子句间优化技术在语义缓存查询求值中的应用

郝小卫 李 磊 章 陶
(中山大学软件研究所 广州510275)

摘 要 缓存技术用来弥补网络通信能力的不足。语义缓存既缓存查询结果,也缓存查询语义,可更好节省网络开销。实用性是语义缓存技术产生和发展的基础,根据实用性要求,语义缓存查询求值所消耗的时空代价要远小于网络通信能力不足带来的时空代价,所以必须对求值算法进行优化。文章从语法一级分析求值算法存在的问题并给出了两级优化方法和实现技术,降低了算法复杂度,减少了对数据库的无效访问,使语义缓存技术向实用化迈进了一大步。

关键词 语义缓存,查询求值,优化

The Application of Optimization Technique between Clauses in Semantic Caching Query Evaluation

HAO Xiao-Wei LI Lei ZHANG Tao
(Software Research Institute of SUN YAT-SEN University, Guangzhou 510275)

Abstract The caching technology is used for making up insufficient network communication capacity. Semantic caching can save network overhead by caching query result as well as query semantics. Practicality is the foundation of producing and developing semantic caching technology. According to the request of practicality, semantic caching query evaluation should cost much less space and time overhead than insufficient network traffic capacity caused. Therefore, optimizing the evaluation algorithm becomes a necessary. This article analyzed the problems of existing evaluation algorithm in a grammar level, and then proposed a two-level optimized method and its realization technique, which made the semantic caching technique a big progress in practicality by reducing algorithm complexity and invalid accessing to database.

Keywords Semantic caching, Query evaluation, Optimization

1 引言

计算机应用再也不可能是一个单机系统,而网络性能和速度的发展总是难以满足人们对其的要求。如无线网络比有线网络更加灵活,代表网络通信发展的主流和方向,但无线网络存在通信带宽窄、网络断接频繁等缺陷,成为移动数据存取的瓶颈[1];基于 Web 的数据库应用系统日益被人们接受并具有较大的优越性,但 Web 通信能力造成响应速度太慢已经成为制约其发展的主要问题[2]。

缓存技术预先把常用的数据缓存到客户端,用来弥补网络性能和速度的不足,是解决上述问题的主要手段之一。语义缓存技术既缓存数据本身,也缓存对数据的语义描述比只缓存数据本身可以更好地节省网络开销[3],该技术应用到分布式数据库系统中,即为在客户端既缓存查询结果,也缓存查询本身。

目前,人们对语义缓存技术进行了大量的研究,文[3]给出了语义缓存的明确定义,并描述了语义缓存技术较完整解决策略;文[4]专门讨论了语义缓存技术中的替代问题,并给出解决方案;文[5]针对移动计算模式下数据一致性问题提出了采用弱一致性增量式维护策略的方法;文[7,8]对如何从语义缓存中导出当前查询(部分)结果的问题,研究了查询从缓存导出的充分条件和相应的算法。但是,对语义缓存技术各个方面的研究尚处于理论和测试阶段,到真正实用还有一段距离,而实用性对语义缓存技术来说是至关重要的。

利用缓存的数据和语义进行查询求值是语义缓存技术是否实用的关键。如果利用缓存进行查询求值过于复杂,造成较高的时空代价,就会丧失语义缓存技术的本质和优势。因此,讨论利用缓存数据进行查询求值的代价并对其进行优化是非常必要的。在对实用性语义缓存技术进行研究的基础上,本文在语法一级专门讨论语义缓存查询求值代价并对求值算法进行优化,使得查询求值速度大大提高,从而满足语义缓存技术的实用性要求。

2 缓存模型

为满足语义缓存技术的实用性要求,考虑到嵌入式数据库自身的特点和优势,在我们的研究中,客户端采用定制的嵌入式数据库来缓存查询历史(查询语义及结果)。下面给出缓存的逻辑模型,并对其如何组织和管理数据进行描述。

2.1 预备定义

设 A、B 是原子公式。

定义2.1 如果存在替换 θ,使得 $A\theta=B$,则说 A 包含(Subsume)B,并记为 A SUB B。

定义2.2 如果 A SUB B ∧ B SUB A,则说 A 与 B 相等(Equal),并记为 A EQU B。

定义2.3 如果存在替换 θ 和 δ,使得 $A\theta=B\delta$,则说 A 与 B 是相关(相交 Interset)的,并记为 A INT B,否则说它们是无关的。

