一种提高并行数据挖掘效率的方法*)

佘春东[1] 范植华[2] 孙世新[1] 车著明[3] 唐 剑[2]
(电子科技大学计算机科学与工程学院 成都610054)[1]
(中国科学院软件研究所 北京100080)[2] (西昌卫星发射中心技术部 西昌615000)[3]

摘 要 发现关联规则是数据挖掘的一项重要任务,本文介绍了几种数据挖掘的串行和并行算法。其中IDD算法是一种高效的和易于扩展的发现关联规则的并行算法,然而,当处理器数目增加时,由于负载的失衡导致其效率的严重下降,于是通过引入近似算法成功地解决了这个问题。我们给出了两种近似算法和其性能证明,其一是在线算法,另一种是离线算法。在本文的最后,我们进行了改进的IDD算法的复杂性分析。

关键词 数据挖掘,并行处理,关联规则,负载平衡,可扩展性,近似算法,在线算法,离线算法

A Method to Improve the Performance of Parallel Data Mining

SHE Chun-Dong[1] FAN Zhi-Hua[2] SUN Shi-Xing[1] TANG Jian[2] CHE Zhu-Ming[3]
(School of Computer Science and Engineering, UESTC, Chengdu, 610054)[1]
(Institute of Sofware, Chinese Academy of Sciences, Beijing 100080)[2] (Department of Technology, XSLC, Xichang 615000)[3]

Abstract Discovery of association rules is an important data mining task. Several parallel and sequential algorithms have been proposed in this paper to solve the problem. IDD algorithm is an efficient and scalable parallel method applied in the discovery of association rules in the field of data mining. However, it becomes less effective when processors increases due to the imbalance. Therefore, IDD is improved by means of introducing approximate algorithms to solve the problem of load balance effectively. There are two approximate algorithms, one is called online algorithm, and the other is named offline algorithm. After that, we give the proof of their performance ratio. In the last part, it is the complexity analysis of the improved IDD algorithm.

Keywords Data mining, Parallel processing, Association rules, Load balance, Scalability, Approximate algorithm, Online algorithm, Offline algorithm

1. 引言

最近,各种各样的可用的信息和数据发展十分迅速,这提供了前所未有的机会去发展自动的数据驱动技术来提取有用的知识。数据库中的知识发现技术(Knowledge discovery in database, KDD),我们常称之为数据挖掘(Data mining)技术,作为在知识发现进程中所迈出的重要一步,包括一些发现深藏在一般数据中的有用的模式。数据挖掘思想是建立在其它各类领域诸如机器学习、模式识别、统计、数据库系统以及数据可视化等基础之上的。但应用于这些领域和传统学科的技术往往已经不适用,因为当今的数据集合有着许多独特的性质,如数据量巨大、多维性以及异构的特性[1,3]。

推动数据挖掘技术迅猛发展的主要动力来自于应用在商业、信息检索、金融等领域的庞大的数据集合中的算法的开发应用。更为重要的是,由于最近的技术进步,产生了大量的可应用的科学数据,其产生的速度之快已远远超出人工分析这些数据的能力。比如,运行于高性能计算机上的计算模拟程序数小时内可产生数以兆兆字节的数据,而人工分析这些数据要花费几周甚至更长的时间才能从中发现有用的信息。数据挖掘技术在开发新的工具集用于自动地分析这些模拟程序所产生的数据方面将大有作为,如此就可帮助工程师和科学家们弄清深藏于动态物理过程背后的偶然关系。近期可发现数据挖掘在其他领域的广泛应用,比如基因学中分析和理解基因的作用以及天体物理学中对行星和星系的分类。

可利用的数据规模之大以及其多维的特性使得对大规模数据挖掘的应用需求日益增强。于是在很大程度上高性能的并行计算成为解决此类问题非常有效的手段。此外,数据挖掘的成效还取决于其可用的数据来源。可以毫不夸张地说,不久的将来数据挖掘技术将是超级运算的主要应用者。这样一来,为各种数据挖掘技术开发行之有效的并行算法尤为重要。

2. 并行数据挖掘

数据挖掘的一个重要任务就是从事务数据库中发现关联规则[2],其中每个事务都包括一个项目集,统计这些项目的子集(或者称为候选集)的出现频度是非常耗时间的一个操作。由于通常基于事务的数据库都包含大量不同的项目,因此候选集的总数也很大,所以当前的关联规则发现技术都是通过要求满足一个最小支持度以尽量减小搜索空间。支持度是候选集在数据库事务中出现次数的量度。

