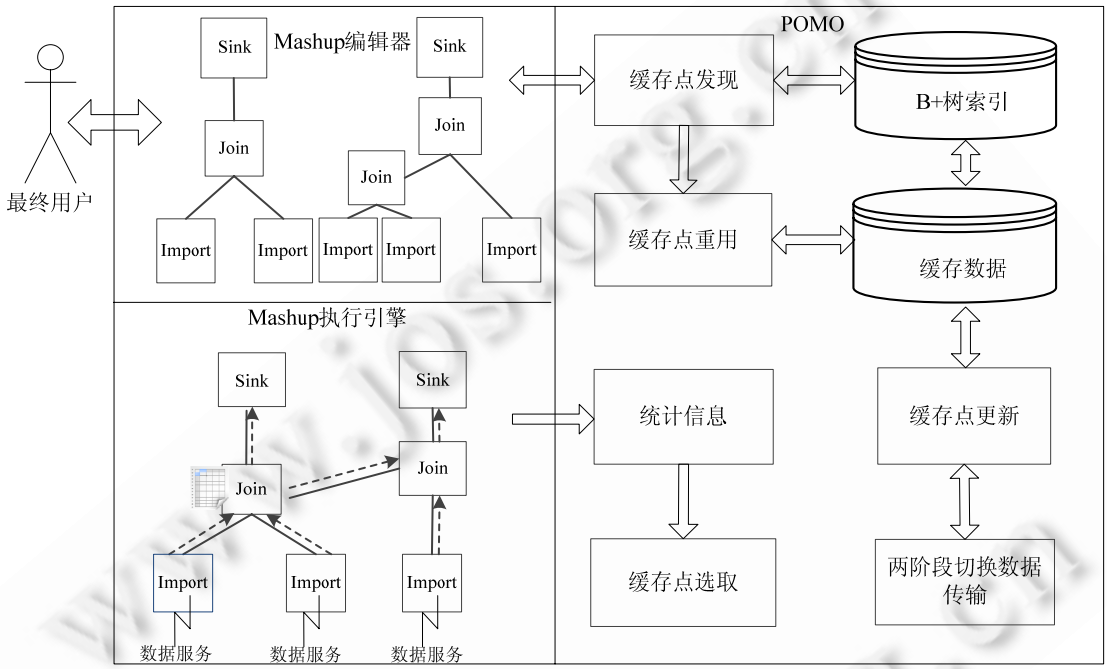


Fig.1 Mashup platform with POMO

图 1 植入 POMO 的 Mashup 平台

2 缓存点选取

在本节中,我们首先介绍如何把动态缓存点选取建模成一个优化问题,然后介绍求解该问题的算法.

动态缓存点选取可以表述成一个在成本和收益之间折中的优化问题.由于 Mashup 平台可以提供足够的存储空间满足缓存点的存储需要,因此,本文动态缓存点选取模型不会对缓存点的存储空间进行限制.

2.1 缓存点选取模型

在 Mashup 树中,除了根节点以外,其他任何节点都可以作为缓存点,因此,它们被称为潜在缓存点(potential cache point,简称 PCP),它们的集合被称为潜在缓存点集合(potential cache point set,简称 PCPS).如果潜在缓存点被选取,那么该节点被称为缓存点(cache point,简称 CP).表示 MpSet 中 Mashup 的潜在缓存点集合,其中,表示多个 Mashup 中公共算子序列的潜在缓存点只有 1 个.一个潜在缓存点 Pcp_k 的所有子节点的总计算次数加上 Pcp_k 的计算次数被称为 Pcp_k 的累积计算次数,表示为 $CC^{Pcp_k}=C^{Pcp_k}+\sum_{Pcp_h\in Descendent(Pcp_k)}(C^{Pcp_h})$.

Pcp_k 被选取为缓存点的收益是这个缓存点可以被其他包含 Pcp_k 作为其公共部分的 Mashup 所重用可以避免 Pcp_k 及其子节点的重新计算次数.Pcp_k 的请求频率 RF^{Pcp_k} 表示如果 Pcp_k 不是缓存点,单位时间内满足最终用户的请求时 Pcp_k 需要被计算的次数.注意,RF^{Pcp_k} 是所有包含 Pcp_k 的子树作为其中一部分的 Mashup 的请求频率的总和.因此,Pcp_k 设置缓存点获得的单位时间收益是 $RF^{Pcp_k}\times CC^{Pcp_k}$.

Pcp_k 被选取为缓存点的成本包括缓存点更新的传输时间和缓存点更新的计算次数.缓存点更新的传输时

间是指从数据源到叶子节点的传输时间,缓存点更新的计算次数是指从 Pcp_k 的叶子节点发生更新到缓存点发生更新的计算次数.这里首先考虑计算次数.假设 Pcp_k 的子树通过重新计算来更新缓存数据,那么 Pcp_k 以及 Pcp_k 的所有子节点都需要重新计算,因此,Pcp_k 单位时间更新的计算次数表示为 $UF^{Pcp_k}\times CC^{Pcp_k}$, 其中, UF^{Pcp_k} 表示 Pcp_k 的子树的叶子节点的更新频率之和.

Pcp_k 被选取为缓存点的净收益是 Pcp_k 被选取为缓存点的收益和成本进行折中的结果,表示为

$$CBT^{Pcp_k}=RF^{Pcp_k}\times CC^{Pcp_k}-UF^{Pcp_k}\times CC^{Pcp_k}.$$

从 CBT^{Pcp_k} 的计算公式中可以看出,满足最终用户请求的计算次数和缓存点更新的计算次数同样重要.在不同的场景中,两者的重要程度不同.为此, CBT^{Pcp_k} 可以通过选择合适的权重来体现它们的重要度.

如前面所说,动态缓存点选取的目标是选取一组缓存点使得总的净收益最大.假设 X^{Pcp_k} 是一个{0,1}变量,表示 Pcp_k 是否被选为缓存点. $X^{Pcp_k}=1$ 表示 Pcp_k 被选为缓存点,反之为 0.因此,目标函数表示为每个潜在缓存点 $Pcp_k\in PcpSet$ 分配一个 X^{Pcp_k} ,使得 $\sum_{Pcp_k\in PcpSet}(CBT^{Pcp_k}\times X^{Pcp_k})$ 最大.然而,上述问题求解的最优方案可能会得到不理想的结果,例如选取的缓存点出现了重复或者相互依赖.为了避免这种情况,我们引入下面一个条件,即:对任意一对潜在缓存点(Pcp_k,Pcp_i),如果 $Pcp_k\in Descendent(Pcp_i)$或者相反,那么 $X^{Pcp_k}+X^{Pcp_i}\leqslant 1$. 因此,缓存点的成本和收益模型如下:

$$\text{Max}\sum_{Pcp_k\in PcpSet}(CBT^{Pcp_k}\times X^{Pcp_k})$$

$$\text{s.t. } \forall(Pcp_i DescendentPcp_k), X^{Pcp_k}+X^{Pcp_i}\leqslant 1.$$

2.2 缓存点选取算法

下面介绍缓存点选取算法.首先,POMO 收集 Mashup 的相关统计信息,包括 Mashup 的请求频率、潜在缓存点的更新频率以及算子的累积计算次数;然后,POMO 对每个 Mashup 执行如下步骤:

算法 1. Caching Point Selection(CPS).

