第10期　　电　子　学　报　　Vol.41　No.10
2013年10月　　ACTA ELECTRONICA SINICA　　Oct.　2013

基于查询优化策略的语义缓存谓词化简

李　东,陈　锐,徐　扬
(华南理工大学软件学院,广东广州 510006)

摘　要： 语义缓存技术可以有效地支持移动计算应用.现有的语义缓存查询处理算法在时空效率和裁剪的复杂度上较高,限制了语义缓存的实用性.为此,本文提出20条逻辑规则和语义缓存动态合并策略,来降低查询裁剪的复杂性.在Android系统上的实验表明,在简单查询下,采用全合并的缓存管理策略和谓词析取式优化算法相结合的方法,能较好地优化查询处理.在复杂查询方面,基于谓词复杂度的语义缓存动态合并策略能很好地平衡缓存与查询两端的谓词复杂度,有效地提高语义缓存的查询处理效率.

关键词： 语义缓存；查询优化；查询裁剪；谓词化简
中图分类号： TP301　**文献标识码：** A　**文章编号：** 0372-2112 (2013) 10-2021-08
电子学报 URL： http://www.ejournal.org.cn　**DOI：** 10.3969/j.issn.0372-2112.2013.10.024

Predicate Simplification Based Query Optimization Strategy for Semantic Caching

LI Dong, CHEN Rui, XU Yang
(*School of Software Engineering, South China University of Technology, Guangzhou, Guangdong* 510006, *China*)

Abstract： The semantic caching can efficiently support the applications in the context of mobile computing. Most existing algorithms for query trimming suffer high time and space complexity and they can't be used in small mobile devices. To this end, we propose 20 rules and a dynamic semantic merging strategy to simplify the complexity of query trimming, which is measured by the numbers of predicates that are depicted in the description for the semantic cache or queries. Some experiments on Android system show that disjunction simplification algorithm along with Completive Merging strategy can get the best performance for simple queries among the alternatives. For complex queries, the dynamic merging strategy based on predicate simplification can balance the complexity between caches and query processing and gain a good performance in a wide range.

Key words： semantic caching; query optimization; query trimming; predicate simplification

1　引言

移动计算环境提供了在任何时间、任何地点访问信息的能力,但是移动计算环境下设备移动性、低带宽、网络频繁断接性以及有限的计算资源限制了移动计算环境中的应用,语义缓存技术是解决这些问题的一条有效途径.

语义缓存技术是一种基于客户端的缓存技术,它缓存的内容包括从服务器端返回的查询结果以及所提交查询的语义描述.移动设备(或者客户端)可以通过查询的语义描述判断查询请求能否在本地满足还是需要发往服务器获取结果.通过语义缓存,移动客户端能够减少与服务器交互的数据量,因而减少与服务器的通信开销,提高响应效率.

语义缓存上的查询可以分成两类:探测查询(probe query)[1]和剩余查询(remainder query)[1].快速的查询裁剪和简洁的裁剪结果表达是提高语义缓存实用性的关键,换句话说,如果查询裁剪复杂度过高,将会很快耗尽移动设备有限的资源,降低其可用性.现有的语义缓存查询处理机制在时空效率和裁剪结果表达的复杂性这两方面存在很大的缺陷.这些查询处理算法都只是利用逻辑与运算和逻辑差运算裁剪出探测查询和剩余查询,这些算法的时空复杂度很高.为此,我们提出一系列优化规则和一种基于谓词复杂度的查询化简策略来解决这个问题.本文中,谓词复杂度由缓存语义中谓词的数目决定.基于谓词复杂度,我们可以对语义缓存的查询

收稿日期:2012-08-15;修回日期:2013-04-18;责任编辑:郭游
基金项目:国家自然科学基金重大项目(No.71090403);国家留学基金委留学回国人员科研启动基金项目(No.B7110020);广州市科技局应用基础研究项目(No.B2111420)

进行分类,分为简单查询和复杂查询.对于简单查询,提出了优化规则进行查询裁剪优化,大大减小了裁剪过程中谓词的数目,提高了裁剪效率.对于复杂查询,提出了基于谓词复杂度的语义缓存动态合并策略,能根据缓存与裁剪语句的谓词数目来决定缓存的合并方式,降低了查询处理时间快速激增的趋势.实验比较了简单查询和复杂查询下的查询处理效率,验证了本文提出的优化策略,实验结果有效地说明了不同类型查询使用不同优化机制的有效性.本文的主要贡献包括:(1)提出优化的20条逻辑规则和基于谓词复杂度的语义缓存动态合并策略可对简单查询和复杂查询进行查询裁剪优化.(2)通过在Android系统上的实验验证了优化规则和算法的有效性.