2.1 基本概念

*)本文得到中国科学院知识创新工程方向性研究项目基金(名称:大型数字对象应用环境及其并行模拟,批准号:KGCX2-JG-09)和总装备部西昌卫星发射中心实验技术项目基金的资助。**佘春东** 博士研究生,主要研究方向为并行计算与数据库技术。**范植华** 研究员,博士生导师,主要研究方向为大规模并行计算和数据库技术。**孙世新** 教授,博士生导师,主要研究方向为并行计算技术。

设 T 为事务集合，其中每个事务都是项目集 I 的子集，C 为 I 的子集，则支持数(support count)定义为：

$$\sigma(C)=|\{t|t\in T, C\subseteq t\}|$$

所以支持数为包含 C 的事务个数。

关联规则是指 $X\overset{s,\alpha}{\Rightarrow}Y$，其中 $X\subseteq I, Y\subseteq I$。规则 $X\overset{s,\alpha}{\Rightarrow}Y$ 的支持度 s 定义为：$\sigma(X\cup Y)/|T|$，而置信度 α 定义为 $\sigma(X\cup Y)/\sigma(x)$。置信度接近1的规则十分重要，因为它提供了对项目之间关联的比较准确的预测。支持度也很重要，它表明了该规则在事务中的出现频度，低支持度的规则也没什么用，因为它毕竟只适用于很小范围，这就是为什么算法中只考虑满足最小支持度的规则[7~9]。当然，只考虑满足最小支持度的情况也能有效地减少计算规模。注意总的规则数目与项目集的子集数$2^{|I|}$成正比，因此实际使用时通过加限制条件进行过滤是必要的。

发现关联规则的主要任务就是找出形如 $X\overset{s,\alpha}{\Rightarrow}Y$，并且 s 大于等于某个给定的最小支持度，α 大于等于某个给定最小置信度。通常有两个步骤：首先找出频繁出现项目集(满足最小支持度的候选集)；然后再从这些集合中找出关联规则。显然第一步的计算代价比第二步高得多。因此，本文着重于第一步，即为频繁项目集的产生。第二步的并行实现简单明了，在文[10]中有详细的讨论。

2.2 串行算法

大多数发现关联规则的算法的思想核心都来源于 Apriori 算法[3]，它的性能明显优于早期的其他算法[4~7]。Apriori 算法利用传递闭包的性质减少大小为 k 的候选集个数：一个频繁项目集的所有子集也是频繁项目集，换句话说，大小为 k 的候选集如果满足最小支持度，则它的所有子集也满足最小支持度。这样在第 k 次迭代中，算法就根据所有的大小为 k−1的频繁项目集算出大小为 k 的候选集的出现次数了。为了提高效率，算法将所有大小为 k 的候选集组织为一张哈希树，另外，算法不需要事务驻留在主存中，但哈希树必须在主存中。如果，哈希树太大不能全部放在主存中，则哈希树需要被划分，大量数据需要在各个事务数据库之间传递(对哈希树的每个分块)。但即使像 Apriori 这样的高效算法，从许多应用中发现关联规则的任务还是必须要并行机才能完成。而且，几乎所有高效的并行数据挖掘算法都是基于 Apriori 算法的[11]。

2.3 并行算法

文[10]提出了两种基于 Apriori 的并行实现方案：计数分发(count distribution，CD)和数据分发(data distribution，DD)。CD 算法能够线性扩展、高加速比，并且能用于大量事务的系统。但也有两个问题：首先，建立哈希树的过程非并行，在串行算法中，建树的过程所花时间相对较少，但在并行计算中，建树过程是个瓶颈；其次，如果哈希树不能完全放入到主存中，则在各个分块之间传递数据还需要额外的磁盘 I/O 操作，对于 I/O 较慢的计算机而言这是不小的开销。所以，CD 算法，与串行的 Apriori 算法类似，从增大候选集的大小的角度是不能扩展的，因此，CD 算法适合于处理较小的项数、较高的最小支持度的情况。

