Application-Level Isolation Using Data Inconsistency Detection

Amgad Fayad, Sushil Jajodia, and Catherine D. McCollum

The MITRE Corporation
1820 Dolley Madison Boulevard
McLean, Virginia 22102
{afayad, jajodia, mccollum}@mitre.org

Abstract

Recently, application-level isolation was introduced as an effective means of containing the damage that a suspicious user could inflict on data. In most cases, only a subset of the data items needs to be protected from damage due to the criticality level or integrity requirements of the data items. In such a case, complete isolation of a suspicious user can consume more resources than necessary. This paper proposes partitioning the data items into categories based on their criticality levels and integrity requirements; these categories determine the allowable data flows between trustworthy and suspicious users. An algorithm, that achieves good performance when the number of data items is small, is also provided to detect inconsistencies between suspicious versions of the data and the main version.

1 Introduction

Recently, increasing emphasis has been placed on supplementing protection of networks and information systems with intrusion detection [Lun93, MHL94, LM98], and numerous intrusion detection products have emerged commercially. When a suspicious thread of activity is discovered, the intrusion detection system or the system security officer must decide how to react and whether to allow continued access to the associated user, process, or host. Since the rate of detection (percentage of intrusions that are detected) is directly proportional to the rate of errors (percentage of reported intrusions that are not intrusions), if our goal is to achieve a high rate of detection, we must be prepared to tolerate a high rate of errors.

In [JLM98, LJM99], isolation at the application level was introduced as a means to achieve both a high rate of detection and a low rate of errors. The basic idea is to isolate a suspicious user S transparently into a separate environment that appears to S as the actual system. This approach allows S to be kept under surveillance without risking further harm to the system.

Jajodia, et al. [JLM98], investigated isolation at the database level, and considered the following attack scenario: Suppose a user S of the database comes under suspicion for some reason. To let S continue working and at the same time to isolate the database from any further damage from S, an on-the-fly "separate database version" is created to accommodate accesses by S. All transactions submitted by S are then directed to the separate database version, while transactions from other trustworthy users are applied to the main database version.

Advantages of using isolation at the database level follow: It permits finer grained monitoring of the suspicious user activities, the substitute host is not sacrificed, and the suspicious user S can interact with the system's resources. Therefore, if S proves to be innocent, S has not been subjected to denial of service and any valid results produced by S do not have to be discarded. The drawback of isolation at the database level is that in some cases merging data modified by a suspicious user, after he is found to be legitimate, can be time consuming [JLM98].

This paper's contribution to the database isolation scheme is in two areas. First, in most cases, only a subset of the data items needs to be protected from damage due to their criticality level or integrity requirements. In such a case, complete isolation of a suspicious user can consume more resources than necessary. Therefore, we divide, or categorize, data according to their criticality and

integrity levels. Three categories are proposed: unconstrained, constrained but noncritical, and constrained and critical. These categories determine the allowable data flows between trustworthy and suspicious users.

Second, if a suspicious user is eventually identified as a legitimate user, an algorithm to detect mutual inconsistency is presented. The proposed algorithm has several advantages. First, the amount of work done by this algorithm is a factor of the number of data items accessed. By keeping the constrained but noncritical data items set small, better performance can be achieved. Second, this algorithm reports inconsistency in terms of data items instead of transactions. This process makes it easier for the database administrator to restore the integrity of inconsistent data items by executing appropriate compensating transactions [GMS87]. Third, this algorithm simplifies the case involving multiple suspicious users.

The remainder of the paper is organized as follows. Section 2 describes the categories of data items based on criticality and integrity requirements, and the resulting data flows between the trustworthy and suspicious users. Section 3 discusses the isolation architecture and provides an isolation protocol that preserves the data flows and an algorithm to implement the isolation. Section 4 describes the inconsistency detection algorithm. Section 5 shows examples of how the algorithm works. Section 6 shows the correctness of the algorithm. Section 7 reviews related work. Finally, Section 8 provides conclusions.

2 Categorizing Data and Resulting Data Flows

Traditionally, security technology views each access decision as Boolean: either the user request is granted or it is denied. When suspicious users are involved, the choice is not so straightforward. Certain portions of the database may be highly sensitive (e.g., data which, if revealed to competitors or adversaries, may compromise vital interests of the organization). Other portions of the database may have high integrity requirements and, therefore, any modifications by anyone not trustworthy should be disallowed.