本文其余部分组织如下:第二节介绍该领域的研究概况和相关工作,第三节给出了相关的定义,语义缓存的查询处理过程,20条优化规则和语义缓存合并策略.第四节分析了优化算法的性能,第五节对本文进行总结.

2 相关工作

语义缓存技术的研究主要集中在语义缓存查询处理、语义缓存替换策略以及语义缓存合并策略等三个方面.

文献[1]中,Qun Ren和Margaret H. Dunham提出了移动计算环境下基于位置的语义缓存查询处理流程和语义缓存替换策略.文献[2]中,Parke Godfrey和Jarek Gryz扩展了语义包含(semantic containment)的概念,提出了语义重叠(semantic overlap)的概念,基于语义重叠提出了在语义缓存部分满足查询时将查询裁剪分割执行的思想.文献[3]中,Björn Tór Jónsson et al.采用更先进的服务器和更先进的硬件来对语义缓存查询处理的性能和负载进行重新评估.文献[4]提出了一种基于客户端缓存的语义缓存机制,提出了一种缓存合并策略并且讨论了不同情况下的查询处理流程.文献[5]提出了语义缓存的形式化定义并且详细讨论了语义缓存的查询处理策略.在文献[6]中,Ali-Asghar Safaeei et al.提出了一种可以处理复杂查询 (例如,group by) 的语义缓存框架,它使用查询图模型来实现这一目标.在此架构中缓存段的结构建立在查询图模型的基础上,文献也讨论了缓存管理算法和缓存替换算法.文献[7]提出了基于缓存区域(semantic region)的概念,并给出了一种新颖的语义缓存合并策略,并证明了在有限内存下该策略具有良好的效果.文献[8]研究了面向聚集查询的语义缓存形式化描述,在此基础上对查询匹配进行分类,最后给出了查询裁剪算法.文献[9]提出了一种基于谓词分类的语义缓存查询裁剪算法.文献[10]通过使用维诺图(voroni diagram)来对近邻查询进行索引,提高了查询处理的响应效率.文献[12]提出了一个跨平台的,可伸缩的移动设备资源处理架构,利用该架构缓存处理能够动态响应资源变化.

3 语义缓存查询处理优化

3.1 语义缓存与查询

语义缓存不同于传统的页缓存和元组缓存,它是基于客户端查询语义建立的一种缓存.语义缓存的内容是由历史查询结果以及相对应的查询语义描述构成的,在本文中,我们借用文献[1]和[3]中语义缓存的定义,并扩展如下:

定义 1 查询 $Q = < Q_R, Q_A, Q_P, Q_T >$.其中 Q_R 为查询对应的关系集合,Q_A 是查询对应的属性集合,Q_P 是查询对应的谓词集合,Q_T 是该查询被提交的时间.其中,R, A, P 的含义与文献[3]的描述相同,Q_P 是原子查询谓词 P[1],定义为{Attribute, Operation, Data},其中 Attribute 为关系中对应属性,Operation 取值{ =, <, >, ≤, ≥, like, Not like}, Data 取值{String, Integer, Float, Double, Boolean}

定义 2 谓词合取式 C_P: $< P >$,表示形式为 $C_P = P_1 \wedge P_2 \wedge P_3 \wedge \cdots P_{i-1} \wedge P_i \wedge P_{i+1} \cdots \wedge P_n$. P_i 为一个原子查询谓词.

定义 3 谓词析取式 D_P: $< C_P >$,表示形式为 $D_P = C_1 \vee C_2 \vee C_3 \vee \cdots C_{i-1} \vee C_i \vee C_{i+1} \cdots \vee C_n$. C_i 为谓词的合取式形式.

定义 4 简单查询 Q_S,满足 $Q_P = C_P$,且 $\forall\, i \in$ integer, P_i(Attribute) = P_{i+1}(Attribute), P_i(Operation)取值{ =, <, >, ≤, ≥ = }, P_i(Data)取值{Integer, Float, Double, String}.

定义 5 复杂查询 Q_C,满足 $Q_P = D_P$.

定义 6 移动语义缓存 S_{CM},表示为 $S_{CM} = < S_Q, S_L, S_C >$,其中 $S_Q = \{Q_S, Q_C\}$, S_L 为缓存的位置特征或属性,S_C 是缓存片段[3]的集合

定义 7 谓词复杂度(Predicate Complexity)为查询 Q(或者 S_{CM})包含的所有谓词数量的总和.

3.2 语义缓存查询处理算法

语义缓存查询处理的关键是进行查询裁剪,它可得到探测查询和剩余查询,探测查询可以在本地缓存得到结果,剩余查询需要发往服务器并由服务器端返回结果给客户端.查询处理的一般流程如图1所示.查询裁剪算法的细节可参见文献[11],优化/简化规则与合并策略将在后面讨论.

