Data Dependence Analysis for an Untrusted Transaction Manager

Myong H. Kang
Naval Research Laboratory
Code 5542
Washington, D.C. 20375

Abstract

There are two components in the scheduler for multilevel-secure databases which use the replicated architecture; global and local schedulers. Since the global scheduler, which enforces data consistency among replicas, has to make scheduling decisions based on transactions (i.e., without any knowledge of actual data or physical layout of data), an accurate analysis technique which can detect conflicts among queries is needed. The data dependence analysis introduced here provides a method for precisely determining whether the portions of relations affected by various database operations overlap without the knowledge of actual data.

1. Introduction

There are many approaches for multilevel database systems which protect classified information from unauthorized users based on the classification of the data and the clearances of the users [3, 9]. One approach, which is called the replicated architecture approach [4], uses a physically distinct backend database management system for each security level. Each backend database contains information at a given security level and all data at lower security levels. The system security is assured by a trusted frontend which permits a user access to only the backend database system which matches his/her security level.

The SINTRA[1] database system, which is currently being prototyped at Naval Research Laboratory, is a multilevel trusted database management system based on this replicated architecture. The replicated architecture system contains a separate database system for each security level and few interfaces among different database systems. The database at each security level contains data at the current security level and replicated data from lower security levels.

At first glance, a database mangement system for each security level may seem excessive. However, we think this approach has the following merits:

- The security policy can be easily enforced by carefully designing interfaces among different database systems.
- Development cost can be reduced because commercial database systems for backend computers are widely available.
- The amount of trusted software can be minimized.
- Performance can be improved by using optimization and parallelization techniques which have been developed for conventional databases. This is the case because the replicated architecture uses conventional database systems as backend database systems, and uniprocessor or multiprocessor computers can be chosen as backend computers without affecting the security policy.

The SINTRA database system consists of one trusted front end (TFE) and several untrusted backend database systems (UBD). The role of a TFE includes user authentication, directing user queries to the backend, maintaining data consistency among backends, etc. Each UBD can be any commercial off-the-shelf database system. Currently, we are using Honeywell XTS-200 system as a trusted frontend and ORACLE databases which are running on SUN4/300 as backend databases. The backend and frontend computers are connected through Ethernet. Figure 1 illustrates the SINTRA architecture.

---

1. Secure INformation Through Replicated Architecture

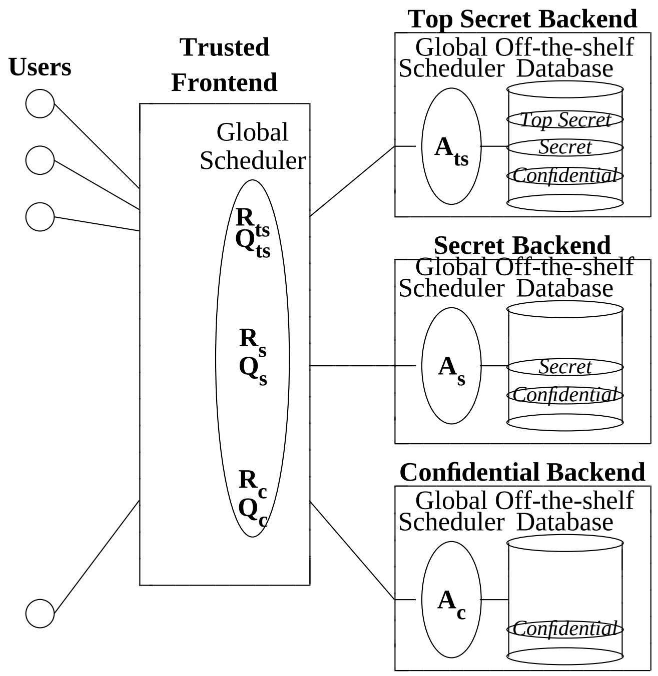

Figure 1: The SINTRA Architecture.

Concurrency Control Problem in the SINTRA system

Since each UBD in a replicated architecture contains data from lower levels, update queries have to be propagated to higher security level databases. If this propagation of update queries is not carefully controlled, inconsistent database states among backend databases can be created. It turned out that even the order of non-conflicting transactions which is determined at lower security level must be preserved at higher security level [7] to preserve *one-copy serializability* (1SR).

Consider the security lattice in figure 2, and two non-conflicting **L**-level transactions T_i and T_j. Also consider an **M1**-level transaction T_u, and an **M2**-level transaction T_v. Let's further assume that T_u conflicts with T_i and T_j, and T_v conflicts with T_i and T_j. Since two transactions, T_i and T_j, are not conflicting and our security model does not allow *write-down*, an execution order $<T_i, T_u, T_j>$ at security class **M1** and an execution order $<T_j, T_v, T_i>$ at security class **M2** will generate the same result on *replicas* of security class **L** data. However, the reversed order between T_i and T_j at security classes **M1** and **M2** will create confusion. Specifically, at security class **H**, a consistent ordering among T_i, T_j, T_u, and T_v cannot be determined then *1SR* will be violated. Consequently, any global scheduler which does not enforce the same ordering among transactions at each relevant UBD may fail to produce consistent schedules. Thus any algorithm which gives 1SR schedules must preserve the orderings which is determined at lower levels.