DD 算法针对 CD 算法存在的存储问题进行改进，将候选项目集按循环方式划分到各个节点，每个节点计算本地存储的候选项目集在所有事务中的出现次数，因此每个节点需要访问存放在其它节点的事务。而且随着处理器个数的增加，算法能处理的候选项目集个数也相应增加。但是，这个算法由于以下三个方面的原因而导致了效率低下：第一，由于用于数据迁移的效率低下的模式导致的高数据通信代价；第二，处理器之间的交互可能导致处理器空闲；第三，每个事务的处理都涉及多个哈希树，这会导致冗余计算。

为了解决 DD 算法的问题，文[11]引入了智能数据分发(intelligent data distribution，IDD)算法。在 IDD 中，本地事务通过基于环的多对多广播，这种方法主要好处是不会引起网络上的竞争，可运行在任何能嵌入环的并行机系统中。首先，在这个逻辑环上的节点检测它的左邻节点和右邻节点，每个节点都有一个发送缓冲区(sbuf)和一个接收缓冲区(rbuf)。初始化时将 sbuf 置为一块本地事务数据，然后所有节点向右邻节点进行异步发送操作，发送的数据存放在 sbuf，同时向左邻节点进行异步接收操作，接收的数据存放在 rbuf。在处理这些异步操作的同时，各个节点的处理器处理 sbuf 中的事务，并修改该节点处理的候选项目集的计数，完成上述操作后，等待异步操作的完成。接下来，交换 sbuf 和 rbuf 中的内容。整个循环进行 P−1次。与每个节点都要向其它节点发送数据的 DD 算法相比，IDD 算法只进行相邻节点之间的点对点通信，以此来消除网络竞争。而且，如果处理缓冲区的时间不是特别大的话，处理器等待的时间也会很短。

为了消除划分候选项目集带来的冗余工作，必须找出一个快速的方法检测一个事务是否含有该节点应处理的候选项目。在使用轮转方式划分候选项目集的情况下很难找到快速的方法。但是，如果根据事务中项目的开头的一个前缀进行划分，则可以通过检测事务长的项目的前缀以决定哈希树中是否含有以该前缀开头的项目。如此一来，解决了冗余工作的问题。

但是，对哈希树的静态划分导致了负载不平衡，文[11]明确指出，随着处理器数目的增加，由于负载不平衡而导致了算法的效率严重下降，即使是最优通信方案，通信代价也会随着事务个数增长成线性变化，为此，本文提出了通过引入多处理机调度算法以平衡负载，并选择合适的近似算法降低调度的开销。

3. 多处理机调度算法

首先给出多处理机调度问题本身：(Minimum Scheduling On Identical Machines)

实例：工作集 T，处理机个数 p，对 $t_i\in T$，l_i 为其执行时间

求解：T 在 p 个处理机上的调度方案

度量：完成 T 中所有工作的时间，即为：

$$\max_{i\in[1\cdots p]}\sum_{t_j\in T\cdot f(t_j)=i} l_j$$

使用顺序算法解决上述问题，即寻找一种调度使得工作集 T 的所有元素在 p 个处理机的系统上最快地完成执行。显然，这个问题是一个 NP-hard 问题，即使 p=2也是如此。

如果已知各个工作的到达顺序，为了进行调度应该确定如何将一个工作分配给一个处理机。一个比较显然的方法是：将这个工作分配给当前负载最小的处理机。更具体地说：如果前 j−1个工作已经被分配，设 Ai(j−1)为第 I 个处理机完成当前已分配给它的工作的时间，即 $A_i(j-1)=\sum_{1\le k\le j-1,\ f(t_k)=i} l_k$，则第 j 个工作被分配给完成时间最小的处理机。这个算法被称为 List Scheduling。