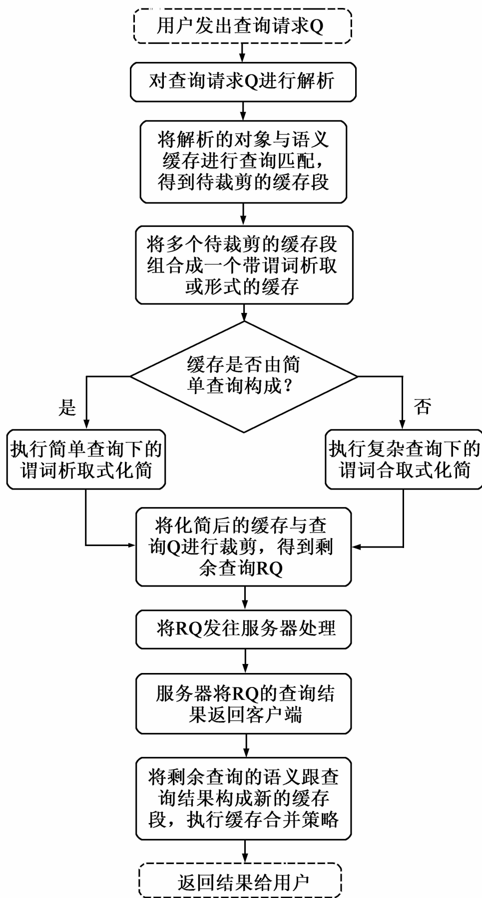

图1 语义缓存查询处理流程

3.3 逻辑优化规则

本文对析取范式进行优化处理,我们假设析取范式的形式如下:

$C = P_1 \wedge P_2 \wedge P_3 \wedge \cdots P_{i-1} \wedge P_i \wedge P_{i+1} \cdots \wedge P_n$, P_i 为一个谓词表达式.

$D = C_1 \vee C_2 \vee C_3 \vee \cdots C_{i-1} \vee C_i \vee C_{i+1} \cdots \vee C_n$.

我们在前面工作的基础上[11],给出了 20 条逻辑蕴涵规则,它们具有以下形式:

规则 1 如果 $\exists P_i, P_j \in C, P_i \wedge P_j = \text{false}$,则 $C = \text{false}$.

规则 2 如果 $\forall C_i \in D, C_i = \text{false}$,那么 $D = \text{false}$.

规则 3 如果 $\exists C_i \in D, C_i = \text{false}$,那么 $D = C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_{i-1} \vee C_{i+1} \cdots \vee C_n$.

规则 4 如果 P_j能由 P_i 推导得出,记为 $P_i \rightarrow P_j$,那么 $P_i \wedge P_j \leftrightarrow P_i$

规则 5 如果 P_j 能由 P_i 推导得出,记为 $P_i \rightarrow P_j$,那么 $P_i \vee P_j \leftrightarrow P_j$

规则 6 如果 P(Data)取值{Integer,Float,Double}. c, d 为常数,且 $c > d$,则 $\exists A \in P, (A >= c \wedge A <= d) \leftrightarrow P = \text{false}$.

规则 7 如果 P(Data)取值{Integer,Float,Double}. c, d 为常数,且 $c >= d$,则 $\exists A \in P(A > c \wedge A <= d) \leftrightarrow P = \text{false}$.

规则 8 如果 P(Data)取值{Integer,Float,Double}. c, d 为常数,且 $c >= d$,则 $\exists A \in P(A >= c \wedge A < d) \leftrightarrow P = \text{false}$.

规则 9 如果 P(Data)取值{Integer,Float,Double}. c, d, e 为常数,且 $c < e < d$,则 $\exists A \in P(A > c \wedge A < d \wedge A < e) \leftrightarrow \exists A \in P(A > c \wedge A < e)$.

规则 10 如果 P(Data)取值{Integer,Float,Double}. c, d, e 为常数,且 $c < e < d$,则 $\exists A \in P(A > c \wedge A < d \wedge A > e) \leftrightarrow \exists A \in P(A > e \wedge A < d)$.

规则 11 如果 P(Data)取值{Integer,Float,Double}. c, d, e 为常数,且 $c < e < d$,则 $\exists A \in P(A > c \wedge A < d \wedge A = e) \leftrightarrow \exists A \in P(A = e)$.

规则 12 如果 P(Data)取值{Integer,Float,Double}. c, d, e, f 为常数,且 $c < e < d < f$,则 $\exists A \in P(c < A < dVe < A < f) \leftrightarrow \exists A \in P(c < A < f)$.

规则 13 如果 P(Data)取值{Integer,Float,Double}. c, d, e, f 为常数,且 $e = d$,则 $\exists A \in P(c < A <= dVe < A < f) \leftrightarrow \exists A \in P(c < A < f)$.