Input: PCPS. //潜在缓存点集合

Output: CPS. //缓存点集合

1. *SMCount*←*PCPS.getSMCount*; *Height*←*PCPS.getHeight*
2. $CSP\leftarrow \max_{Pcp}\left(\frac{SMCount}{Height}\right)$
3. *CP.add*(*CSP*)
4. *mode*←−1
5. **do** {
6. $CBT^{CSP}\leftarrow RF^{CSP}\times CC^{CSP}-UF^{CSP}\times CC^{CSP}$
7. **if** (*mode*!=1) *C*←*CSP.getChildren*(); $CBT^{C}\leftarrow RF^{C}\times CC^{C}-UF^{C}\times CC^{C}$
8. **if** (*mode*!=0) *P*←*CSP.getParent*(); $CBT^{P}\leftarrow RF^{P}\times CC^{P}-UF^{P}\times CC^{P}$
9. **if** ($CBT^{P}>CBT^{CSP}$)
10. *CPS*←*P*; *mode*←1; *CP.add*(*CSP*)
11. **else if** ($CBT^{C}>CBT^{CSP}$)
12. *CPS*←*C*; *mode*←0; *CP.add*(*CSP*)
13. **else**
14. *CP.add*(CSP); *mode*←−1
15. **end if**
16. } **while** (*mode*!=−1)
17. **end while**
18. **return** *CPS*

算法1中,Step 1~Step 3是从被多个Mashup共享并且在Mashup树的较低层的潜在缓存点开始遍历,这一步可以通过对每个潜在缓存点计算*SMCount*/*Height*,然后选择结果最大的潜在缓存点来实现.其中,*SMCount*表示共享这个潜在缓存点的Mashup的个数,*Height*表示Mashup的高度.之所以从这个潜在缓存点开始遍历,是因为它们能够以相对较低的更新成本产生较高的重用度,因此能够带来较大的收益.假设算法从潜在缓存点 Pcp_k 开始遍历,那么该潜在缓存点也被称为当前搜索节点(current search point,简称CSP).Step 5~Step 16是比较 CBT^{CSP} 和*CSP*的父亲节点和儿子节点的*CBT*的大小.如果 Pcp_k 有多个儿子,就计算这些儿子节点的*CBT*的总和.如果父亲节点的*CBT*大于 Pcp_k 的*CBT*,那么这个父亲节点被选为新的*CSP*,算法从这个节点向上继续遍历;否则,如果儿子节点的*CBT*大于 Pcp_k 的CBT,那么儿子节点被选为新的*CSP*,算法从这个节点向下继续遍历.如果 Pcp_k 有多个儿子节点,那么算法从每个儿子节点向下继续遍历.如果*CSP*的父亲节点和儿子节点的*CBT*都小于它的*CBT*,则算法终止,算法终止时的所有*CSP*都被加入到缓存点集合*CSP*中.该算法通过上述方式选取每个Mashup的所有缓存点,并且最终生成缓存点选取方案.

通过分析可知,该算法能够在 $O(n\log(n))$ 时间内得到最优的缓存点选取方案,其中,n 表示潜在缓存点的个数.其实,该算法根据统计信息会不断动态地调整缓存点集合,例如,随着Mashup的增多,一个Mashup的某个潜在缓存点可以被其他Mashup重用的次数增多,那么该潜在缓存点可能被选取为缓存点;同理,当某个潜在缓存点被选取为缓存点后,如果其所带来的收益减少到一定程度,那么该缓存点也可能会被取消.

3 缓存点重用

如何快速发现并且重用缓存点是接下来需要解决的问题.实际上,缓存点中的数据是Mashup树中一个子树的计算结果,这个子树本身或许有1个或者多个叶子节点对应的数据服务.缓存点 Cp_h 可以被一个标识符 Mp_1 的Mashup所重用的条件是,当且仅当 Cp_h 表示的子树精确匹配 Mp_1 中的一个子树.因为如果它们精确匹配,那么 Mp_1 的一个子树和 Cp_h 表示的子树不仅算子的结构相同,而且算子的参数也相同.由于Mashup平台托管了大量的Mashup,所以需要一种高效的解决方案,使得Mashup平台能够快速发现新建Mashup匹配的缓存点.为此,POMO定义了B+树的Mashup索引表示,并且利用索引匹配实现了上述目标,该过程会在下一节具体介绍.当POMO发现 Mp_1 的一个或者多个子树精确匹配缓存点时,POMO会对 Mp_1 进行如下变换:把这些子树通过import算子进行替换,这些import算子引用缓存点的数据.例如,如果 Mp_1 的某个子树 St_q 匹配缓存点 Cp_h,那么 St_q 通过一个引用到 Cp_h 的数据的import算子进行替换;然后,这个被变换的Mashup发送到Mashup平台去执行.图2展示了一个新建的Mashup是如何通过变换来重用缓存点的过程.下面,我们重点介绍Mashup索引表示以及索引匹配的过程.

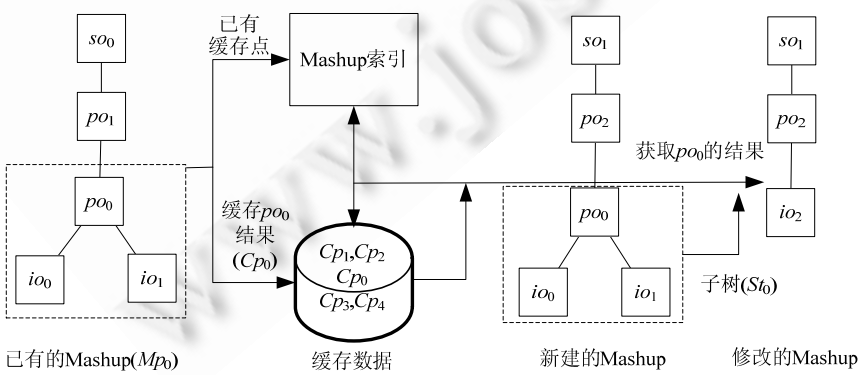


Fig.2 Rationale for caching point reusing

图2 缓存点重用原理

我们使用B+树对缓存点建立索引,这些索引存储在图1所示POMO的数据库中.由于OpSet中的每个算子

都有一个唯一标示符,因此,一个 Mashup 可以通过其算子的唯一标示符来表示.不同于其他算子,join 算子连接两个组件的输入.一个 join 算子的表示首先是一个特殊符号 SU,表示开始一个连接块;接着是第 1 个组件;接着是一个特殊字符 MU,出现在两个连接的组件中间;接着是第 2 个组件;最后是一个特殊符号 EU,表示连接块结束.图 3 给出了这样一个 Mashup 的字符串表示的例子,其中的破折号不是 Mashup 字符串表示的一部分,只是属性之间的分隔符.

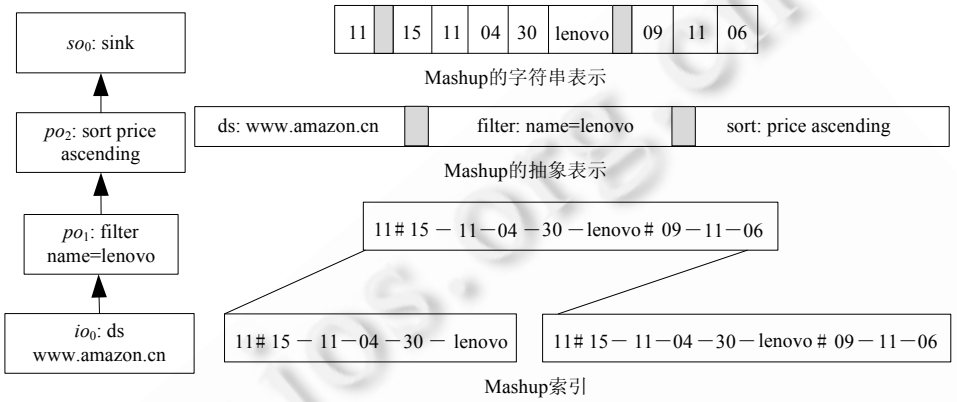


Fig.3 B+ tree index for Mashup

图 3 B+树的 Mashup 索引

B+树中每个节点的索引内容是 Mashup 字符串中的子串,它们基于字母顺序插入到索引中.例如图 3 所示的 Mashup,它首先从数据服务中获取“www.amazon.cn”提供的数据;接着,通过 name=“lenovo”过滤数据;然后,根据“price”进行排序.如果我们决定缓存 filter 算子执行后的数据,那么“11#15–11–04–30–lenovo”将被插入到索引中;然而,如果我们决定缓存整个 Mashup 执行后的数据,那么“11#15–11–04–30–lenovo#09–11–06”将被插入到索引中.图 3 表示了选取上述两个点作为缓存点而建立的 Mashup 索引,索引中的数字表示 Mashup 组成元素的 ID.例如,filter 建立的索引 15–11–04–30–lenovo 的解释如下:“15”是 filter 的 ID,“11”是数据服务的 ID,“04”表示数据服务的属性 name,“30”是 equality 算子的 ID,“lenovo”是属性“04”过滤的数值.在上述 Mashup 字符串表示中,#作为一种特殊字符,表示算子之间的分隔符.由于在每个算子执行后进行缓存的索引内容都包含了其前驱算子执行后进行缓存的索引内容,因此我们能够对 Mashup 建立索引而不必担心丢失 Mashup 中算子的执行顺序.