In this paper, we place data items into three separate categories based on their criticality levels and integrity requirements. As shown in Figure 1, data items can be categorized based on their criticality and integrity ratings and suspicious users are allowed/disallowed R/W operations based on those ratings. For integrity reasons, we partition the data items as being either constrained or unconstrained. By definition, *constrained* data items are those to which the integrity constraints must be applied. A data item that is not constrained is said to be *unconstrained*. Based on security considerations, we partition the data items as being either critical or noncritical. We combine the two ratings to obtain the following three categories of data items:

- Unconstrained and noncritical data items – These data items have low criticality and integrity ratings.
- Constrained but noncritical data items – These data items have high integrity ratings; however, they have low criticality ratings.
- Critical data items – All data items that are highly sensitive are considered critical, regardless of their integrity ratings.

In the normal course of events when all users are believed to be trustworthy, each data item has only one version, that is, the main version; any updates by a user are seen by all other users of the database. Once a suspicious user is identified, the above categories determine the allowable data flows between trustworthy and suspicious users.

In general, data flows can be one of the following two types:

- *Disclosing* data flow, which permits data to flow from trustworthy users to suspicious users, or
- *Corrupting* data flow, which allows flow of data from suspicious users to trustworthy users

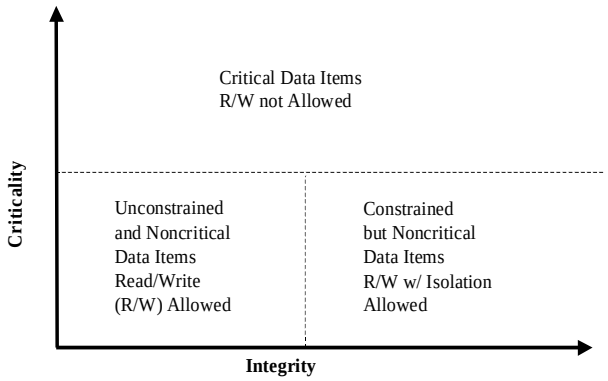

Figure 1. Categories of Data and Resulting Data Flows

A data flow can also be characterized according to its mode:

- *Full* data flow, in which all updates are shared by trustworthy and suspicious users
- *Blocked* data flow, in which updates from suspicious users are not given to trustworthy users but all updates from trustworthy users are given to suspicious users
- *Selective* data flow, in which some updates are shared by trustworthy and suspicious users

On unconstrained and noncritical data items, we allow full data flow that is both disclosing and corrupting. Therefore, there is no isolation on these items (i.e., all updates by trustworthy users are disclosed to suspicious users, and vice versa), and each unconstrained and noncritical data item has only one version.

On critical data items, we allow neither disclosing nor corrupting data flow. This means that suspicious users cannot read or write critical data items. We enforce a selective data flow policy on constrained but noncritical data items: We do not allow disclosing data flow; thus, suspicious users are not given any updates by trustworthy users to such items; suspicious users instead see the values of these items before the isolation was initiated. Trustworthy users, on the other hand, are given versions created by suspicious users; they have the option to select the version they would like to use.

3 Isolation Architecture and Algorithm

The architecture in this paper is similar to the architecture presented in [JLM98]. As shown in Figure 2, the *Intrusion Detector* identifies suspicious users and ultimately determines if a suspicious user is malicious or legitimate. The *Damage Recovery Manager* deals with handling data inconsistencies between the main database and the suspicious versions of the database.

When a user S is identified as being suspicious, S can read or write unconstrained data items and constrained but noncritical data items only. Whenever S modifies a constrained but noncritical data item d, the new value of d is labeled as suspicious (d_s). However, when S modifies an unconstrained and noncritical data item, no special labeling is performed; only the old value of d is replaced by the new value. Critical data items cannot be accessed by S; whenever S tries to access these data items, an error message is generated and given to S.

Only one version of unconstrained data items and critical data items is maintained. This is because changes to unconstrained data items by user S are automatically considered good, and user S is given neither read nor write access to data labeled critical. Only constrained but noncritical items may have multiple versions; exactly how these versions are created is explained in the isolation algorithm below.