定理 给定多处理机调度问题的实例 x，处理机个数为 p，对任何工作序列，List Scheduling 算法能找到度量为 $m_{LS}(x)$ 的近似解满足：

$$m_{LS}(x)/m^*(x)\leqslant(2-\frac{1}{p})$$

其中 $m^*(x)$ 为最优解的度量。证明见文[12]。

容易看出，一个简单而实用的改进 LS 算法的方法就是先对工作按照处理时间进行非增排序，即为：

$$l_1\geqslant l_2\geqslant\cdots\geqslant l_{|T|}$$

改进后的算法称为 LPT 算法（Largest Processing Time），这个算法相比前面的算法性能有显著的提高。

定理 给定多处理机调度问题的实例 x，处理机个数为 p，对任何工作序列，LPT 算法能找到度量为 $m_{LPT}(x)$ 的近似解满足：

$$m_{LPT}(x)/m^*(x)\leqslant(\frac{4}{3}-\frac{1}{3p})$$

其中 $m^*(x)$ 为最优解的度量。证明见文[12]。

4. 通过多处理机调度算法平衡 IDD 算法中各个节点的负载

为了使 IDD 算法中候选项目集的划分尽可能地均匀，可以使用一个类似传统的多处理机调度问题的算法进行划分。具体做法是，首先计算出由某个特定元素开头的候选项目集个数，这时不需要存储这些候选项目集，只用算出个数，注意，如果使用的是 LPT 算法，则在计数的同时对候选项目集进行非增排序。接下来用解多处理机调度问题的算法，将上述候选项目集分配到 P 个节点，使得各个节点候选项目集个数大致相同。只要划分完毕，接下来每个节点产生并存储由该节点处理的候选项目集。注意，每次循环都会重新调用多处理机调度问题算法。

需要注意的是：这种方法在由其中一个特定元素开头的候选项目集个数特别多的情况下并不能均匀地划分候选项目集。例如，如果由 a 开头的候选项目集个数大于 M/P，则处理它的节点即使不再处理其它候选项目集也需要处理大于 M/P 的候选项目集。在 P 增大时，这个问题更加严重。解决这个问题的一个方法是，根据前两个或者更多的元素划分候选项目集。即为如果第一个元素开头的候选项目集个数大于 M/P，则对该集合按照第二个元素再进行一次划分……以此类推；另外一种方法是，如果第一个元素开头的候选项目集个数大于 M/P，则由前两个元素开头进行划分所有候选项目集。

还有一点需要注意的是：划分均匀并不完全能保证负载平衡。这是因为遍历哈希树以访问相应的叶节点的时间不仅和哈希树的大小和形状有关，而且和哈希树的具体的节点有关。然而，访问时间和哈希树的大小的关系非常紧密。

需要指出的是，虽然 LS 算法比 LPT 性能要差，但是它是在线算法，因此可以对每个产生的候选项目进行立即处理，而不用等到所有候选项目都产生后才开始处理，因此它的开销非常小，LPT 算法则是离线算法，因为需要对由某个特定元素开头的候选项目个数按照大小排序，因此必须等到所有候选项目产生后才能开始处理。LPT 算法适用于候选项目集较大的情况，LS 算法适用于实时性要求较高的情况。

5. 算法性能分析

5.1 理论分析

在本节中，对算法的复杂性和可扩展性进行分析。这里的可扩展性指，如果问题规模增加，则通过增加节点个数，性能不会降低。设 T_{serial} 为串行算法的执行时间，T_p 为并行算法的执行时间，则并行算法的效率定义为：

$$E=\frac{T_{serial}}{P\times T_p}$$

如果 $P\times T_p$ 和 T_{serial} 保序，则称并行算法是可扩展的。Apriori 算法的问题规模随着 N 或者 M 的增加而增加。

如文[11]所述，Apriori 算法的每次迭代分为两个步骤：(1)产生候选项目集和构造相应的哈希树；(2)对候选项目集在事务中的出现进行计数。考虑含有 I 项的一个事务，在第 k 次循环，可能的候选项目集个数为 $C=\binom{I}{k}$，因此需要在哈希树上检查这 C 个节点。注意对于给定事务，所有需要检查哈希树的某个叶结点的候选项目集只需对这个叶结点检查一次即可。这样，如果这个叶节点被同一事务的不同候选项目集访问时，不需要再进行检查。显然，检查叶结点的开销与该事务需要访问的不同的叶结点的个数成比例。假设处于叶结点的平均候选项目集个数为 S，则哈希树上的平均叶结点个数为 L=M/S。算法实际实现中，可以通过调整哈希树的分支数来获得需要的 S。通常，遍历所有候选项目集的开销主要由要访问的哈希树的叶结点的深度决定。为简化分析，假定每次遍历开销相等，则整个开销与 C 成正比。对每个候选项目集，定义 $t_{traverse}$ 为遍历的开销，t_{check} 为检查叶结点的开销。

IDD 算法的每个节点访问的平均的叶结点个数为 L/P，但每个节点需要被检查的候选项目个数较少。而且，使用近似算法后，多处理机调度的开销相对整个算法运行而言可以忽略，因此，对每个事务而言：

$$T_{trans}^{IDD}=\frac{C}{P}\times t_{traverse}+V_{\frac{C}{P},\frac{L}{P}}\times t_{check}$$