2.2 逻辑模型

设 $query(P_1),\cdots,query(P_n)$ 是已执行过的数据库查询,缓存的信息可以用 PROLOG 事实来表示:$meta\_base(P_1)$,

郝小卫 博士生,主要研究领域为数据库,人工智能,知识库。

$meta\_base(P_2),\cdots,meta\_base(P_n)$。我们把这些事实的整体称作 meta 库或历史库，每个事实中的 $P_i(i=1,2,\cdots,n)$叫作历史[9]。

如果将所有数据库查询加以缓存，则 meta 库中事实的数量将随着程序的执行而迅速地增加，这可能引起内存空间的溢出，同时影响效率，实际上这也是不必要的。

令 $Query(Q)$为当前数据库查询，$P_g,P_1,P_2,\cdots,P_n$ 为历史并且 P_g 包含 $P_i(i=1,\cdots,n)$。考虑一般情形：

(1) Q 与 P_g 无关，则 Q 与 P_i 无关；

因为假定 Q 与 P_i 相关，则有 θ 和 δ，使得 $Q\theta=P_i\delta$，由于 P_g 包含 P_i，于是存在 n 使得 $Pg\text{n}=P_i$，故有 $Q\theta=P_g\text{n}\delta$，这同 Q 与 P_g 无关矛盾。

(2) P_g 包含 Q；

(3) Q 与 P_g 相关，且(2)为假，则$\{Q\}-\{P_g\}=\{Q\}-\{P_1\}-\cdots-\{P_n\}$。

因为 P_g 包含 P_i，所以$\{P_i\}\subseteq\{P_g\}(i=1,\cdots,n)$，于是$\{P_g\}\cup\{P_1\}\cup\cdots\cup\{P_n\}=\{P_g\}$，故有$\{Q\}-\{P_g\}-\{P_1\}-\cdots-\{P_n\}=\{Q\}-(\{P_g\}\cup\{P_1\}\cup\cdots\cup\{P_n\})=\{Q\}-\{P_g\}$。

显然，情形(1)时需提取并缓存$\{Q\}$；情形(2)时 $Query(Q)$可完全由缓存得到解答，无需访问服务器，且$\{Q\}$也不必再缓存；情形(3)时只需要提取并缓存$\{Q\}-\{P_g\}$。

总之，如果 P_i 被某一 P_g 包含，所有情形都与 P_i 无关。也就是说，对于任何历史 P_i，如果有历史 P_g，使得 P_g 包含 P_i，则 P_i 在 meta 库中是冗余信息。根据这一结果，在历史库中去掉所有不必缓存的历史，可获得最小历史库存贮空间代价。

2.3 Meta 库管理算法(历史库维护算法)

设 $Query(Q)$为当前数据库提问，$P_i(i=1,\cdots,n)$为历史，如果有 P_i 包含 Q，则 Q 不加入到 meta 库中；否则，如果 P_i 被 Q 所包含，则删除所有 P_i，并将 Q 加入到 meta 库中。

此算法可用 PROLOG 描述为：

history(Q,H):- *setof* $(Q,meta\_base(Q),L),!,test(Q,L,H)$.

history$(Q,[])$:- *append_clause* $(meta\_base(Q))$.

test$(Q,L,[])$:- *subsumed*$(Q,L),!,fail$.

Test (Q,L,L):- *clear* $(meta\_base(Q))$, *append_clause* $(meta\_base(Q))$.

其中 *subsumed* 测试是否有 $P_i\in L$，使得 P_i 包含 Q；如果 Q 包含 P_i，*clear* 从 meta 库中删除所有 P_i；*history* 参数中的 H 返回 meta 库中所有与 Q 相关但不包含 Q 的元组构成的集合，用于下一步求值运算。

3 查询求值

利用语义缓存技术对客户端发出的查询 Q 进行求值，首先要将它同缓存的查询历史进行比较，判断其可否由缓存得到完全解答或部分解答，如可得到完全解答，无需再访问数据库服务器；如可得到部分解答，则要对查询 Q 进行修剪，将其分为可由缓存解答的部分 Q_p 和不可由缓存解答的部分 Q_r，只需将 Q_r 发往数据库服务器，从而减少了网络负载。