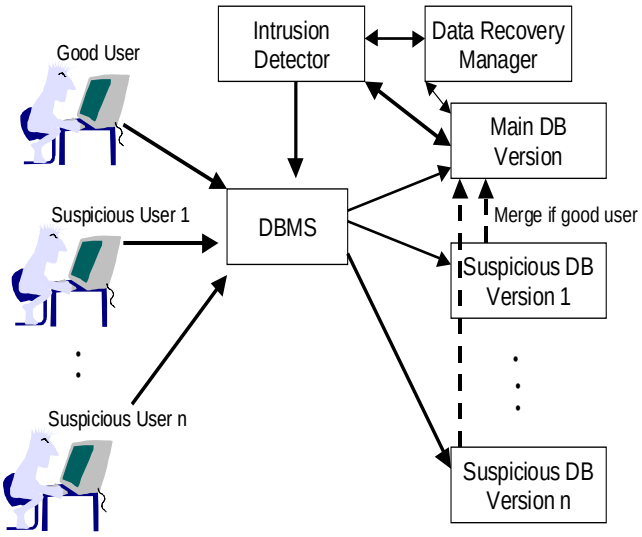

Figure 2. Isolation Architecture

The following list illustrates our isolation protocol, which is adapted from [JLM98].

- At the beginning of the main history, each constrained but noncritical data item x is associated with the same version number MAIN, denoted x[MAIN].
- When constrained but noncritical data item x is read or updated by a trustworthy transaction T, x[MAIN] is given to T, and after x[MAIN] is updated by T, if the version includes other version numbers, then a new version x[MAIN] associated with only one version number MAIN is generated which denotes the value that has been updated by T, and the old value with the same set of version numbers except that MAIN is excluded remains.
- When the first suspicious user S_1 is identified, We attach the specific version number INIT to each constrained but noncritical data item. As a result, a version of x may be associated with several version numbers, for example, x[MAIN][INIT] denotes the value of x which should be the original value of x for each following suspicious database version, and it can be updated by a trustworthy transaction.
- Whenever there are no active suspicious histories in the database, the value of a specific signal variable

RESETABLE is set to be TRUE if it has not been set yet.

- When a suspicious user S_i is identified (including the first one), a unique version number, for example, the time stamp t_i is attached to the INIT version of x. x[INIT][t_i] denotes the value of x which has not been updated by S_i but can be updated by a suspicious transaction submitted by S_i.
- When x is read or updated by a suspicious transaction submitted by S_i, the version of x which includes the version number t_i will be given to S_i. If the version includes the version number INIT, then a new version x[t_i] is generated which denotes the value that has been updated by S_i, and the old value with the same set of version numbers except that t_i is excluded remains.
- When a suspicious user S_i is going to be isolated, if the value RESETABLE is TRUE, then we first set the value RESETABLE to FALSE, and then attach both the INIT and the t_i version number to the current main database version, namely, data items associated with the MAIN version number and remove the old INIT database version.
- When a trustworthy user G_i attempts to read x, G_i is presented with the two copies of x, namely x[INIT] and x[INIT][t_i]. If G_i decides to use x[INIT[t_i], then a separate copy of x is no longer maintained. If G_i, on the other hand, decides to use the original value of x, then two copies of x are maintained.

The following terminology is used in the isolation algorithm. A data item d_i *precedes* data item d_j if d_i is written by a transaction which precedes a transaction that reads d_i and d_j or data item d_i is read by the same transaction which writes d_i and d_j. Data items d_i and d_j are *inconsistent* if d_i precedes d_j and vice versa. The *precedence-set* of a data item is the set of all data items which precede it. The *link set* of a data item d_i is the set of all data items that were written by the last transaction which wrote d_i. Throughout we assume that the write-set of each transaction is always contained in the read-set.