3.1 基于值域的近似匹配

上面介绍的 B+树索引主要用来查找精确匹配的缓存点,然而,如果 B+树索引只支持精确匹配,那么可能会失去很多重用缓存点的机会.为了解决这个问题,POMO 通过支持两种近似匹配的方式改进 B+树索引,也即基于值域的近似匹配和基于语义的近似匹配.采用近似匹配主要基于这样一个事实:B+树的叶子节点的关键词是有序的,因此,字母顺序相近的缓存点被存储在相同或者邻近的索引节点上.

假设一个新建 Mashup 的参数 p 的值域是$[x,y]$,如果只支持精确匹配,当缓存点的索引表示中参数 p 的值域不是$[x,y]$时,新建 Mashup 由于无法重用缓存点而不得不从头开始执行,特别是在缓存点的索引表示和新建的 Mashup 的索引表示除了参数 p 的值域$[a,b]\neq[x,y]$外,其他都相同的情况.由此产生的问题是,这个缓存点还能被新建 Mashup 重用么?为此,我们需要考虑以下 3 种情况:

- 第 1 种情况是 $x\geqslant a$ 且 $y\leqslant b$,那么这意味着值域$[x,y]$完全包含在值域$[a,b]$中,在这种情况下,缓存点被命中,POMO 利用过滤算子在缓存点的数据中找出新建 Mashup 所需要的数据.
- 第 2 种情况是 $x\geqslant a$ 且 $y>b$ 或者 $x<a$ 且 $y\leqslant b$ 或者 $x<a$ 且 $y>b$,那么值域$[x,y]$的部分数据包含在值域$[a,b]$中.如果只支持精确匹配,那么缓存点不会被命中.值得注意的是,这种局部包含关系在下面两种情况下是有用的:第 1 种情况是一部分数据能够从缓存点的数据中找到,其余数据能够从其他缓存点的数据

中找到,那么通过对这两部分数据利用过滤算子后再进行合并就能得到全部数据;第 2 种情况是当数据服务无法通过网络进行正常的数据传输时,新建 Mashup 无法执行,因此重用缓存点的数据可以提供给最终用户有效但是不完整的结果.例如,当 $x \geqslant a$ 且 $y>b$ 时,缓存点中值域$[x,b]$的数据能够提供给最终用户.尽管这些数据不完整,但是它们对最终用户仍然可能有用.

- 第 3 种情况是 $x<a$ 且 $y<a$ 或者 $x>b$ 且 $y>b$,那么值域$[x,y]$全部不在值域$[a,b]$中,缓存点没有被命中,因此,新建 Mashup 不得不重新开始执行.

上述分析只针对非统计算子,如果 Mashup 包含统计算子,例如求和算子,那么基于值域的近似匹配无法获得最终用户希望看到的统计结果.因此在这种情况下,缓存点只支持精确匹配.

下面通过一个例子说明 POMO 如何利用 B+树本身具有的快速查找能力实现缓存点重用.

假设一个 Mashup 首先通过数据服务从 www.amazon.cn 中获取数据,然后根据“price<5000”过滤其中的数据,它的执行结果被缓存.这个缓存点的索引表示成“11#15–06–32–5000”,其中最后 4 位对应 price 过滤的数值.此时,最终用户新建了一个 Mashup,它也通过数据服务从“www.amazon.cn”获取数据,然后根据“price<4000”过滤数据,这个 Mashup 的索引表示成“11#15–06–32–4000”.这个例子对应上述第 1 种情况,其中,新建 Mashup 的 price 值域完全包含在缓存点的 price 值域内.我们发现,这两个字符串表示具有一个公共部分“11#15–06–32”.如果只采用精确匹配,则从缓存点中搜索“11#15–06–32–4000”的结果将是空;如果采用近似匹配,则从缓存点中搜索“11#15–06–32–4000”的过程将是在 B+树中基于关键词的字母表顺序遍历索引的过程,最后会到达关键词“11#15–06–32–5000”,并且返回该关键词对应的缓存点.不同于返回空,近似匹配返回的字符串表示和新建的 Mashup 的字符串表示除了过滤的数值不同外其他都相同,因此 m 新建 Mashup 不需要重头开始执行,而只需在缓存点的数据中删除价格在 4 000~5 000 的数据项即可得到最终结果.这种 B+树查找方式提高了缓存点重用的效率.如果把“price<4000”改为“price<6000”,那就对应上述第 2 种情况,此时,新建 Mashup 的部分结果包含在缓存点的数据中.如果采用精确匹配,则将返回空,新建 Mashup 需要重头开始执行;但是如果由于网络故障导致新建 Mashup 执行出现问题,则会返回缓存点的数据,只提供给最终用户“price<5000”的结果.有人或许认为这种方式没有返回完整的数据,尽管这是事实,但在很多情况下,部分数据也能满足最终用户的需求.例如,最终用户很可能在缓存点提供的不完整的数据中发现了自己想要的结果.

3.2 基于语义的近似匹配

上面的例子说明了基于值域的近似匹配能够很好地支持数值型参数,然而在大多数情况下,Mashup 的参数是非数值类型,它们的内容很可能是具有语义关系的关键词.例如一个 Mashup 通过数据服务从“sport.sina.com”获取所有体育运动商店的商品,它的执行结果被缓存.假设一个最终用户新建了一个 Mashup,它通过数据服务从“sport.sina.com”获取所有网球商店的商品.从语义上说,这个新建 Mashup 的执行结果是缓存点的数据的子集.然而,由于此时的参数是关键词而不是数值,所以基于值域的近似匹配无法适用这种情况.

为此,我们给出了一种基于语义的近似匹配.其核心思想是,构建一个具有语义的关键词分类树,能够定义不同关键词的关系.这个假设前提是在一个关键词分类树中,上层的关键词的语义包含所有在其下的关键词的语义.通过利用关键词分类树所提供的语义信息来提高缓存点重用的可能性.当新建一个 Mashup 时,近似匹配会首先查找包含关键词的祖先节点的所有缓存点.例如,一个 Mashup 根据非数值的关键词参数 $p=X$ 进行过滤,假设这个 Mashup 已经被缓存,此时,一个新建 Mashup 和上述 Mashup 除了 $p=Y$ 以外其他都相同,如果采用精确匹配,则会返回空,新建 Mashup 不得不重新开始执行;然而,如果提供了上述包含语义的关键词分类树,近似匹配可以发现 X 和 Y 之间的关系,如果 X 是 Y 的祖先节点,那么新建 Mashup 的结果就是缓存点的数据的子集.尽管如此,缓存点的数据也无法直接被新建 Mashup 所重用,它需要在本地进一步过滤出新建 Mashup 的结果.例如,一个 Mashup 通过数据服务从“sport.sina.com”获取的数据,然后根据条件“category=体育”过滤数据,它的执行结果被缓存.这个缓存点的索引表示为“09#15–09–02–32–体育”,其中,“体育”是过滤数据的取值.此时,一个最终用户新建了一个 Mashup,这个 Mashup 通过数据服务从“sport.sina.com”获取数据,然后根据条件“category=网球”过滤数据.这个新建 Mashup 的索引表示为“09#15–09–02–32-网球”,其中,“网球”是过滤数据的取值.