显然，查询求值主要是完成对查询 Q 的修剪，在我们的研究中，采用求差算法实现。

3.1 理论基础

事实上，关系数据库有一个逻辑解释[6]并且可以用逻辑表示。

设 $R:R(A_1,\cdots,A_n)$是一个模式，则 R 可以看作一个 n 元谓词。

设$\{R\}=\{(a_{11},\cdots,a_{1n}),\cdots,(a_{m1},\cdots,a_{mn})\}$是 R 对应的关系，则$\{R\}$可以看作谓词 R 的解释。

设 $R_1,\cdots,R_n$ 是模式，则 $D=\{R_1\}\cup,\cdots,\cup\{R_n\}$是数据库。

设 $query(R(t_1,\cdots,t_n))$是一个$\{R\}$上的数据库提问，其中 $t_1,\cdots,t_n$ 是项(常数或变量)，则 $R(t_1,\cdots,t_n)$可以看作是一个在 D 中待满足的公式。如果 $query(R(t_1,\cdots,t_n))=\varnothing$，则说 $R(t_1,\cdots,t_n)$在 D 中是不可满足的；如果 $query(R(t_1,\cdots,t_n))=\neg\varnothing$，则说 $R(t_1,\cdots,t_n)$在 D 中是可以满足的。

如果 $R(t_1,\cdots,t_n)$在 D 中是可以满足的，则说 D 是 $R(t_1,\cdots,t_n)$的模型，且$\{R(t_1,\cdots,t_n)\}=\{T_1,\cdots,T_n|R(T_1,\cdots,T_n)\wedge T_1=t_1\wedge,\cdots,\wedge T_n=t_n\}$是 $R(t_1,\cdots,t_n)$在 D 上是最小模型。

3.2 求值算法

对已执行过的数据库查询 $query(P_1),\cdots,query(P_n)$，$\{P_1\},\cdots,\{P_n\}$是结果集，根据2中讨论的结果，我们在缓存 $query(P_1),\cdots,query(P_n)$查询语义的同时，也缓存查询结果集的最小模型$\{P\}=\{P_1\}\cup\cdots\cup\{P_n\}$，以获得最小空间代价。

设 $query(Q)=\{Q\}$是当前数据库查询，$query(P)=\{P\}$，并且$\{P\}=\neg\varnothing$，$\{Q\}=\neg\varnothing$，则显然下面定理成立。

定理3.1 如果 P SUB Q，则$\{P\}\supseteq\{Q\}$

定理3.2 如果 P EQU Q，则$\{P\}=\{Q\}$

定理3.3 如果$\neg(P$ INT $Q)$，则$\{P\}\cap\{Q\}=\varnothing$，其中$\neg(P$ INT $Q)$表示 P INT Q 不成立。

但是，P INT Q 并不能得出$\{P\}\cap\{Q\}=\neg\varnothing$。因为$\{P\}\cap\{Q\}$是否等于空与 D 有关。例如：设 $P:P(a,Y)$，$Q:P(X,b)$，$D=\{P(a,a),P(b,b)\}$，则$\{P\}=\{P(a,a)\}$，$\{Q\}=\{P(b,b)\}$，明显$\{P\}\cap\{Q\}=\varnothing$。如果假定 $D=\{P(a,b)\}$，则有$\{P\}=\{Q\}=\{P(a,b)\}$，$\{P\}\cap\{Q\}=\{P(a,b)\}=\neg\varnothing$。

根据上述定理，查询求值可分下面四种情况执行：

(1) 如果 P SUB Q，并且由于$\{P\}\supseteq\{Q\}$，则$\{Q\}$只需从$\{P\}$中提取；

(2) 如果 P EQU Q，并且由于$\{P\}=\{Q\}$，则 $query(Q)$不必执行；

(3) 如果$\neg(P$ INT $Q)$，并且由于$\{P\}\cap\{Q\}=\varnothing$，Meta 库中没有符合 Q 的事实，直接将 $query(Q)$发往数据库服务器；

(4) 如果$(P$ INT $Q)\wedge\neg(P$ SUB $Q)$，则需求出 Q_r 并发往数据库服务器。

显然 $Q_r=query(Q\wedge\neg P)=\{Q\}-\{P\}=\{Q\}-(\{P_1\}\cup\cdots\cup\{P_n\})=\{Q\wedge P_1\wedge\cdots\wedge\neg P_n\}$