规则 14 如果 P(Data)取值{Integer,Float,Double}. c, d 为常数,且 $c < d$,则 $\exists A \in P(A > cVA < d) \leftrightarrow P = \text{true}$

规则 15 如果 P(Data)取值{Integer,Float,Double}. c, d 为常数,且 $c <= d$,则 $\exists A \in P(A >= cVA < d) \leftrightarrow P = \text{true}$

规则 16 如果 A 为布尔型,且 $c != d$,则 $\exists A \in P(A = c \wedge A = d) \leftrightarrow P = \text{false}$.

规则 17 如果 A 为布尔型,且 $c = d$,则 $\exists A \in P(A != c \wedge A = d) \leftrightarrow P = \text{false}$.

规则 18 如果 A 为字符型,且 $c != d$,则 $\exists A \in P(A = c \wedge A = d) \leftrightarrow P = \text{false}$.

规则 19 如果 A 为字符型,且 $c = d$,则 $\exists A \in P(A != c \wedge A = d) \leftrightarrow P = \text{false}$.

规则 20 如果 A 为字符型,且 c 是 d 的子串,则 $\exists A \in P$(A *like*‘%c%’ $\wedge$ A *like* %d%’)$\leftrightarrow \exists A \in P$($A$ *like*‘%d%’).

规则 4 证明:

如果 $P_i \wedge P_j = \text{true}$,则 P_i 为 true,并且 P_j 为 true,所以当 $P_i \wedge P_j = \text{true}$ 时,$P_i = \text{true}$,即 $P_i \wedge P_j \leftrightarrow P_i$

如果 $P_i = \text{true}$,由 $P \rightarrow P_j$ 得到 P_j 也为 true,那么 $P_i \wedge P_j = \text{true}$

综上所述,如果 $P_i \rightarrow P_j$,那么 $P_i \wedge P_j \leftrightarrow P_i$, 证毕

由于其他规则的证明都比较简单直观,限于篇幅的原因,故本文不给出其证明过程.

3.4 简单查询的谓词化简策略

在语义缓存查询处理流程中,剩余查询裁剪算法是整个流程的瓶颈,而查询与缓存的谓词数目又直接关系到剩余查询裁剪算法需要比较的次数.因此缩减谓词的数目成了一个关键问题.对于简单查询,谓词合取式中的谓词属性为数值类型且属性名称相同,根据优化规则 1,4,6 – 11,20,我们可以把合取式中的谓词进行化简,得到一个相与后的数值范围区间.而当我们把待裁剪的缓存组合成析取式形式的查询时,我们可以根据优化规则 2,3,5,12 – 19,对谓词的析取式进行化简,最终得到一个更简化的谓词析取式形式,经过以上的化简,使得剩余查询裁剪算法的效率大大提升.具体的优化算法如算法 1 所示.

Algorithm 1

Dis-Optimize(D_P) //析取式的优化算法

输入:$D_P = C_1 \vee C_2 \vee C_3 \vee \cdots C_{i-1} \vee C_i \vee C_{i+1} \cdots \vee C_n$ 其中 Ci 为谓词合取式

$Ci = P_1 \wedge P_2 \wedge P_3 \wedge \cdots \wedge P_n$; Pi 为比较谓词.

输出:优化后的谓词析取式

(1)Ts←group(D_P) //对 D_P 根据每个谓词的属性名称进行分组,产生分组集合 *Ts*

(2)For(each Di in Ts){

(3)//对分组后的每个合取式,得出一个相交后的范围

For(each Ci in Di){

(4) For(each Predicate(x op c) in Ti){

(5) Find the range of x, rx = (a, b);

(6) Case op of

(7) <:(a, b) = (a, min(b, c));

(8) >:(a, b) = (max(a,, c), b);

(9) =:(a, b) = [c, c];

(10) End of case

(11) If($a > b$) break;

(12) }

(13) RangeList.add(*rx*)

(14) }

(15)//对 RangeList 按照范围的左边界从小到大进行排序

RangeList.sort();

(16)//得到 *RangeList* 相交的范围

For(each range in RangeList){

(17) If(rangeOne.getRight() < rangeTwo.getLeft())

(18) ResultList.add(range);

(19) else

(20) RangeTwo = new Range(rangeOne.getLeft(), rangeTwo.getRight());

(21) If(RangeTwo = = INFINITY)

(22) continue;

(23) }

(24) //将相交范围转为谓词析取式

D_P = ResultList.transformDisjuction();