则每个节点的计算量为：

$$T_{comp}^{IDD}=N\times T_{trans}^{IDD}+O(\frac{M}{P})+O(N)=N\times\frac{C}{P}\times t_{traverse}+N\times V_{\frac{C}{P},\frac{L}{P}}\times t_{check}+O(\frac{M}{P})+O(N)$$

其中第一部分为 N 个事务的代价，第二部分为构造大小为 M 的哈希树的代价，第三部分为数据搬运代价。

IDD 算法的哈希树的遍历代价具有线性加速比。而且，检查叶结点的时间代价也是具有线性加速比，并没有冗余工作。

IDD 算法随着 N 的增加而不可扩展，随着 M 的增加可以扩展。

5.2 实验结果分析

实验是在曙光 TC1700并行机上进行的，OS 为 Linux，并行计算环境为 MPI，CPU 个数为8个。随机产生的事务数据大小为2.2k，最小支持度为1%，在千兆以太网环境下的性能数据如表1所示。

表1

CPU 个数	1	2	4	8
响应时间1(s)	2.53	2.05	1.52	1.07
响应时间2(s)	2.05	1.43	1.07	0.76

其中响应时间1为普通 IDD 算法的响应时间，响应时间2为引入 LPT 算法的响应时间。可以看出引入 LPT 算法的性能有着明显的提高，取得了令人满意的效果。

小结 本文简要介绍了发现关联规则的 Apriori 算法及其多种并行实现，包括 CD、DD、IDD 算法。然后，针对 IDD 这

（下转第166页）

3.3 测试小结

评测的结论与以前版本基本一致。应该指出，尽管设备层对性能的影响最大，但在具体接口(如 SendRecv)实现中使用的策略也会对性能有影响。此外，在具体运行中，MPI 程序的差异也可能造成使 MPI 实现表现出有区别的性能。总的来说，如果应用环境是固定的，而且不在多设备等方面有进一步要求，那么可以选择 LAM 使用，因为它的性能更好。如果进行研究与移植，那么 MPICH 是更好的选择。

4. 改进与结论

MPI 库的性能，很大程度上依赖于通信环境。如果要重写设备层来减小软件开销，主要需要考虑1.3.1节的三方面。数据的拷贝更少，调用的层次更少，或通信的协议更先进，都可能导致性能的提高。

MPI 标准规定的非阻塞通信，为体现计算与通信的并行工作。MPICH 和 LAM 中对队列处理函数频繁调用，很好地模拟了计算与通信的并行执行。但单线程的本质，使它们不能够做到更细粒度的并行。大粒度的并行，对通信效率也有影响，比如在 LAM 调用 MPI_Send 时，先建立发送请求对象，然后放入队列，再对队列进行阻塞处理。队列处理函数的阻塞模式，是反复处理队列中的所有请求对象，直到其中没有阻塞标志的对象完成为止。可见，要发送的消息并不一定以最快的速度发出。从计算与通信的角度看，这是由于并行粒度大，导致的通信与通信抢占资源，而不能与计算并列执行。一个直观的解决设计，就是多线程。让队列处理的任务，在一个与用户计算独立的进程中执行，可以提高并行度。MPICH 与 LAM 都给多线程的实现留出了一定的余地，但 MPICH 更加易于实现多线程。这还是在于 ADI 层的抽象设备。MPID_Device 与 API 层的独立，是 MPICH 设计中考虑过的，但没有完成。ADI 层在队列处理函数留有线程锁等代码，这些都为多线程的实现提供了便利条件。多线程设计，最重要的应是解决同步关系。被同步的对象应尽可能地少，因为同步(无论是否经过系统的)会耗费时间。MPICH 给改写人员更大的自由度，可以自己处理请求队列。因为 MPICH 的请求队列完全置于设备层中，所以队列可以交给设备进程。LAM 要做到这一点，需要做更大的改动。此外在进程通信方面，也要考虑像函数调用与返回这样的问题。

MPICH 与 LAM 实现了 MPI1，但没有完全实现 MPI2。现在 MPICH2的 beta 版已经公布，它实现了 MPI2的全部内容。新的 ADI 更巧妙地实现了多设备同时存在并可并行工作，虽然目前的 MPICH2 beta 版还没有实现 TCP 以外部分和多线程，但正式版总会实现多个 ADI 通讯设备的并行处理。现在 MPI 受到广泛欢迎的情况下，相信 MPI 库还会得到不断的进步。