从缓存点中查找“09#15–09–02–32–网球”的过程将是在 B+树中基于关键词的字母表顺序遍历索引的过程,直到到达关键词“09#15–09–02–32–体育”.如果使用精确匹配来查找“09#15–09–02-32-网球”,则将返回空;取而代之,近似匹配会发现新建 Mashup 的索引表示和缓存点的索引表示具有公共部分“09#15–09–02–32”,因此它会分别从缓存点和新建 Mashup 中抽取出对应参数的关键词,然后使用关键词分类树查找它们之间的关系.由于在关键词分类树中,“网球”是“体育”的儿子节点,所以近似匹配认为新建 Mashup 的执行结果包含在缓存点的数据中.最后,通过对缓存点的数据进行过滤得到新建 Mashup 的执行结果.图 4 描述了上述过程.

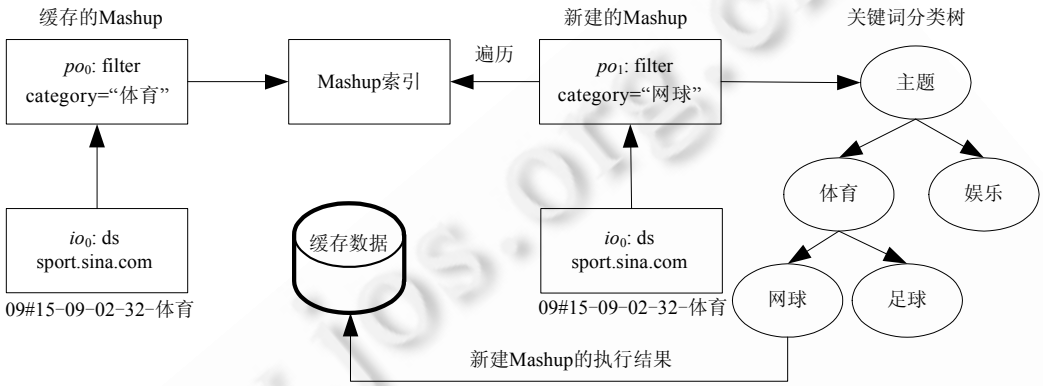


Fig.4 Approximate matching based on semantic taxonomy

图 4 基于语义分类的近似匹配

4 缓存点更新

如前所述,当数据源发生更新时,为了维护缓存点的数据一致性,缓存点需要更新.其中,import 算子会调用数据服务,数据服务通过网络传输数据.为了减少传输时间,本文借鉴了分布式文件系统领域的数据分块技术[15],把数据服务的输出数据按照一定规则分割成一些数据块,对每一个块都计算哈希签名,从而使得数据服务只需传输缓存点缺少的部分数据,这种思想也称为增量数据传输.通过增量数据传输减少传输时间有一个基本假设,就是数据具有很高的重合度,甚至是完全一致的.然而在许多应用场景下,这个假设并不成立.有一些数据服务在多次不同的调用中,由于时间的变化,输出数据的重合度很低,再加上哈希签名计算和校验消耗的时间,导致传输时间并没有减少.另一方面,当数据服务输出的数据量很小时,数据传输可以在很短的时间内完成,如果进行额外的哈希签名或者增量识别计算,反而增加了时间上的开销.基于上述原因,我们给出两阶段切换数据传输协议,该协议通过比较增量传输的成本和收益来选择增量传输或者全量传输.这里,数据服务端表示为 PS,服务端表示为 IS,两者之间通过网络进行数据传输.协议的发起方是 PS 中触发缓存内容更新的触发器,当数据服务发生更新时,触发器会执行缓存点更新的两阶段切换数据传输协议,以此保证缓存的数据一致性.协议的前置条件是 PS 已经调用了数据服务,得到了输出的数据,协议的执行结果是将数据服务输出的数据全部传输到客户端,并且尽可能地减少传输时间.设 PS 调用数据服务后得到的数据为 d,d 的数据量大小表示为 $size(d)$.整个协议的执行过程分为 i 阶段和 ii 阶段:

在 i 阶段,PS 启动两个线程 T_1 和 T_2,其中,T_1 的任务是计算增量数据;T_2 的任务是将数据全量传输到 IS,即,T_2 预期传输的数据量大小为 $size(d)$.T_1 计算 D 分块后的每一块数据的 SHA-1 签名值,并将签名值集合以及块序号发送到 IS,IS 根据收到的签名值集合识别出增量数据块序号,发送给 T_1.所谓增量数据块,就是 PS 端有而 IS 端没有的数据块.T_1 根据收到的增量数据块序号再计算得到增量数据块的总大小 $size(\Delta d)$.T_1 和 T_2 是并行执行的线程,当出现以下事件时,i 阶段结束,协议进入 ii 阶段:

(1) T_2 先于 T_1 完成任务.这种情况一般是由于 $size(d)$较小,因此,即使是全量数据传输也可以在较短的时间内完成.此时,IS 已经得到了 PS 输出的全部数据,因此,增量数据识别的计算就没有必要了,故终止 T_1.

(2) T_1先于T_2完成任务.这种情况一般由于$size(d)$较大,因此全量数据传输需要一定的时间.而此时T_1已经完成任务,即,T_1已经求得了增量数据的块序号集合和增量数据的数据量大小$size(\Delta d)$.此时须考虑两种情况:设此时在第1阶段已经传输到IS端的数据量大小为$size(copied(d))$,如果$size(\Delta d)\geqslant size(d)-size(copied(d))$,则应该继续全量传输,因为继续全量传输的时间开销小于采用增量传输的时间开销,继续把剩余的大小为$size(d)-size(copied(d))$的数据传输到IS端;如果$size(\Delta d)\leqslant size(d)-size(copied(d))$,则应该终止线程$T_2$,即,停止全量数据传输,而只需要把大小为$size(\Delta d)$的增量数据传输到IS端.

T_1和T_2同时完成任务,这种情况与情况(1)类似,但两个线程已经停止,因此不需要额外操作.

在如图5所示的ii阶段,无论是通过上述哪一种情况到达,最终都还需要执行一次缓存更新,即,把收到的数据d进行分块后以键值对的方式插入到缓存中,其中,键为数据块的SHA-1签名值,值为数据块的内容.整个协议的执行过程也可以用图5来表示.