(25) Return D_P

(26) }

3.5 复杂查询的缓存合并策略

对于简单查询,可以采用 Algorithm1 以及逻辑优化规则对查询裁剪进行化简,而对于复杂查询,因为其包含不同属性,我们采用另外的化简策略.假设用户的查询为 Q,缓存中与查询 Q 相交的缓存段集合为 S,$\neg Q$ 和 $\neg S$ 分别为 Q 和 S 的补集,当查询与缓存相交时,查询与缓存就被分割成 3 个独立部分,分别为 *Part*1: $S \wedge \neg Q$, *Part*2: $S \wedge Q$, *Part*3: $\neg S \wedge Q$,如图 2 所示,其中实线椭圆代表查询 Q,虚线椭圆代表缓存 S,有三种语义缓存段的合并策略:

- 完全合并:将三个部分合并为一个新的语义单元
- 部分合并:将查询结果与缓存进行部分合并,将会产生 2 个缓存片段,如图 2 所示
- 不合并:完全分开,将产生 3 个缓存片段 Part1、Part2、Part3,如图 2 所示

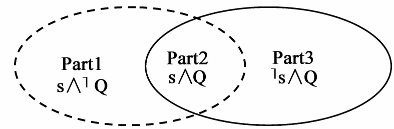

图2 查询与缓存的关系

不合并策略会使缓存块有一个更小的引用粒度,简化了查询处理,但是它的缺点是会增加缓存管理的负担.完全合并策略的好处是它减小了缓存段的数目,能够降低缓存管理的开销并且减少缓存与查询的匹配时间,因此缩短了查询响应时间.但是它的缺点是,随着查询的增多,缓存中的语义单元可能会越来越大,将降低了缓存的匹配效果,影响其利用率.对于简单查询来说,完全合并策略通过 OR 连接词将缓存段与缓存段之间的谓词析取式连接起来,构成更长的谓词析取式,在经过谓词析取式优化算法处理之后,使得谓词数目相比部分合并策略少了更多,因此,采用完全合并策略是对谓词析取式优化算法的补充,两者结合能进一步减少查询处理的时空开销.缓存完全合并的算法如算法 2 所示:复杂查询采用谓词析取式表达方式,谓词由多个属性组成,因此无法直接应用上述优化算法.

Algorithm2

SRC-mergingAll(S_{RC}, Q_S) //缓存完全合并算法

输入:S_{RC},能跟查询 Q 相匹配并进行裁剪的缓存段集合

Q_S,用户发出的简单查询

输出:合并后的缓存段
(1)　Query result←null
(2)　result = Q;
(3)　for(each S_Q in S_{RC})
(4)　//如果查询语句表名相同与缓存段相同,
　//并且 result 的属性包含查询的属性
　if(result.Get_Name_Q_R() = = S_Q.Get_Name_Q_R()
& &result.Get_Q_A().contains(S_Q.Get_Q_A()))
(5)　//合并两个语句的谓词析取式
　result.Get_Predicate_D_p().addAll(S_Q.Get_Predicate_D_p());
(6)　return result;

Algorithm3

SRC-dynamicMerging(S_{RC}, Q_C)　　//缓存动态合并算法
输入:S_{RC},能跟查询 Q 相匹配并进行裁剪的缓存段集合
　　Q_C,用户发出的复杂查询
输出:result (cache1, cache2)　//合并后的缓存段
(1)Query remainder$Q$ = S_{RC}.remainderQuery(Q_C)
(2)Query remainder$S$ = Q_C.remainderQuery(S_{RC})
(3)　if(remainderS.complexity() < remainderQ.complexity())　//根据谓词复杂度进行比较
(4)　　return (remaindS, Q_C);
(5)　else
(6)　　return(S_{RC}, remainder$Q$);

部分合并策略是完全合并策略与不合并策略的折中,它能够充分利用缓存,减少缓存碎片,在我们的研究中,我们采用动态部分合并策略(Dynamic Partial Merging 见算法 3),该策略基于谓词复杂度,能很好地平衡多个缓存的谓词复杂度,不会出现其中一方的谓词复杂度骤增的情况,从总体上缩短查询处理时间.以图 2 为例,Part2 跟 Part1 或者 Part3 合并的依据是:合并后产生的谓词复杂度在所有可选择的情况下最小.当缓存与查询合并时,有两种合并方法:一种是将 Part1 与 Part2 合并成为一块缓存,Part3 单独成一块缓存,此种方法成为 S-Partial-Merging.另一种是将 Part2 与 Part3 合并成为一块缓存,Part1 单独成一块缓存,此种方法称为 Q-Partial-Merging.方法的选择取决于哪种合并所产生的谓词复杂度最小,采用动态的部分合并策略的好处是可自适应地降低谓词复杂度,较好地降低裁剪时间开销.

4　实验结果及分析

4.1　实验环境

实验测试系统由移动客户端,服务器和无线网络组成,移动客户端展示了一个移动位置服务的应用,部署在 Android 操作系统上,服务器端维护地理信息数据库,语义缓存在移动客户端,所有的查询均由客户端发出.当有查询请求时,先在移动客户端处理,如果语义缓存命中,则在本地缓存进行处理;如果部分命中或没有缓存命中,则需通过网络向服务器端提交查询请求.系统框架如图 3 所示.移动客户端和服务器端的实验参数如表 1 所示.