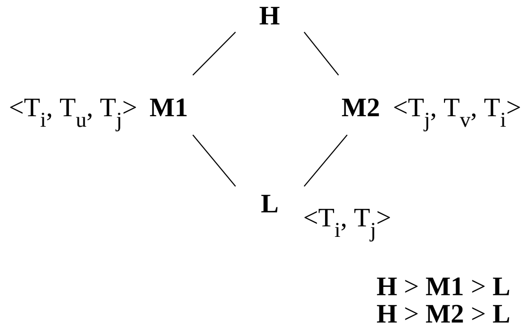

Figure 2: A security lattice

The scheduler for the SINTRA architecture has two components; global and local schedulers. The local schedulers, which are the concurrency controllers of the off-the-shelf database systems, enforce serializability among transactions which are submitted to backend database systems. On the other hand, the global scheduler enforces data consistency among the UBDs. To preserve 1SR, the global scheduler of the SINTRA system has to pass the information on serialization order from the lower security level to higher security level UBDs, and that serialization order has to be maintained. One way to achieve this goal is:

(1) When update transactions are propagated to higher security levels, this propagation order must be the same as the serialization order.

(2) When the update transactions from the lower security level arrive, submit one transaction and wait until that transaction is committed at the backend, and then submit next transaction.

This serial execution is necessary since most of the off-the-shelf database systems do not guarantee that the serialization order of transactions is the same as the submission order.

An alternative approach of the SINTRA global scheduler (whose detailed description appears in [7]) may replace step (2) to the followings:

(2.1) Submit update transactions from lower levels to the backend as they arrive.

(2.2) If update transactions from lower levels are serialized in the same order as they are submitted to UBD then commit those and propagate them up (i.e., the propagation order is the same as the serialization order).

(2.3) Otherwise test if there is a conflict among transactions whose serialization order is different from the submission order.

[a] If there is no conflict among these out-of-ordered transactions then commit them, and rearranges those transactions so that the submission order is preserved before the global scheduler propagates them up (i.e., since only the order of transactions which do not conflict has been changed, the results of execution will not be altered).

[b] Otherwise roll back and re-submitt the part of these transactions.

It is clear from the above that the global scheduler needs a good tool to detect conflicts.

The local scheduler typically uses locks or timestamps based on the knowledge of actual data or physical layout of the data in each UBD. The basic units for locks or timestamps may be relations, fixed-sized pages, or tuples depending on the granularity of local schedulers. However, the global scheduler has very little knowledge about the behavior of the local scheduler or the physical layout of data. For example, the global concurrency controller has no knowledge about where a specific tuple is located or which physical page should be locked. Sometimes the tuples which will be modified are unknown until the computation based on existing data is completed. The above factors may force the global concurrency controller to use relations as basic units to detect conflicts among transactions. Such a scheduler will be too restrictive and inefficient because it ignores the fact that referencing only a few tuples or few attributes of a relation is not the same as referencing the entire relation.

In this paper, we introduce the data dependence analysis which can detect conflicts without any knowledge of actual data or physical layout of data on the backend. The primary goal of data dependence analysis is to precisely determine if the portions of relations affected by various operations overlap. If there is no overlap, the query processing will permit the operations to be executed out of sequence, or even in parallel, without altering the results. A similar concept has appeared in database concurrency control literature. What dependence analysis traditionally defines as any of several different types of *dependence* corresponds to the concept of a *conflict* for databases.

This paper is organized as follows. Section 2 discusses transaction model for the SINTRA system. The data dependence analysis of database queries which is used by the global scheduler is presented in section 3. Finally, section 4 summarizes the contributions of this paper.

2. Transaction Model

We adopt a layered model of transactions, where a transaction is a sequence of queries, and each query can be considered as a sequence of reads and writes. For example, `replace` and `delete` queries can be viewed as a read operation followed by a write operation, `insert` can be viewed as a write operation, and `retrieve` can be viewed as a read operation.

Definition 1.

A **transaction** T_i is a sequence of queries, i.e., T_i = $<q_{i1}, q_{i2}, ..., q_{in}>$. Each query, q_{ij}, is an atomic operation and is one of `retrieve, insert, replace,` or `delete`.

To model the propagation of updates produced by a given transaction to higher security level databases, *update projection* is defined.

Definition 2.

An **update projection** U_i, which corresponds to a transaction T_i, is a sequence of update queries, e.g., $U_i = <q_{i2}, q_{i5}, ..., q_{in}>$ obtained from transaction T_i by simply removing all `retrieve` queries.

Note that unlike interactive user transactions, a complete query sequence of an update projection is always known to the global scheduler.