Procedure Isolation

```
begin
  for each Constrained & Noncritical (C&N)
  data item d do begin
     /* initialize the precedence/link-sets */
     link-set(d) = { };
     precedence-set(d) = { };
  end
  for each transaction T do begin
     if T is a trustworthy user transaction then
        apply the following to trustworthy
        user precedence/link-sets
     else
        apply the following to the corresponding
        suspicious user precedence/link-sets
  if accessed data item is critical and user is suspicious
        reject transaction
  if accessed data item is unconstrained & noncritical
        allow read/write w/o precedence/link-set updates
  if read-set(T) ∩ write-set(T) <> {} begin
        for all C&N d in write-set(T) do begin
           precedence-set(d) = precedence-set(d) ∪
                                    read-set(T);
        end
  end
  for all C&N d in read-set(T) do begin
        if link-set(d) ∩ read-set(T) <> {}begin
           for all C&N data items f in read-set(T)
          do begin
              precedence-set(f) = precedence-set(f) ∪
                                    link-set(d);
           end
        end
  end
  for all C&N d in write-set(T) do begin
        link-set(d) = write-set(T);
  end
end.
```

Figure 3. Isolation Algorithm

The algorithm maintains with each data item several pairs of sets, one pair corresponding to the trustworthy users and one pair corresponding to each suspicious user. Each pair consists of a precedence-set and a link-set. These sets are initialized to be empty. Whenever a transaction T is executed by a user, the corresponding pair is updated as follows: If the intersection of the read-set of

T and the write-set of T is nonempty, then append the read-set of T to the precedence-set of each data item in the write-set of T. For each data item d in the read-set of T, if the intersection of the link-set of d with the read-set of T is nonempty, then append the link-set of d to the precedence-set of every data item in the read-set of T.

Copy the write-set of T to the link-set of each data item in the write-set of T.

In addition, the algorithm allows trustworthy users to select between suspicious values of data and the original value. If a trustworthy user G attempts to access a data item d which was modified by a suspicious user to the value d_s. G will be prompted to select between d and d_s. If G chooses d, then two versions of d will be maintained. Otherwise, one version of d will be maintained.

When a suspicious user accesses unconstrained and noncritical data items, the precedence/link-sets are not updated with these items. This is because only one version of unconstrained and noncritical data is maintained. If the suspicious user attempts to access critical data, the transaction will be rejected.

A more formal description of the isolation algorithm is given in Figure 3.

4 Data Inconsistency Detection Algorithm

When a decision is reached and the suspicious user S is found to be malicious, the corresponding suspicious database version is simply discarded. However, if S is found to be innocent, updates made by S must be merged with the main database version; any conflicting updates must be identified so that conflicts can be resolved.

Before we look for inconsistencies, we compute the transitive closure of each precedence set using Warshall's algorithm (e.g., see [Baa88]). Combine the trustworthy user precedence-sets with the suspicious user precedence-sets by computing the set unions.

Now we look for inconsistencies by performing the following: For each d_j and d_i, if dj is in precedence-set(d_i) AND d_i is in precedence-set(d_j), then we report that d_i and d_j are inconsistent. The system administrator must resolve all data items that cause inconsistencies (based on the algorithm).

A more formal description of the algorithm is given in Figure 4. Conceptually, this algorithm utilizes a precedence graph G consisting of data items as nodes, instead of a precedence graph that consists of transactions as nodes (as in [JLM98]). We construct a precedence graph G based on the computed precedence-sets. For data items d_j and d_i, we place a (directed) *precedence edge* from d_j to d_i if d_j precedes d_i. Cycles in the graph G indicate data inconsistency.

Note that in the event that multiple suspicious users are present, detection of inconsistency can be performed by computing the union of all the precedence-sets for each data item [Ram89].

Procedure Check Inconsistencies

```
begin
      apply Warshall's algorithm to compute
      transitive closure on precedence-sets;
for each data item d do
        full-precedence-set(d) =
           trustworthy-precedence-set(d) ∪
           suspicious-precedence-set(d);
      end
      for each data item d do
         for each f in precedence-set(d) do
            if d is in precedence-set(f) then
            /* inconsistency detected! */
               report d and f
    end
end.
```

Figure 4. Algorithm for Detecting Inconsistencies

5 Examples

In this section, we give two examples to illustrate how the algorithm works. For simplicity, we assume that all data items that are being accessed are constrained but noncritical.