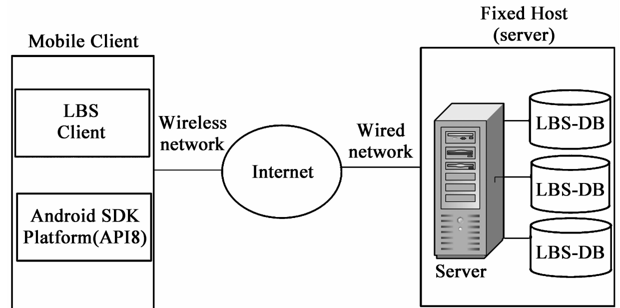

图3　系统框架

表 1　实验参数

参数	值	含义
Client-OS	Android2.2	客户端操作系统
Client-CPU	512M	客户端处理器主频
Client-Memory	96M	客户端内存容量
Server-CPU	2.93G	服务器端处理器主频
Server-Memory	1G	服务器端内存容量
Database	MySQL	服务器端数据库
NumberAttri	4	数据库表的属性数目
DataCount	500	数据库记录条数
CacheCount	0	初始的缓存个数
BandWidth	10M/19.2kb	网络带宽

本实验的测试用例按查询类型分别设计.对于简单查询(定义为 Q_S),它的测试用例是在移动客户端执行单表的选择操作,其中谓词合取式数目为 2,谓词属性为整形,范围在 0 – 500 之间,查询范围为 5 – 10 之间的随机值.例如:Select param From info-table Where param > 16 and param < 25.对于复杂查询(定义为 Q_C),测试数据的产生采用文献[13]的方法,即采用基于移动位置的基准测试下的随机运动模式(详细内容参见论文[13]).测试的场景如下:移动客户端在 400X400 的二维坐标平面内,查询距离当前位置 L 某一范围 R 内的对象集 O.例如 Select hotels From MapInfo-table Where locX > 4 and locX < 16 and locY > 51 and locY < 70.

L:是移动客户端提交查询时的位置

R:是 10 到 20 范围的随机值,即 $R \in [10,20]$

O:是查询对象的类型,$O \in$ [academic buildings, dormitories, hotels, playgrounds].

4.2 性能比较

4.2.1 简单查询下有优化和无优化的谓词化简时空效率对比

实验的第一部分选择了两项主要指标:析取式裁剪的时间和空间消耗,来对优化的查询裁剪的性能进行分析.实验首先比较了采取谓词析取式化简后相比化简前谓词的数目变化,接着在采用相同的缓存管理策略下,各自比较了谓词化简前后对查询处理时间的影响,在图4、图5、图6中,水平坐标是查询次数,垂直坐标是缓存中谓词的复杂度和查询处理的响应时间.

从图4可以看出,在未优化的情况下,查询语句的谓词长度是随着查询比较次数的增加而递增,到了最后,当缓存基本能完全包含查询时,递增的趋势才减慢下来.而采取谓词析取式化简算法之后,谓词的数目大大减少,在查询的中后期,谓词析取式之间合并的速度加快,会导致谓词数目逐渐减少,到了最后,当缓存完全包含查询时,谓词数目达到最低.

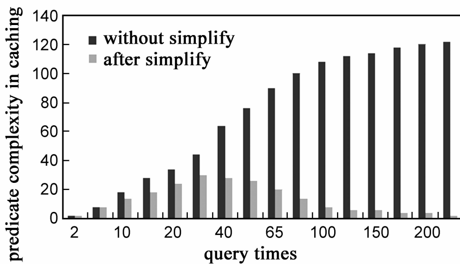

图4 谓词化简前后空间效率对比

从图5可以看出,在采用谓词部分合并策略的情况下,随着查询次数的增多,有优化的析取式化简相比未优化的情况下,递增明显减慢,这是因为查询处理过程中需要的谓词比较次数减少的缘故.

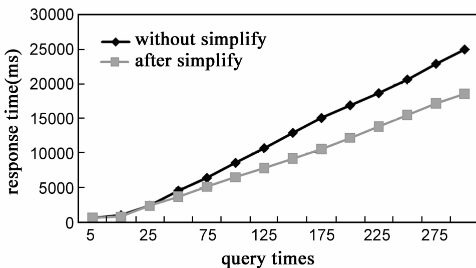

图5 S-Partially merging策略下的谓词化简响应时间对比

从图6可以看出,在采用谓词完全合并策略的情况下,随着查询次数的增多,有优化的析取式化简相比未优化的情况下,响应时间大大减少,到后期已经接近常数水平,这是因为缓存完全策略下,缓存段的数目也大大减少,再加上谓词析取式化简算法,两者结合以后,得到缓存查询效率提高明显.