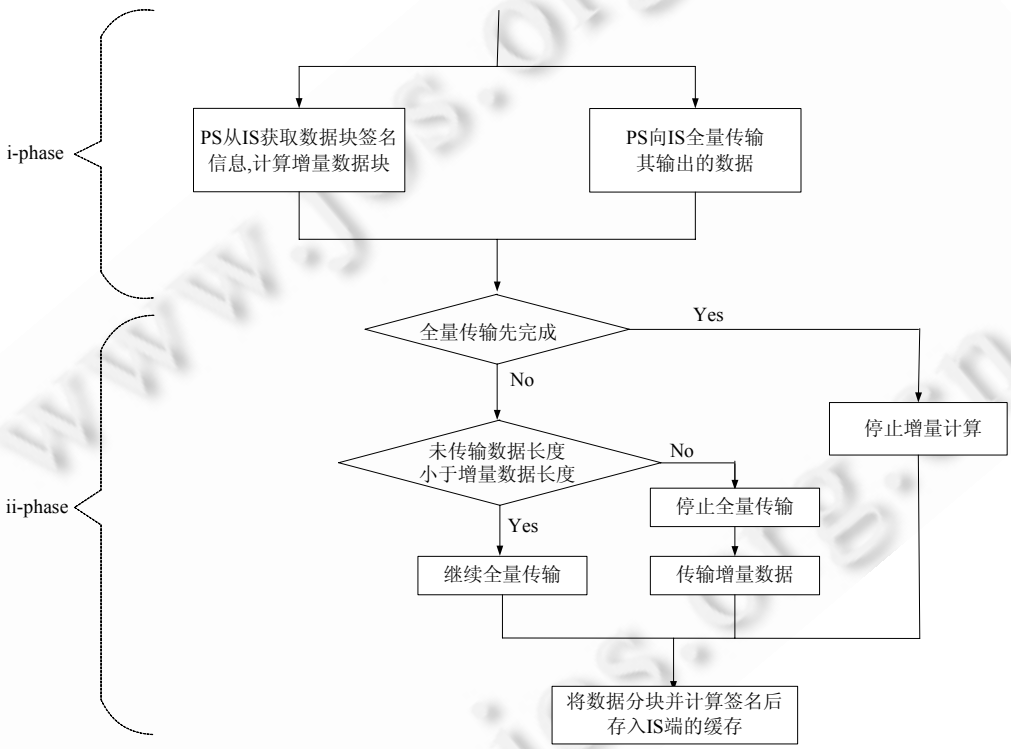


Fig.5 Two-Phase alternation data transmission protocol

图5 两阶段切换数据传输协议

5 实验分析

我们的实验包含以下4个目标:(1) 测试动态缓存点选取对Mashup性能的影响;(2) 测试B+树索引对Mashup性能的影响;(3) 测试近似匹配对缓存点重用的影响;(4) 测试两阶段切换数据传输协议对缓存点更新的影响.

5.1 实验配置

我们在植入POMO的Mashroom平台上进行实验,其中采用从syndic8种子库[15]中抽取出来的80个数据源,它们的大小在100KB~10MB之间.syndic8种子库是一个RSS和Atom种子的目录.最终用户通过使用Mashroom提供的算子创建Mashup,Mashroom平台执行这些Mashup,并且把执行结果返回给最终用户.我们模

拟 YahooPipes 构建了 500 个 Mashup,这些 Mashup 通过数据服务读取这 80 个数据源提供的数据.每个算子的成本函数通过执行这些的 Mashup 来估计,每个种子的更新频率通过数据服务的更新频率来模拟.为了构建关键词分类树,我们利用 Google 的关键词检索工具,该工具可以对检索的关键词进行归类,并且当检索一个关键词时,它会把关键词的类别和关键词列表下载到一个 CSV 文件,以后可以考虑利用 Wiki 或者本体构建关键词分类树.

5.2 缓存点选取实验

在第 1 组实验中,我们定量分析动态缓存点选取对 Mashup 性能的影响.动态缓存方式与其他 3 种方式进行比较:只缓存 Mashup 最后执行结果的末端缓存方式、不设置任何缓存的无缓存方式以及文献[12]提出的共享公共算子序列的方式.本文作者也参与了文献[12]的工作,并且 POMO 在 Mashup 拓扑结构合并过程中也应用了其中的成果.然而,针对 Mashup 的请求,公共算子序列至少需要执行 1 次重新计算;与之不同的是,公共算子序列可以看作缓存点,针对 Mashup 的请求,POMO 只需要返回缓存点的结果,不需要重新计算,但是缓存点在更新时需要重新计算.由于 POMO 是根据成本和收益的分析动态选取缓存点,因此,即使当发现 Mashup 之间存在公共算子序列可以减少重复计算,但是如果缓存点成本更高,那么也不会缓存公共算子序列的计算结果.从理论上说,POMO 至少等于文献[12]提出的方法.实验比较了满足最终用户不同请求频率的 4 种方式的 Mashup 的总成本,也即是总的计算次数.

在第 1 个实验中,我们比较 Mashup 的平均请求频率从 20 次/秒递增到 100 次/秒的总成本.此时,Mashup 的总个数设置为 500,因此 Mashup 的总请求频率从 10 000 次/秒递增到 50 000 次/秒.数据服务的平均更新频率设置为 60 次/秒.该实验假设有足够的存储空间.图 6 显示了单位时间的总成本.如图 6 所示,随着请求频率增加,末端缓存方式的总成本基本保持不变,这是因为在更新频率不变的情况下,末端的缓存点更新的计算次数不变,无论请求频率如何变化,Mashup 平台只需要直接返回末端的缓存点而不需要重新计算,因此总的重新计算次数保持不变,也即是总成本不变.在末端缓存方式中,其成本主要来自缓存点更新的计算次数.当请求频率较低时,无缓存方式的总成本和 POMO 的总成本差不多;然而随着请求频率的增加,无缓存方式的总成本快速增长.尽管 POMO 的总成本随着请求频率的增加而增长,但是当接近末端缓存方式的总成本时,其成本增长曲线逐渐变得平缓.值得注意的是,POMO 的总成本始终少于共享公共算子序列的总成本.

在第 2 个实验中,我们研究更新频率对 4 种方式的影响.该实验配置与上一个实验配置基本相同,其中,Mashup 的平均请求频率设置为 60 次/秒,数据服务的更新频率从 20 次/秒递增到 100 次/秒.如图 7 所示,实验结果再一次说明 POMO 好于其他 3 种方式.然而在这个实验中,无缓存方式的总成本保持不变.这是因为在请求频率不变的情况下,获得 Mashup 最后执行结果的重新计算次数不变,无论更新频率如何变化,由于没有缓存点,Mashup 平台不会产生缓存点更新的重新计算,因此总的重新计算次数保持不变,也即是总成本不变.值得注意的是,POMO 的总成本始终少于共享公共算子序列的总成本,这也证实了之前的推测.

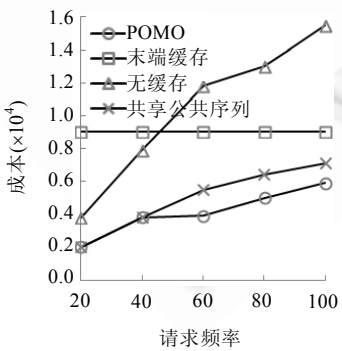


Fig.6 Total cost with variable request frequency

图 6 请求频率改变时的总成本

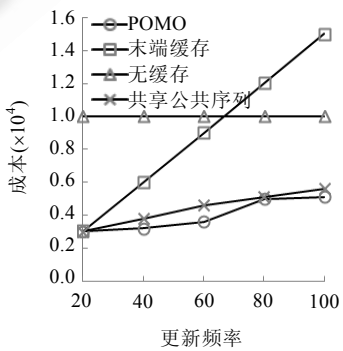


Fig.7 Total cost with variable update frequency

图 7 更新频率改变时的总成本

POMO可以获得上述较好结果的原因是因为它能够通过向上或者向下移动缓存点来适应动态变化的更新频率和请求频率.图 8 通过计算当更新频率从 20 次/秒递增到 100 次/秒时缓存点的平均层次来验证这个结论.其中,Mashup 的平均请求频率设置为 60 次/秒.如图 8 所示,随着更新频率的增加,为了减少缓存点更新计算,POMO 逐渐选取 Mashup 树中较低层次的节点作为缓存点,而末端缓存方式选取的缓存点则一直在同一层次.

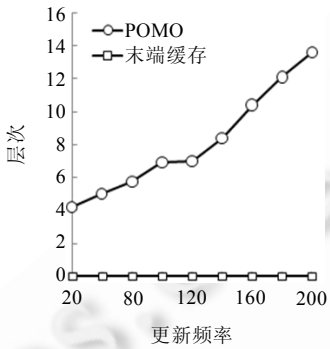


Fig.8 Caching point level with variable update frequency

图 8 更新频率改变时缓存点的层次

5.3 缓存点查找实验

在第 2 组实验中,我们通过计算访问 B+树索引对应的缓存点的平均延迟来研究 B+树索引对 Mashup 性能的影响.在第 1 个实验中,我们研究请求频率对索引查找时间的影响.该实验使用了 500 个 Mashup,数据服务的更新频率设置为 60 次/秒.