实际上，$\{Q\wedge\neg P_1\wedge\cdots\wedge\neg P_n\}$可表示为：

$\{t'_1,\cdots,t'_m|Q(t_1,\cdots,t_m)\wedge\neg P_1(t_{11},\cdots,t_{1m})\wedge\cdots\wedge\neg P_n(t_{n1},\cdots,t_{nm})\}$，其中 $t'_k(k=1,\cdots,m)$为新引入的变量。

由于 Q 与 P_i 相关，因此它们对应于同一数据库关系，于是上式可化简为：

$\{t'_1,\cdots,t'_m|Q(t'_1,\cdots,t'_m)\wedge T(t'_1=t_1\wedge\cdots\wedge t'_m=t_m)\wedge\neg(t'_1=t_{11}\wedge\cdots\wedge t'_m=t_{1m})\wedge\cdots\wedge\neg(t'_1=t_{n1}\wedge\cdots\wedge t'_m=t_{nm})$

即为：$\{t'_1,\cdots,t'_m|Q(t'_1,\cdots,t'_m)\wedge(t'_1=t_1\wedge\cdots\wedge t'_m=t_m)\wedge(t'_1\neq t_{11}\vee\cdots\vee t'_m\neq t_{1m})\wedge\cdots\wedge(t'_1\neq t_{n1}\vee\cdots\vee t'_m\neq t_{nm})$，其中$(t'_1=t_1\wedge\cdots\wedge t'_m=t_m)$部分叫做正子表达式，$(t'_1\neq t_{11}\vee\cdots\vee t'_m\neq t_{1m})\wedge\cdots\wedge(t'_1\neq t_{n1}\vee\cdots\vee t'_m\neq t_{nm})$部分叫做负子表达式。

把关系运算表达式写成 SQL 语句是没有困难的，其中 Where 子句部分由正和负子表达式构成。

4 求值优化

从查询求值算法不难看出，求差运算结果的关系代数表

达式中的负子表达式非常复杂，这就会导致SQL语句中的Where子句过于复杂，通常SQL语句的求值代价对于Where子句的复杂程度是敏感的；同时，一般数据库系统对Where子句中的子表达式个数是有限制的，从而可能造成某些自动生成的SQL查询无法执行。这些问题势必降低利用语义缓存进行查询求值的速度和效率，影响语义缓存技术的实用性，所以必须进行优化。

4.1 一次优化

我们知道在逻辑演算中有如下性质：

性质1 $1 \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_n = A_1 \wedge A_2 \wedge \cdots \wedge A_n 0 \vee A_1 \vee A_2 \vee \cdots \vee A_n = A_1 \vee A_2 \vee \cdots \vee A_n$

分析3中求差的结果$\{t'_1,\cdots,t'_m \mid Q(t'_1,\cdots,t'_m) \wedge (t'_1 = t_1 \wedge \cdots \wedge t'_m = t_m) \wedge (t'_1 \neq t_{11} \vee \cdots \vee t'_m \neq t_{1m}) \wedge \cdots \wedge (t'_1 \neq t_{n1} \vee \cdots \vee t'_m \neq t_{nm})$，由于当$t_{ij}(i=1,\cdots,n;j=1,\cdots,m)$为变量时，$(t_k \neq t_{ij}) \equiv$“假”$(k=1,\cdots,m)$，当$t_l(l=1,\cdots,m)$为变量时，$(t'_k = t_l) \equiv$“真”$(k=1,\cdots,m)$，它们可从上式中去掉，表达式得以进一步简化。

例4.1：$Q=act(a,Y,Z)$，$P_1=act(a,b,Z)$

由上述化简，$\{Q,\neg P_1\}$可以直接表示为：

$\{X,Y,Z \mid act(X,Y,Z) \wedge X=a \wedge (X \neq a \vee Y \neq b\}$

但是由于：$X=a \wedge (X \neq a \vee Y \neq b) \equiv X=a \wedge Y \neq b$

因此上述表达式还可以进一步化简，并容易证明：

令Q为当前数据库查询，P_i为历史，如果Q的第j个参数为常数，则P_i的第j个参数的内容对它们的差所对应的关系表达式没有任何影响。

根据这一结论，下面两例：

例4.2：$Q=act(X,b,Z)$，$P_1=act(a,b,c)$

例4.3：$Q=act(X,b,Z)$，$P_1=act(a,Y,c)$

其差具有相同的关系运算表达式：

$\{Q,\neg P_1\} \equiv \{X,Y,Z \mid act(X,Y,Z) \wedge Y = b \wedge (X \neq a \vee Z \neq c)\}$