4.2.2 简单查询下语义缓存合并策略对查询裁剪时间效率对比

实验的第二部分集中在简单查询下不同的缓存合并策略对查询处理效率的影响上,通过比较不同的缓存合并策略对查询响应时间的不同,达到验证缓存合并策略的目的,实验分成两个阶段,分别在有执行谓词析取式化简算法的条件下和没有执行谓词析取式化简算法的条件下进行比较.在缓存合并策略中,S-Partial-Merging将探测查询与原始的缓存进行合并.Q-Partial-Merging将探测查询与用户发出的查询缓存进行合并.不合并策略是将用户查询与缓存裁剪后的3个部分进行单独管理.完全合并策略则是将用户的查询裁剪后的结果与原始缓存进行合并,构成一个新的缓存段进行管理.在图7、图8中,水平坐标是查询次数,垂直坐标是查询处理的响应时间.

从图7可以看出,在不执行谓词化简算法的情况下,不合并策略所花费的时间大大超过其他合并策略,这是因为要进行多次的裁剪,才能得到独立的3个缓存段,因此耗费了多余的时间.S-Partial-Merging 和 Q-Partial-Merging 在时间效率上比较接近,而完全合并策略相比 Q-Partial-Merging 缩短了 12% 的响应时间,在完全合并策略下,通过将查询裁剪后的语句包含到缓存中,使得后续查询要比较的次数变少,因此缩短了缓存查询的响应时间.

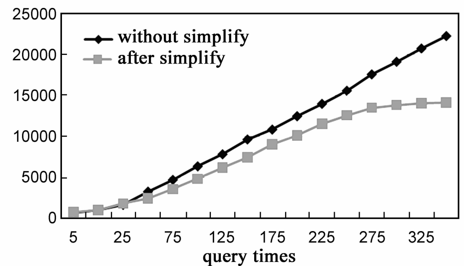

图6 缓存完全合并策略下的谓词化简响应时间对比

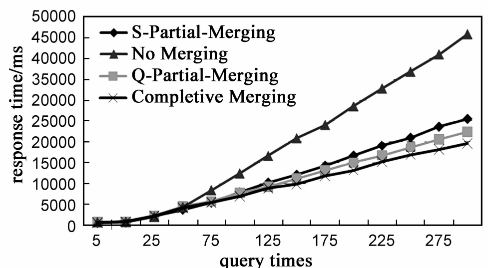

图7 不执行谓词化简下的缓存合并策略对比

从图8可以看出,在执行了谓词化简的条件下,完全合并策略相比未执行谓词化简算法下的完全合并策略,能缩短 19% 的响应时间,到了查询的后期,响应时间更是接近常数时间.将完全合并策略与谓词化简算

法相结合,使得语义缓存查询处理过程中要比较的谓词数目大大缩短,因而查询处理时间大幅度缩短,因此,在简单查询条件下,采用完全合并策略与谓词化简相结合是很好的策略.

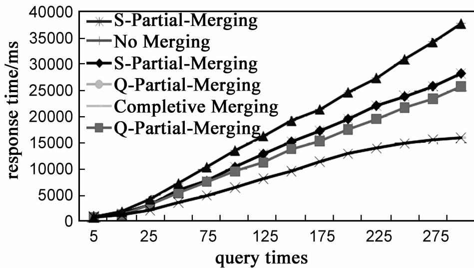

图8　在执行谓词化简下的缓存合并策略对比

4.2.3　复杂查询下缓存合并策略对查询裁剪时间效率对比

实验的第三部分关注的是复杂查询下缓存合并策略对查询处理时间的影响.我们比较 5 种缓存合并策略,比较的结果如图 9 所示,水平坐标是查询次数,垂直坐标是查询处理的响应时间.

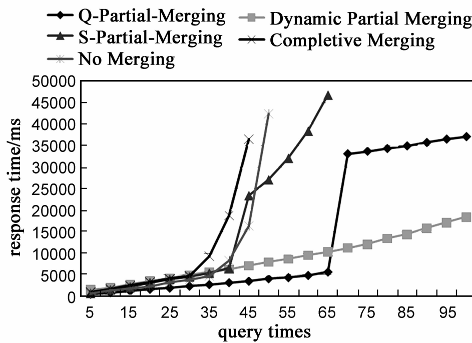

图9　复杂查询下的缓存合并策略对比

从图 9 可以看出,前 30 次查询,动态合并策略查询处理所需的时间要高于其他缓存合并策略,这是因为动态合并策略要进行两次剩余查询裁剪,但是从运行趋势上看,我们可以看到响应时间增长平稳,这是因为动态合并策略有效平衡了缓存与查询两端的谓词复杂度,保证了不会出现裁剪时间的骤增,而 Q-Partial-Merging 和 S-Partial-Merging 会在某次查询时生成了一个复杂度较高的缓存块,该缓存块又会作为下次裁剪的对象,使得裁剪次数递增很快,相应的查询处理时间也会骤增.而完全合并策略和不合并策略在运行效果上也明显不如动态合并策略.