如图 9 所示,随着请求频率的增加,索引查找时间逐渐减少.这是由于以下两个方面的原因:第一,随着请求频率的增加,POMO 趋向选择靠近 Mashup 树的根节点的节点作为缓存点.当靠近根节点时,Mashup 树的宽度开始收缩,并且在索引中的缓存点的个数也开始减少;第二,POMO 是从新建 Mashup 的根节点开始往下查找匹配的缓存点,当请求频率较高时,缓存点靠近根节点,因此查找匹配的缓存点的速度更快.实验结果也显示了 B+树索引的一个重要功能:当请求频率较高时,它通过快速响应来提高 Mashup 平台的性能.

在第 2 个实验中,我们研究 Mashup 的深度对索引查找时间的影响.该实验还是使用 500 个 Mashup,这些 Mashup 的平均请求频率和平均更新频率都设置为 60.图 10 显示了当 Mashup 的深度从 5 增加到 20 时的索引查找时间.在初始阶段,索引查找时间随着 Mashup 深度的增加而呈线性增长,其原因是,随着 Mashup 深度的增加,从 Mashup 树的较低层次选取缓存点的可能性逐渐增高,因此查找匹配缓存点的时间逐渐增多.然而,当 Mashup 的深度到达 15 时,索引查找时间开始下降.

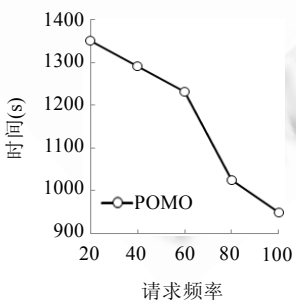


Fig.9 Indexing time with variable request frequency

图 9 不同请求频率的索引查找时间

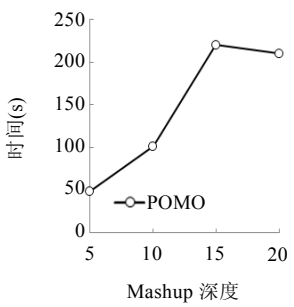


Fig.10 Indexing time with variable Mashup depth

图 10 不同 Mashup 深度的索引查找时间

5.4 缓存点重用实验

在这组实验中,我们研究在 B+树索引中采用近似匹配对缓存点重用的影响.本实验仍然采用 500 个 Mashup,数据服务的更新频率被设置为 60.图 11 显示了使用基于值域的近似匹配能够减少 Mashup 大约 9%的总成本,这是因为近似匹配增加了 Mashup 中缓存点重用的可能性.如图 12 所示,当索引不支持基于值域的近似匹配时,索引查找的结果可能是空;然而,当索引支持基于值域的近似匹配时,索引查找的命中率显著增加.在这个实验中,对于参数是数值型的 filter 算子,我们从 1 000~5 000 的范围内随机选取一个数值作为其过滤条件.

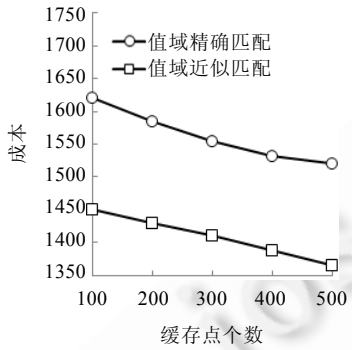


Fig.11 Total cost with range matching

图 11 值域近似匹配的总成本

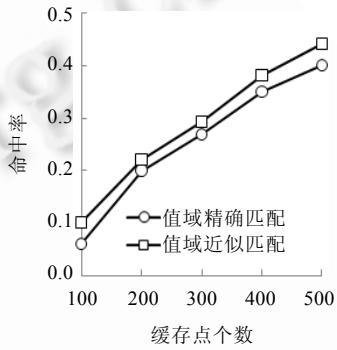


Fig.12 Hit ratio with range matching

图 12 值域近似匹配的命中率

基于语义的近似匹配的效果显示在图 13 中.在这个实验中,当新建一个 Mashup 时,对于参数取值是关键词的 filter 算子,一个随机的关键词或者类别会弹出.图 3 中显示了当使用基于语义的近似匹配后,Mashup 的总成本开始减少.这是缓存点命中率不断增加的结果,这在图 14 中得到了证明.

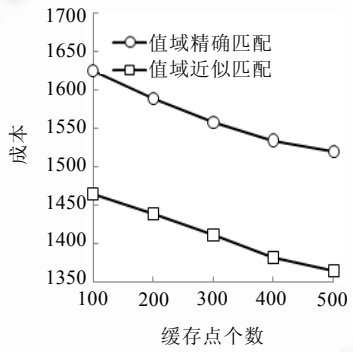


Fig.13 Total cost with semantic matching

图 13 语义近似匹配的总成本

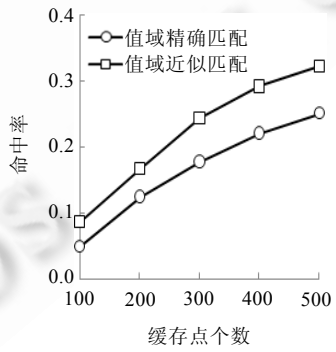


Fig.14 Hit ratio with semantic matching

图 14 语义近似匹配的命中率

虽然实验证明了近似匹配的效果,但是这种性能上的提高有以下两个方面的局限性:

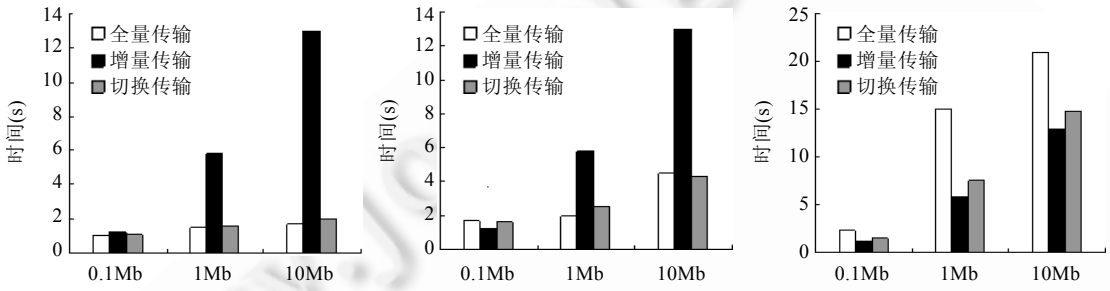
- 首先,近似匹配只能支持部分算子,例如,基于值域的近似匹配只能支持 fliter,truncate 和 tail 等算子而不能支持 import,nest,unique 等算子.
- 其次,由于最终用户能够选取的关键词的个数以及在算子中能够使用的数值个数很多,所以发现近似匹配所支持的算子的概率会很低.为此,POMO 可以通过在近似匹配过程中考虑最终用户选取关键词或者选取数值的使用模式来做进一步改进.例如:如果大部分最终用户对价格低于 5 000 的电脑感兴趣,那么这暗示着大部分最终用户创建的包含 filter 算子的 Mashup 很可能包含“price<5000”;同理,如果大部分最终用户关心体育,那么大部分最终用户创建的 Mashup 很可能使用与体育相关的关键词.因

此,在 Mashup 中考虑最终用户的使用模式,可以为缓存点重用带来很多帮助.