总结以上分析，可以得到一个非常简化的求差$\{Q,P_1,\cdots,P_n\}$算法——差优化算法。

算法4.1：差优化算法

第1步：根据Q形成二元组集合$\{(x,i),(y,j),\cdots,(z,k)\}$，其中：$\{x,y,\cdots,z\} = VAR(Q)$并且$i,j,\cdots,k$分别代表参数$x,y,\cdots,z$在$Q$中的位置的整数；

第2步：对于第i个P_i做第3步；

第3步：对于每个$(x',i') \in \{(x,i),(y,j),\cdots,(z,k)\}$做第4步；

第4步：如果P_i的第i'个参量为常数C，则构造$(x' \neq C)$并以“$\vee$”相连加到P_i的负子表达式中；

第5步：所有负子表达式以“$\wedge$”相连加到表达式中；

第6步：根据Q构造正表达式并以“$\wedge$”相连加到表达式中（正子表达式的构造方法是简单的）。

例4.4：令$Q=act(a,Y,Z)$，$P_1=act(a,b,Z)$，$P_2=act(X,b,2)$

则二元组集合为$\{(Y,2),(Z,3)\}$；P_1,P_2对应的负子表达式分别为$(Y \neq b)$和$(Y \neq b \vee Z \neq c)$；Q对应的正子表达式为$(X=a)$；差优化结果为：

$\{X,Y,Z \mid act(X,Y,Z) \wedge (X=a) \wedge (Y \neq b) \wedge (Y \neq b \vee Z \neq c)\}$

对应的SQL提问为：

```
Select X,Y,Z
From act
Where X=a and Y≠b and (Y≠b or Z≠c)
```

4.2 二次优化

采用差优化算法，降低了查询求值的复杂度，从而减小了查询求值的时间代价，但求差结果关系表达式复杂且子表达式个数多的问题仍然存在，还应进一步进行优化，使查询求值真正满足实用性要求。

4.2.1 优化方法1 首先给出基公式的定义：P是一个基公式，如果P中的所有项为常数。

例4.5：P：$P(a,b,c)$是基公式，Q：$P(X,b,c)$不是基公式。

设P是基公式，Q是非基公式，则：P SUB $Q \equiv F$且$\{P\}$中仅可能有一个元组。

假定P_1：$R(a_{11},\cdots,a_{1n})$，P_2：$R(a_{21},\cdots,a_{2n})$，$\cdots$，P_m：$R(a_{m1},\cdots,a_{mn})$是基公式和历史，$Q(A_1,\cdots,A_n)$是当前待满足公式。

由于仅有可能Q SUB $P_i(i=1,\cdots,m)$，因此子句间优化结果为$Q \wedge \neg P_1 \wedge \cdots \neg P_m$，差优化的结果为：

$\{A_1,\cdots,A_n \mid R(A_1,\cdots,A_n) \wedge (A_1 \neq a_{11} \vee \cdots \vee A_n \neq a_{1n}) \wedge \cdots \wedge (A_1 \neq a_{m1} \vee \cdots \vee A_n \neq a_{mn})\}$

在上式中负子表达式的个数为$m \times n$，显然，这里造成Where子句中子表达式过多的主要原因是历史库中的基公式过多。

设P是基公式，由于$\{P\}$中仅有一个元组，因此优化利益不大。但它可造成Where子句中子表达式过多，因此可以认为优化代价过高，并且可能使得Where子句因过于复杂而无法求值。因此根据利益/代价原则，可以得出对于基公式进行优化是没有意义的。

4.2.2 优化方法2 如果我们把数据库D看作为解释，而查询$query(Q)$中的Q看作为公式，则Q在D中可能被满足也可能不被满足。如果Q在D中被满足，则D是Q的模型。明显$\{Q\}$是Q在D中的最小模型。如果$\{Q\}=\varnothing$，则Q在D上是不可满足的。

定理4.1 Q是公式，D是数据库，如果Q在D中是不可满足的，则对任意Q'，如果Q SUB Q'，则Q'也是不可满足的。

4.1节中讨论的差优化算法是与解释D无关的，仅处理了Q是可满足的情况。如果我们能进一步利用解释D的语义信息，则可以进一步提高优化利益和减少优化代价。

假定P是历史，并且P在D中是不可满足的，$query(Q)$是当前查询，则：

（1）如果P SUB Q，则$query(Q)$是不必要的。如果我们对不可满足的公式进行了缓存，则凡是被不可满足公式包含（SUB）的公式所对应的查询也是不必要的。因此可进一步减少对数据库的访问。