5　结论

本文研究了语义缓存的查询优化问题.针对简单查询,提出了逻辑优化规则和谓词化简算法.针对复杂查询,进一步提出了基于谓词复杂度的语义缓存动态合并策略.在 Android 系统上的实验表明,在简单查询下,全合并的缓存管理策略和谓词析取式优化算法相结合,能明显地优化查询处理.在复杂查询方面,基于谓词复杂度的语义缓存动态合并策略能从总体上很好地平衡各缓存块的谓词复杂度,总体查询处理最优.本方法简洁实用,效果较好.我们的下一步工作是研究语义缓存的预提取技术.

参考文献

[1] Qun Ren, Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing[J]. in proceeding of MOBICOM 2000, 2000. 210 – 221.

[2] Parke Godfrey, Jarek Gryz, Answering Queries by Semantic Caches[A]. In proceeding of DEXA 1999[C], 1999. 485 – 498.

[3] Björn Tór Jónsson, María Arinbjarnar, Bjarnsteinn Tórsson, et. al. Performance and overhead of semantic cache management [J]. ACM Trans. Internet Techn. 2006, 6(3): 302 – 331.

[4] 吴婷婷,苏武运,周兴铭,徐明.移动查询处理的研究[J].计算机研究与发展.2004,41(1):187 – 193.
Wu Tingting, Su Wuyun, Zhou Xingming et al., Mobile Query Through Semantic Cache[J]. Journal of Computer Research and Development, 2004, 41(1): 187 – 193. (in Chinese)

[5] Qun Ren, Margaret H Dunham, Vijay Kumar. Semantic caching and query processing[J]. IEEE Transactions on Knowledge and Data Engineering, 2003, 15(1): 192 – 210.

[6] Ali-Asghar Safaeei, Mostafa Haghjoo, Sulmaz Abdi. Semantic cache schema for query processing in mobile databases [A], in proceeding of ICDIM[C]. 2008: 644 – 649.

[7] S. Kami Makki, Xunhang Zhou. Novel cache management strategy for semantic caching in mobile environment[J]. in proceeding of CSTST 2008. 192 – 197.

[8] 李东,杨晓鹏,罗鹏飞.基于谓词分类的语义缓存查询裁剪[J].华南理工大学学报(自然科学版),2008,36(1):44 – 49.
Li Dong, Yang Xiao-peng, Luo Peng-fei. Query Trimming for Semantic Cache Based on Predicate Classification[J]. Journal of South China University of Technology (Natural Science Edition), 2008, 36(1): 44 – 49. (in Chinese)

[9] B Zheng, D L Lee. Semantic Caching in Location-Dependent Query Processing[A], in proceeding of 7th International Symposium Spatial and Temporal Databases[C], July 2001. 97 – 116.

[10] LI Dong, YE You, XIE Fang-Yong. Optimization technology of query trimming in semantic caching[J]. Application Reseach of Computers. Dec. 2008, 25(12): 3605 – 3609.

[11] M F Bashir, R A Zaheer, Z M Shams and M A Qadir. Semantic Caching Architecture for Efficient Content Matching over Data Grid[A]. AWIC[C], Springer Heidelberg,

Berlin,2007.41 – 46.

[12] 李允,罗蕾,熊光泽.面向普适计算的自适应技术研究[J].电子学报,2004,32(5):740 – 744.
LI Yun,LUO Lei,XIONG Guang-ze.The Adaptive Technology for Pervasive Computing[J].Acta Electronica Sinica,2004,32(5):740 – 744.(in Chinese)

[13] Ayse Y.Seydim,Margaret H.Dunham.A Location Dependent Benchmark with Mobility Behavior[J].in proceedings of the International Database Engineering and Applications Symposium(IDEAS′02),2002.1098 – 8068.

作者简介

李 东 男.湖南人.2001 年获华中科技大学计算机科学与技术博士学位,现为华南理工大学教授.主要从事数据库、XML 和移动计算方面的研究.
E-mail:cslidong@scut.edu.cn

陈 锐 男.1989 年出生,广东揭阳人.华南理工大学软件学院研究生.主要从事移动数据库方向的研究.
E-mail:chengrui@foxmail.com

徐 杨(通讯作者) 男.1970 年出生,湖北人.2007 年获武汉大学计算机科学与技术博士学位,现为华南理工大学讲师.主要从事业务流程管理、服务计算和语义 Web 方面的研究.
E-mail:xuyang@scut.edu.cn