5.5 缓存点更新实验

我们基于 syndic8 种子库提供的数据源,有针对性地构造了一个数据服务集合,它们按照输出数据的特征分为 A,B,C 这 3 类:A 类服务的特征多次调用的输出数据的重合度非常高,这类服务针对更新频率低的数据源的封装;B 类服务的特征是多次调用的输出数据有一定的重合度,但重合度的高低与时间因素有一定的关系;C 类服务的特征是多次调用的输出数据的重合度非常低,这类服务针对更新频率非常高的数据源的封装.

下面将通过实验比较对于 A,B,C 这 3 类服务,分别采用全量传输方式、增量传输方式和两阶段切换数据传输方式来实现数据传输所耗费的时间.每类服务中有 3 个输出数据长度为不同数量级(0.1MB,1MB,10MB)的数据服务.最终的测试结果如图 15 所示.



(a) A 类服务的测试结果 (b) B 类服务的测试结果 (c) C 类服务的测试结果

Fig.15 Data transmission integrated test

图 15 数据传输的综合测试

从图 15 中可以看到:

- 对于 A 类服务,采用增量传输或者两阶段切换数据传输的优化效果十分明显,1Mb 数据传输时间分别减少了 72.1%和 71.9%,10Mb 数据传输时间分别减少了 86.1%和 84.6%.其原因是,在测试中,A 类服务总是返回固定的输出数据,所以在多次调用的过程中,服务提供者只需要花费时间在分块计算和传输数据签名上.
- 对于 B 类服务,采用增量传输或者两阶段切换数据传输相对于全量传输仍然有可观的优化效果,1Mb 数据传输时间分别减少了 66.1%和 57.6%,10Mb 数据传输时间分别减少了 65.3%和 67.2%.其原因是,在测试中,B 类服务的多次输出虽然不是固定的,但是有一定的重合度,所以相对于全量传输,其数据的传输量还是要少一些.与处理 A 类服务的输出数据相比,主要是多花了一部分时间在传输增量数据块上.
- 对于 C 类服务,全量传输的延时最短.这是因为在测试中,C 类服务每次被调用输出的数据基本上是随机的.这就导致每次增量计算识别出的增量数据块的大小也很多,再加上增量计算本身需要耗费的时间,所以增量传输的效率反而低于全量传输.两阶段切换数据传输的时间小于增量传输的时间,1Mb 数据传输时间减少了 52.1%,10Mb 数据传输时间减少了 31.8%.主要是因为其在第 1 阶段采用了增量识别和全量传输同时进行的方式,在第 2 阶段会根据第 1 阶段的处理结果来选择是否启用增量传输,较好地适应数据重合度低的情况;另外一方面,两阶段切换数据传输也会花费部分时间在线程的切换上,所以在 C 类服务的测试中比全量传输的时间稍多,1Mb 数据传输时间增多了 25.2%,10Mb 数据传输时间增多了 15.3%.

总的来说,在针对各种类型的服务的测试中,两阶段切换数据传输均取得了较低的传输时间.

6 相关工作

目前的 Mashup 可以分为 3 种操作方式:

- 第 1 种是可视化编程的操作方式,相关的 Mashup 平台包括 PAIRSE[16],AquaLogic[17]等.这些平台为最终用户提供一个可视化的操作环境,通过接受用户在该环境中的各种操作(包含对可视化控件的拖拽以及键盘输入),生成操作数据资源的脚本语言.
- 第 2 种是数据流编程的操作方式,相关的 Mashup 平台包括 Yahoo Pipes,IBM Damia 等.这些平台支持最终用户在图形界面上构造数据流,该数据流以数据服务为数据源,中间经过一系列的数据操作,最后向最终用户返回结果.
- 第 3 种是 Spreadsheet 编程的操作方式,相关的 Mashup 平台包括 SheetMusiq[18],SpreadATOR[19],Mashroom 等.这些平台为最终用户提供一个类似于 Excel 电子表格的操作界面,在这个界面里,数据资源以表格形式呈现,最终用户直接在表格上对数据进行增删改等操作,其中,复杂的数据操作被分割为若干可组合、可重复使用的小操作,最终用户进行每步操作之后,能够立即观察到该操作带来的数据变化.

然而,上述所有的 Mashup 平台主要关注 Mashup 的操作方式,没有考虑 Mashup 的性能问题.

在 Web 应用中,大量的研究工作关注 Web 内容缓存,其中的研究点包括缓存粒度、缓存布局、缓存替换和缓存更新,相应的各种缓存技术也被提出[20].而在 Web 服务中,缓存也能带来性能的提高,例如,Li 提出的 SigsitAccelerator[9]中间件能够通过缓存减少服务调用过程中的数据传输时间.SigsitAccelerator 中间件的本质是在 SOAP 通信的两端通过缓存来减少网络上的通信开销,其中,通过使用加密哈希来发现发送的数据与已经缓存的数据的相似性,并且只发送加密哈希后的结果而不是原始数据.然而,该方法没有考虑如何选取缓存点.与该方法相比,本文提出的方法重点考虑如何根据 Mashup 的拓扑结构选择缓存点.IBM 提出的中介缓存模式(cache mediation pattern)[21]利用 ESB 解耦服务之间的调用关系,并且可以判断服务调用前是否有缓存数据可用.Tatemura 提出的中间件架构 Wrex 可以对 Web 服务响应进行缓存[8].此外,Papageorgiou 等人讨论了针对移动客户端的 Web 服务的缓存机制[10,22].然而,上述 Web 服务的缓存技术都只能在固定点进行缓存,无法利用 Mashup 的拓扑结构动态选取缓存点.为此,Zhang 等人提出了一种动态 Mashup 的缓存机制[23],该机制通过动态选取缓存点来提高 Mashup 的运行效率,但是忽略了缓存点更新.针对以数据服务作为数据源的 Mashup,我们的前期工作提出了类似的缓存点选取方法[13],但是该方法假定影响缓存点选取的参数是静态的,因此无法支持动态缓存点选取.

在数据仓库中,物化视图可以看成一种缓存形式,物化视图的优点在于:直接访问物化视图比重新计算视图更快,并且通过在物化视图上构建索引进一步提高查询性能.物化视图的维护主要通过增量更新和重新计算[24]两种方式.增量更新的基本思想是,根据视图定义生成一段增量更新程序,该程序根据关系的变化以及当前关系和视图的值生成更新元组.文献[25]对数据服务上的嵌套关系视图的增量更新进行了深入研究,此外,当前的研究人员更加关注半结构化数据(包括 XML 视图)上的增量更新[26,27].虽然本文的缓存点更新采用的是重新计算方式,但是在获取数据源过程中考虑了增量传输和全量传输相结合的方式,减少了在重新计算之前的传输时间.下一步可以考虑增量更新的计算方式.

7 结束语

由于 Mashup 具有高度个性化的特征,传统的 Web 服务缓存技术不适合应用到 Mashup 上.为此,本文提出了一种 Mashup 运行时的性能优化方法——POMO.POMO 建立在 Mashup 树结构的基础上,其中,树中每个节点表示 Mashup 的一个算子.POMO 的目标是提高 Mashup 性能,其中,通过建立算子序列的缓存点的成本和收益模型实现了动态缓存点选取;通过 B+树索引表示实现了缓存点重用,同时,B+树索引还支持近似匹配,进一步提高了缓存点命中率;通过两阶段切换数据传输,减少缓存点更新的传输时间.相关实验结果表明,POMO 实现了上

述目标.