（2）如果P INT $Q \wedge \neg(P$ SUB $Q)$，则根据差优化算法应对数据库查询$query(Q \wedge \neg P)$。但是由于P在D中是不可满足的，因此$\{Q\}-\{P\}=\{Q\}-\{\ \}=\{Q\}$。因此可进一步得到$query(Q \wedge \neg P)=query(Q)$。根据上述结果Where子句中的负子表达式可进一步减少。

4.2.3 二次优化实现 根据上述讨论，差优化算法可完善和补充为：

1. 设$query(Q)$是当前查询，并且不是基公式。

（1）如果不存在历史P_i，使得Q INT P_i，则对于Q无优化可言。因此直接执行数据库提问$query(Q)$。如果$\{Q\}=\neg\varnothing$则按照4.1节中的差优化算法进行处理。如果$\{Q\}=\varnothing$，则首先标记Q是不可满足的，然后缓存在历史库中。否则：

(2) 如果有历史 P_i，使得 P_i SUB Q，则无论 P_i 是否可满足的，此次数据库访问 *query*(Q)是不必要的。否则：

(3) 如果有历史 $P_1,\cdots,P_n$，并且 Q INT $P_i(i=1,\cdots,n)$，则首先在 $P_1,\cdots,P_n$中滤除不可满足的历史，使得 $H=\{P_i|P_i$ 是可满足的，$i=1,\cdots,n,\}$，然后按照4.1节中所述的差优化算法处理 Q 和 H。

2. 设 *query*(Q)是当前查询，并且是基的，则对 Q 不进行优化，并且由于{Q}中仅可能有一个元组，则结果集中不缓存{Q}，而直接由调用程序消耗。

结论 语义缓存技术主要用来解决网络通信能力不足造成的响应时间太慢等问题。事实上，利用语义缓存进行查询求值的复杂度高于利用数据库服务器进行查询求值的复杂度，如果处理不当，造成利用语义缓存查询求值所消耗的时空代价接近甚至超过网络通信能力不足带来的时空代价，语义缓存技术显然是没有意义的。本文首先给出语义缓存查询求值的一般算法，然后指出采用一般算法得到的结果存在关系表达式过于复杂和子表达式过多的问题，最后针对这些问题从语法一级进行分析并给出了两级优化方法和实现技术。特别是在我们的优化方法中，不仅讨论了通常语义缓存只缓存与数据库内容相关的情况，而且创造性地讨论了缓存与数据库内容无关的情况，并指出如果对失败查询也进行缓存，可以进一步减少了对数据库的无效访问和简化 Where 子句中的子表达式，以较小的优化代价得到较大的优化利益。

采用语义缓存查询求值优化技术，不但可以简化语义缓存查询求值的复杂度，加快语义缓存查询求值的速度，而且减少了用户对数据库服务器的无效访问，减轻了网络通信负载，使语义缓存技术向实用化大大迈进了一步。

致谢 在此，我们向对本文的工作给予支持和建议的同行表示感谢！

参考文献

1 Ugur C, Deno P J K. A Decentralized, Peer-to-Peer Object-Replication System for Weakly Connected Environments. IEEE TRANSACTIONS ON COMPUTERS, JULY 2003,52(7)
2 Li W-S, Po O, Hsiung W-P, Candan K S, Agrawal D. Freshness-driven adaptive caching for dynamic content Web sites Data & Knowledge Engineering,2003,47:269～296
3 Ren Q, Margaret H. Dunham Semantic Caching and Query Processing. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, 2003,1(1)5: 192～210
4 Dar S, Franklin M, Jonsson B, Srivastava D, Tan M. Semantic data caching and replacement. In: Proc. of the 22nd VLDB Conf. Mumbai (Bombay), India, 1996. 330～341
5 Cai jun, Tao kian-Lee. On incremental cache coherency schemes in mobile computing environments [C]. In: Proc. of ICDE,1997
6 Gottlob S C, Tanca L. Logic Programming and Database, Springer-Verlag,1990
7 吴婷婷，周兴铭. 基于语义缓存的移动查询导出. 计算机学报，2002,10:1104～1110
8 吴婷婷，章文嵩，周兴铭. 断接下查询的缓存处理. 计算机学报，2003,10: 1393～1399
9 李磊，左万历，李希春. PROLOG-DBMS 系统实现中的子句间优化技术. 软件学报，1995,6(3):136～141