Example 1. Consider the five transactions given below:

$READSET(T_{s1}) = WRITESET(T_{s1}) = \{ d_1, d_2 \}$
$READSET(T_{s2}) = \{ d_2, d_3 \}, WRITESET(T_{s2}) = \{ d_3 \}$
$READSET(T_{s3}) = \{ d_3, d_4, d_5 \}, WRITESET(T_{s3}) = \{ d_4 \}$
$READSET(T_{t1}) = WRITESET(T_{t1}) = \{ d_5 \}$
$READSET(T_{t2}) = \{ d_1, d_5 \}, WRITESET(T_{t2}) = \{ \}$

Here the subscripts s and t are used to denote the transactions executed by the suspicious and trustworthy users, respectively. Suppose the suspicious and trustworthy users execute transactions in the following order:

$$\mathbf{H} = T_{s1}\ T_{t1}\ T_{s2}\ T_{s3}\ T_{t2}$$

Our algorithm will generate two sets of precedence sets, one involving the suspicious transactions and the

other involving trustworthy transactions, as shown in Table 1. Since the precedence set $P(d_5)$ in the first column includes d_1 and $P(d_1)$ in the second column includes d_5, we conclude that both d_1 and d_5 are inconsistent.

Table 1. Precedence Sets for Example 1

Suspicious Precedence Sets	Trustworthy Precedence Sets
$P(d_1) = \{ d_1, d_2 \}$	$P(d_1) = \{ d_5 \}$
$P(d_2) = \{ d_1, d_2 \}$	$P(d_2) = \{ \}$
$P(d_3) = \{ d_1, d_2, d_3 \}$	$P(d_3) = \{ \}$
$P(d_4) = \{ d_1, d_2, d_3, d_4, d_5 \}$	$P(d_4) = \{ \}$
$P(d_5) = \{ d_1, d_2, d_3 \}$	$P(d_5) = \{ d_5 \}$

Example 2. If we change the READSET(T_{s3}) to $\{d_3,d_4\}$, Table 2 shows the precedence sets for all the data items involved in the transactions. Unlike the previous example, no inconsistencies are indicated.

Table 2. Precedence Sets for Example 2

Suspicious Precedence Sets	Trustworthy Precedence Sets
$P(d_1) = \{ d_1, d_2 \}$	$P(d_1) = \{ d_5 \}$
$P(d_2) = \{ d_1, d_2 \}$	$P(d_2) = \{ \}$
$P(d_3) = \{ d_1, d_2, d_3 \}$	$P(d_3) = \{ \}$
$P(d_4) = \{ d_1, d_2, d_3, d_4 \}$	$P(d_4) = \{ \}$
$P(d_5) = \{ \}$	$P(d_5) = \{ \}$

6 Correctness of the Data Inconsistency Detection Algorithm

Before we can give a proof of the correctness of the algorithm, we need to define the necessary terms.

Definition 1 [Ram89]. For data items d_i and d_j, we say that d_i *immediately precedes* d_j if one of the following items is true:

1) d_i is in the read-set and d_j is in the write-set of a transaction

2) The following conditions hold in a schedule S: there exist T_{1p}, T_{1q}, $p \leq q$, in S such that

a) T_{1p} updates d_i

b) T_{1q} reads d_j, and

a) There is a g in read-set(T_{1q}) intersection write-set(T_{1p}) and there is no $p<u<q$ such that g is in write-set(T_{1u}).

Definition 2 [Ram89]. The relation *precedes* is defined on the data items as the transitive closure of the relation *immediately precedes*.

Definition 3. For each data item d, we define the *precedence-set P(d)* as the set of all data items that *precede* d.

Definition 4. For each data item d, we define the *link-set(d)* as the set of data items in the write-set of the transaction that has most recently updated d and committed after isolation occurs.

The following theorem shows the correctness of the algorithm:

Theorem 1. After each transaction T, procedure *Isolation* updates the precedence-set of each data item correctly. That is, all data items, which immediately precede d, are in its precedence-set.

Proof. We provide a proof using induction on the number of transactions. If T is the first transaction, it is easy to see that the precedence-sets are up to date. Now we assume that after the k^{th} transaction, the precedence-sets are up to date. We must show that after transaction k+1, the precedence-sets remain up to date. Now, let d_i be a data item. It easy to see from the definition that the algorithm adds all data items which immediately precede d_i which resulted from the $k+1^{st}$ transaction. Also, we claim that the link-sets are kept up to date. This follows from the definition of a link-set. QED

Given the correctness of the isolation algorithm and the correctness of Warshall's transitive closure algorithm, all data items, which precede data item d_i, are in its precedence-set. This ensures that the check inconsistencies algorithm detects all data items in need of correction.