References:

[1] Papazoglou MP, Traverso P, Dustdar S, Leymann F. Service-Oriented computing: State of the art and research challenges. In: Proc. of the IEEE Computer. New York: IEEE Computer Society, 2007. 38−45. [doi: 10.1109/MC.2007.400]

[2] Chang KC, He B, Li C, Patel M, Zhang Z. Structured databases on the Web: Observations and Implications. SIGMOD Record, 2004,33(3):61−70. [doi: 10.1145/1031570.1031584]

[3] Wang GL, Yang SH, Han YB. Mashroom: End-user Mashup programming using nested tables. In: Proc. of the 18th Int'l Conf. on World Wide Web (WWW 2009). New York: ACM Computer Society, 2009. 861−870. [doi: 10.1145/1526709.1526825]

[4] Programmable Web. 2009. http://www.programmableweb.com

[5] Al-Masri E, Mahmoud QH. Investigating Web services on the World Wide Web. In: Proc. of the 17th Int'l Conf. on World Wide Web 2008. New York: ACM Computer Society, 2008. 795−804. [doi: 10.1145/1367497.1367605]

[6] Jones MC, Churchill EF, Twidale MB. Mashing up visual languages and Web mash-ups. In: Proc. of the IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC 2008). New York: IEEE Computer Society, 2008. 143−146. [doi: 10.1109/VLHCC.2008.4639075]

[7] Simmen D, Singh A. Damia: A data Mashup fabric for Intranet applications. In: Proc. of the 33rd Int'l Conf. on Very Large Databases (VLDB 2007). New York: ACM Computer Society, 2007. 1370−1373.

[8] Tatemura J, Po O, Sawires A, Agrawal D, Candan KS. Wrex: A scalable middleware architecture to enable XML caching for Web-services. In: Proc. of the Middleware. New York: Springer-Verlag, 2005. 124−143. [doi: 10.1007/11587552_7]

[9] Li WB, Tordsson J, Elmroth E. An aspect-oriented approach to consistency-preserving caching and compression of Web service response messages. In: Proc. of the 2010 IEEE Int'l Conf. on Web Services (ICWS 2010). New York: IEEE Computer Society, 2010. 526−533. [doi: 10.1109/ICWS. 2010.83]

[10] Papageorgiou A, Schatke M, Schulte S, Steinmetz R. Enhancing the caching of Web service responses on wireless clients. In: Proc. of the 2011 IEEE Int'l Conf. on Web Services (ICWS 2011). New York: IEEE Computer Society, 2011. 9−16. [doi: 10.1109/ICWS. 2011.52]

[11] Hassan OAH, Ramaswamy L, Miller JA. Enhancing scalability and performance of Mashups through merging and operator reordering. In: Proc. of the 2010 IEEE Int'l Conf. on Web Services (ICWS 2010). New York: IEEE Computer Society, 2010. 171−178. [doi: 10.1109/ICWS.2010.92]

[12] Lin HL, Zhang C, Zhang P. An optimization strategy for Mashups performance based on relational algebra. In: Proc. of the 14th Asia-Pacific Web Conf. (APWEB 2012). Berlin, Heidelberg: Springer-Verlag, 2012. 366−375. [doi: 10.1007/978-3-642-29253-8_31]

[13] Zhang P, Wang GL, Ji G, Liu C. Optimization update for data composition view based on data service. Chinese Journal of Computers, 2011,34(12):2344−2354 (in Chinese with English abstract).

[14] Han YB, Wang GL, Ji G, Zhang P. Situational data integration with data services and nested table. In: Proc. of the Service-Oriented Computing and Applications. Berlin, Heidelberg: Springer-Verlag, 2012. 1−22. [doi: 10.1007/s11761-012-0103-5]

[15] Kruus E, Ungureanu C, Dubnicki C. Bimodal content defined chunking for backup streams. In: Proc. of the 8th USENIX Conf. on File and Storage Technologies (USENIX 2010). Berkeley: USENIX, 2010. 18−18.

[16] Benslimane D, Barhamgi M, Cuppens F, Morvan F, Defude B, Nageba E. PAIRSE: A privacy-preserving service-oriented data integration system. ACM SIGMOD Record, 2013,42(3):42−47. [doi: 10.1145/2536669.2536677]

[17] Borkar V, Carey M, Koleth S, Kotopoulis A, Mehta K, Spiegel J, Thatte S, Westmann T. Graphical XQuery in the AquaLogic data services platform. In: Proc. of the ICDE. New York: IEEE Computer Society, 2010. 1069−1080. [doi: 10.1145/1807167.1807288]

[18] Liu B, Jagadish HV. A spreadsheet algebra for a direct data manipulation query interface. In: Proc. of the IEEE 25th Int'l Conf. on Data Engineering (ICDE 2009). New York: IEEE Computer Society, 2009. 417−428. [doi: 10.1109/ICDE.2009.34]

[19] Kongdenfha W, Benatallah B, Vayssi J, Saint-Paul RE, Casati F. Rapid development of spreadsheet-based Web Mashups. In: Proc. of the 18th Int'l Conf. on World Wide Web (WWW 2009). New York: IEEE Computer Society, 2009. 851−860. [doi: 10.1145/1526709.1526824]

[20] Guerrero C, Lera I, Juiz C. Performance improvement of Web caching in Web 2.0 via knowledge discovery. Journal of Systems and Software, 2013,86(12):2970−2980. [doi: 10.1016/j.jss.2013.04.060]

[21] Cache mediation pattern specification: An overview. 2006. http://www.ibm.com/developerworks/webservices/library/ws-soa-cachemed/

[22] Béjar R, Lopez-Pellicer FJ, Nogueras-Iso J, Zarazaga-Soria FJ. A protocol for machine-readable cache policies in OGC Web services: Application to the EuroGeoSource information system. Environmental Modelling & Software, 2014,60:346−356. [doi: 10.1016/j.envsoft.2014.06.026]

[23] Zhang P, Wang GL, Ji G, Han YB. An efficient data maintenance model for data service Mashup. In: Proc. of the 9th Int'l Conf. on Services Computing (SCC 2012). New York: IEEE Computer Society, 2012. 699−700. [doi: 10.1109/SCC.2012.10]

[24] Lin ZY, Yang DQ, Wang TJ, Song GJ. Research on materialized view selection. Ruan Jian Xue Bao/Journal of Software, 2009, 20(2):193−213 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3416.htm [doi: 10.3724/SP.J.1001.2009.03416]

[25] Zhang P, Han YB, Wang GL. Implementing dynamic nested view update based on data service. Chinese Journal of Computers, 2013,36(2):226−237 (in Chinese with English abstract).

[26] Liu ZH, Chang HJ, Sthanikam B. Efficient support of XQuery update facility in XML enabled RDBMS. In: Proc. of the IEEE 28th Int'l Conf. on Data Engineering (ICDE 2012). New York: IEEE Computer Society, 2012. 1394−1404. [doi: 10.1109/ICDE.2012.17]

[27] Bonifati A, Goodfellow M, Manolescu I, Sileo D. Algebraic incremental maintenance of XML views. ACM Trans. on Database Systems, 2013,38(3):1−45. [doi: 10.1145/2508020.2508021]

附中文参考文献:

[13] 张鹏,王桂玲,季光,刘晨.基于数据服务的数据服务组合视图的优化更新.计算机学报,2011,34(12):2344−2354.

[24] 林子雨,杨冬青,王腾蛟,宋国杰.实视图选择研究.软件学报,2009,20(2):193−213. http://www.jos.org.cn/1000-9825/3416.htm [doi: 10.3724/SP.J.1001.2009.03416]

[25] 张鹏,韩燕波,王桂玲.基于数据服务的嵌套视图动态更新方法.计算机学报,2013,36(2):226−237.

张鹏(1984－),男,安徽淮南人,博士,副研究员,CCF 高级会员,主要研究领域为服务计算,流数据处理,数据挖掘.

林海伦(1987－),女,博士生,助理研究员,主要研究领域为数据挖掘,知识处理.

刘庆云(1980－),男,博士,高级工程师,CCF 会员,主要研究领域为信息安全,云计算.

孙永(1976－),男,博士,高级工程师,CCF 会员,主要研究领域为数据挖掘,信息安全.

徐克付(1977－),男,博士,副研究员,CCF 会员,主要研究领域为云计算,流数据处理.

谭建龙(1974－),男,博士,研究员,主要研究领域为数据流处理,网络安全.