参考文献

1 Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", 1994. http://www.mpi-forum.org/docs

2 Message Passing Interface Forum, "MPI2: Extensions to the Message-Passing Interface Standard", 1997. http://www.mpi-forum.org/docs/mpi-20.ps

3 Gropp E, et al. A High-Performance Portable Implementation of the Message Passing Interface. Journal of Parallel Computing, 1996,22:789～828

4 Burns G D, Daoud R B, Vaigl J R. LAM: An Open Cluster Environment for MPI. Supercomputing Symposium '94, Toronto, Canada, June 1994

5 Ong H, Farrell P A. Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. In: the Proc. of the 4th Annual Linux Showcase and Conf. Atlanta, Georgia, USA, Oct. 2000

6 Nevin N. The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster: [Technical Report]. Ohio Supercomputer Center, 1996

7 GmbH P. Pallas MPI Benchmarks -PMB, Part MPI-1: [Technical Report]. 2000. http://www.pallas.com

8 Gropp W, Lusk E. An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message-Passing Interface: [Technical Report]. Argonne National Laboratory, USA, 1995

9 Gropp W, Lusk E. MPICH ADI Implementation Reference Manual: [Technical Report]. Argonne National Laboratory, USA, 1995

10 Gropp W, Lusk E. MPICH Working Note: The Second-Generation ADI for the MPICH Implementation of MPI: [Technical Report]. Argonne National Laboratory, USA, 1995

11 Burns G D. The Local Area Multicomputer. In: Proc. of the Fourth Conf. on Hypercube Concurrent Computers and Applications, March 1989

12 Compaq, Intel and Microsoft Corporations. Virtual Interface Architecture Specification. Version 1.0: [Technical Report]. Dec. 1997

13 M-VIA and MVICH, Virtual Interface Architecture Software for Low-Latency, High- Bandwidth, Berkeley Lab VIA Software: [Technical Report]. the Regents of the University of California, 2000

(上接第134页)

种效率和可扩展性较好的算法，引入近似算法，简单而有效解决 IDD 算法中非常重要的候选项目集在各个处理器节点之间的划分问题，尽可能使得各个节点负载平衡。其中，第一种方法为在线算法(online algorithm)，性能比(performance ratio)相对稍差，但是开销非常小，每产生一个候选项目就可以立即处理；第二种方法是离线算法(offline algorithm)，在第一种基础上增加了排序过程，使得性能比大大提高。接下来给出了上述近似算法的性能比的证明。最后，对引入近似算法后的各个算法的复杂度给出了较为详细的分析。

参考文献

1 Stonebraker M, et al. DBMS research at A crossroads: The Vienna update. In: Proc. of the 19th VLDB Conf. Dublin, Ireland, 1993. 688～692

2 Chen M S, Han J, Yu P S. Data mining: An overview from database perspective. IEEE Transactions on knowledge and Data Eng., 1996,8(6):866～833

3 Mannila H, Toivonen H, Verkamo I. Efficient algorithms for discovering association rules. In: AAAI Wkshp. Knowledge Discovery in Databases, 1994

4 Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: ACM SIGMOD Intl. Conf. Management of Data, 1993

5 Park J, Chen M, Yu P S. An effective hash based algorithm for mining association rules. In: ACM SIGMOD Intl. Conf. Management of Data, 1995a

6 Holsheimer M, Kersten M, Mannila H, Toivonen H. a perspective on databases and data mining. In: 1st Intl. Conf. Knowledge Discovery and datamining, 1995

7 Houstma M, Swami A. Set-oriented mining of association rules in relational databases. In: 11th Intl. Conf. Data Engineering, 1994

8 Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proc. of the 20th VLDB Conf. Santiago, Chile, 1994. 487～499

9 Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: Proc. of the 21st VLDB Conf. Zurich, Switzerland, 1995. 432～443

10 Agrawal R, Shafer J. Parallel mining of association rules. IEEE transactions on knowledge and Data Eng., 1996,8(6):962～969

11 Han E, Karypis G, Kumar V. Scalable parallel algorithms for mining association rules. IEEE Transactions on Knowledge and Data Eng., 1999

12 Ausiello G, et al. Complexity and approximation-Combinatorial Optimization Problems and their Appoximability Properties. Springer-Verlag Birlin Herdelberg, 1999