7 Related Work

McDermott and Goldschlag [MG96a, MG96b] initiated this line of research with their work on data jamming. They identify several techniques to detect suspicious users.

Two papers directly relate to this work. Jajodia, et al. [JLM98], propose application-level isolation as a security mechanism to increase the security of information systems vulnerable to authorized malicious users. They develop an isolation scheme in the database context that isolates databases from any further damage caused by malicious users.

The current paper is different from [JLM98] in two respects. First, in this paper the database is partitioned into different categories, and these categories influence the data flows between suspicious and trustworthy users. Second, the way conflicts are identified during the merge of the suspicious versions and the main database version is very different. In [JLM98], the merge algorithm identifies all transactions that violate the serializability requirement. In contrast, the merge algorithm in this paper identifies all data items that may be inconsistent. We believe that it much easier for database administrators to take some compensating actions [GMS87] to restore the integrity of inconsistent data items than to try to identify the inconsistency by examining the cycles involving multiple transactions.

In [LJM99], Liu, et al., develop a probabilistic model to argue that intrusion confinement can be effectively used to resolve the conflicting design goals of an intrusion detection system by achieving both a high rate of detection and a low rate of errors. It is also shown that as a second level of protection in addition to access control intrusion confinement can dramatically enhance the security (especially integrity and availability) of a system in many situations. [LJM99] presents a concrete isolation scheme for file systems.

8 Conclusions

This paper expands the research on the concept of isolating suspicious users at the application level. First, we proposed categorizing data based on criticality and integrity requirements. Then we proposed an isolation scheme, which operates on only one of the categories proposed, namely constrained but noncritical data items. Finally we presented an algorithm which can efficiently detect inconsistencies in data items when suspicious users are present. Based on this paper, the following three conclusions apply: First, the complexity of the algorithm we presented is a factor of the number of data items accessed. Second, a small constrained but noncritical data set can help keep the overhead of this algorithm small. Finally, a potential disadvantage of this scheme is that resolving inconsistencies in data requires manual intervention.

Acknowledgement

This work was funded by Air Force Research Laboratory/Rome. We are grateful to Joe Giordano for his support.

References

[Baa88] S. Baase. "*Computer Algorithms*." Addison-Wesley, Reading, MA, 1988.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. "*Concurrency Control and Recovery in Database Systems*." Addison-Wesley, Reading, MA, 1987.

[Dav84] Susan Davidson. "Optimism and Consistency in Partitioned Distributed Database Systems." *ACM transactions on database systems*, 9(3):456-481. September 1984.

[GMS87] Hector Garcia-Molina and Ken Salem. Sagas. *Proc. International Conf. on Management of Data*, pages 249-259, San Francisco, CA, 1987.

[JLM98] Sushil Jajodia, Peng Liu, and Catherine D. McCollum. "Application-Level Isolation To Cope with Malicious Database Users." In *Proc. 14th Annual Computer Security Applications Conference*, pages 73-82, Phoenix, AZ, December 1998.

[LJM99] Peng Liu, Sushil Jajodia and Catherine D. McCollum. "Intrusion Confinement by Isolation in Information Systems." In *Proc. XIII Annual IFIP WG 11.3 Working Conf. on Database Security*, Seattle, WA, July 1999.

[Lun93] Teresa Lunt. "A Survey on Intrusion Detection Techniques. "*Computers & Security*, 12(4):405-418, June 1993.

[LM98] Teresa Lunt and Catherine D. McCollum, "*Intrusion Detection and Response Research at DARPA,"* Technical Report, The MITRE Corporation, 1998.

[MG96a] J. McDermott, and D. Goldschlag. "storage jamming." In D.L. Spooner, S.A. Demurjian, and J.E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365-381. Chapman & Hall, London, 1996.

[MG96b] J. McDermott, and D. Goldschlag. "Towards a model of storage jamming." In *Proceedings of the IEEE Computer*

Security Foundations Workshop, pages 176-185. Kenmare, Ireland, June 1996.

[MHL94] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. "Network intrusion detection." *IEEE Network*, pages 26-41, June 1994.

[Ram89] K. V. S. Ramarao. "Detection of Mutual Inconsistency in Distributed Databases." *Journal of Parallel and Distributed Computing*, 6:498-514,1989.