(上接第60页)

供的资源信息对其进行身份验证，通过验证的用户允许向 Web 服务器上传自己的 MPI 或者 PVM 并行应用程序包；2)上传的应用程序包通过经纪(Broker)的转换变成 MA 能够识别的请求，将源码包发送到编译移动代理结点，并请求编译；3)移动代理接收到该用户的编译请求后，则根据用户所写的 makefile 文件，通过移动代理选择一个高性能、轻负载的计算结点对用户程序进行编译，并把编译过程中遇到的错误信息回送给浏览器；4)用户根据出错信息对程序进行修改，如此重复，直到编译通过为止；5)用户向经纪发起运行请求，与编译过程的处理过程一样，经纪将其转换成移动代理可识别的请求并发送给移动代理；6)移动代理收到运行请求之后，把已编译好的可执行代码分发到通过智能抉择所选的各个结点；7)移动代理根据所选结点信息生成一个符合 MPI 或 PVM 格式的配置文件，并作为主结点据此发起计算；8)运行结束后，把运行结果回传给浏览器，等待下一个用户进程的请求。

图2 移动代理工作过程示意图

结束语 本文介绍了一种独立于具体应用业务和特定开发平台的基于移动代理的网格计算概念模型。它把分布、异构、动态变化的计算资源集中起来，为用户提供高性能并行计算平台。模型中的 Mobile Agent 就是一组智能地代替用户完成工作的程序。智能 Mobile Agent 能够发现和自动适应资源的不断变化并对应用行为做出相应的调整，结合当前计算结点的性能指标、网络带宽、用户需求等选择较优的计算结点，因此与传统的并行计算随机选择结点相比可大大提高计算的性能。并且 Mobile Agent 可在多种体系结构下编译、运行程序，故该模型可有效屏蔽系统中计算结点的异构性。该模型很好地适用于动态的网格环境，使得开发具有高性能的应用成为可能。这种结构不依赖于具体应用程序的算法，具有可移植和重用性。在以后的研究中，我们将要在移动代理系统的安全性方面做进一步的研究，必须有较好的容错方案来保证移动代理的健壮性，集中在对移动代理的合法性验证、对移动代理所携带的数据的保护以及防止某些恶意攻击及对执行环境非授权的修改等方面。

参考文献

1 都志辉，陈渝，刘鹏. 网格计算. 清华大学出版社，2002
2 Gentzsch W. Grid computing, a vender's vision. In: Proc. of the 2nd IEEE/ACM Int'l Symp on ClusterComputingandtheGrid. LosAlamitos: IEEEComputingSociety Press, 2002. 272～277
3 Cao J W, Spooner D P, Turner J D, et al. Agent-based resource management for grid computing. In: Proc. of the 2nd IEEE/ACM Int'l Symp on Cluster Computing and the Grid. Los Alamitos: IEEE Computing Society Press, 2002. 323～324
4 Fukuda M, Suzuki N, Bic L F. Introducing dynamic data structure into mobile agents. In: Proc. of the1999 Int'l Conf. on Parallel and Distributed Processing TechniquesandA-pplications - PDPTA'99, Las Vegas, NV, June 1999. 1845～1860
5 Foster I, Kesselman C. The Globus Project: A Status Report. In: Proc. IPPS/SPDP'98Heterogeneous Computing Workshop, 1998. 4～18
6 Fukuda M, Tanaka Y, Campos L M, Kobayashi S. Inter-cluster job coordination using mobile agents. In: Proc. 3rd Int'l Workshop on Active Middleware Servic-es, San Francisco, CA, IEEE CS, Aug. 2001
7 Vetter J, Schwan K. Techniques for High - Performance Computational Steering. IEEE concurrency, 1999. 63～74
8 Vetter J S, Reed D A. Real time Performance Monitoring, Adaptive Contral, and Interactive Steering of Computing Grids. International Journal of High Performance Computing Applications, 2000. 357～366
9 Fosteer I. Computer Grids. In: VECPAR2000, 2002. 3～37
10 Fukuda M, Tanaka Y, Suzuki N, Bic L F, Koba-Yashi S. A Mobile-Agent-Based PC Grid. In: Proc. 3rd Int'l Workshop on Active Middleware Services, IEEE CS, 2001