3. Data Dependence Analysis

The primary difference between traditional dependence analysis [8, 10, 12] and the scheme that we propose lies in the need to directly analyze properties of a relation which are affected by operations. The primary goal of data dependence analysis is to precisely determine if the portions of relations affected by various operations overlap. Clearly, operations on different relations do not overlap. Even when operations access different fields of the same tuples of a relation, sometimes there may be no overlap and hence no dependence.

Before we present the detail analysis, we present three basic types of dependence. In our examples, the following two relations constitute the database:

```
EMP(name, position, salary, dept)
PRODUCT(item, price, dept)
```

3.1. Basic Dependence Analysis

In this section, we describe the basic data dependence analysis of database queries treating each relation as a single entity. Three basic types of data dependence (true, anti, and output dependences) have been used [8, 10] to describe the properties of data references in conventional language programs. These types of data dependence can also be applied to describe references to relations within database queries. Consider the following three queries:

```
Q1: delete(EMP, EMP.dept = 'appliance')
Q2: retrieve(EMP.all, EMP.salary > 50)
Q3: replace(EMP, EMP.salary < 20,
         EMP.salary = EMP.salary * 1.1)
```

Q1 and Q2 cannot be executed at the same time since Q2 uses the relation EMP which is modified by Q1. This is called **true dependence** or *flow dependence* (similar to write-read conflict). Q3 also depends on Q1; thus Q1 must be executed before both Q2 and Q3. Q1 and Q2 use the relation EMP before Q3 modifies it. Since Q1 and Q2 use the ''old'' values of relation EMP, these must be executed before Q3; this is called **anti-dependence** (a read-write conflict). The third kind of dependence is shown between Q1 and Q3. Q1 modifies relation EMP and then Q3 modifies the same relation. If Q1 and Q3 are executed at the same time, there may be an uncommitted dependency problem [2] caused by the asynchronous update of the EMP relation. This is called **output dependence** (a write-write conflict).

Since the basic dependence analysis treats each relation as a single entity, it is efficient but not precise. Hence, the basic dependence analysis can be treated as a filter. Alternatively, if the basic dependence analysis determines that there is no dependence between two queries then no additional analysis is required. However, if the basic dependence analysis suggests that there may be dependences between two queries, then further analysis, which will be described in the following section, may be applied.

3.2. Advanced Dependence Analysis

The dependence analysis presented in this section attempts to precisely describe the vertical and horizontal portions of relations which are accessed in each query. We consider the queries that have condition predicates which is Boolean combinations of selection and join conditions.

3.2.1. Notation and Basic Concepts

Each query contains a condition predicate (clause) which is a Boolean combination of atomic conditions. Each atomic condition may specify a selection or a join operation, and each condition is connected by connectives $\in \{\wedge, \vee\}$. The Boolean operator ''not'' is not considered in this paper because a predicate α which contains ''not'' operators can be converted into an equivalent predicate α' which does not contain ''not'' operator in polynomial time [5].

Rosenkrantz and Hunt [11] developed an algorithm which can determine the satisfiability of restricted class of conjunctive Boolean expressions in polynomial time. In this restricted class, an atomic condition must be of the form $x \; op \; c$, $x \; op \; y$, or $x \; op \; y + c$, where c is a constant, x and y are attributes of relation(s), and $op \in \{=, <, \leq, >, \geq\}$. In this paper, conjunctive predicates are assumed to be in this restricted class. If there exists a condition of the form $x \neq y + c$, then this must be converted to $(x < y + c) \vee (x > y + c)$.

In this paper, A, B, C, D, and E are used to represent an atomic condition such as EMP.dept = PRODUCT.dept, and α, β, and γ are used to represent the conjunctive predicates such as EMP.salary > 50 $\wedge$ EMP.dept = 'Business'.

In data dependence analysis, we use a functional notation to represent high-level queries. The general form is O(R, P, V) where O specifies an operation such as append, delete, replace, retrieve on relation in R; P is a predicate over relation in R; and V is attribute value assignments which specify how attributes of relation in R are replaced. The intended semantics of operation is: if O is append or delete, then V is empty and all tuples satisfying P are inserted or deleted in R; if O is replace, then V is not empty, all tuples in relation R satisfying P are modified; if O is retrieve then V is empty and R specifies set of attributes which should be retrieved from the tuples which satisfy P.

Note that a relation which appears in R will be modified (write action) if O is an update operation, and relations which appear in P will be used (read action) to find the tuples that will be either updated or retrieved. If O is either delete or replace, the same relation should appear in both R and P. Therefore, both a read and a write will be done to the relation in R. However, if O is append then tuple(s) which does not exist in relation R is added to R. Hence, append is a write-only action as far as the relation R is concerned.

Before the advanced data dependence analysis is performed, four sets of the database will be defined for each query.

Definition 4.

- **Vertical read set (VRS)** of a given operation is the set of attributes whose values may be examined by the operation.

- **Vertical write set (VWS)** of a given operation is the set of attributes whose values may be changed by the operation.
- **Horizontal read set (HRS)** of a given operation is the set of tuples from which some attribute value(s) may be examined by the operation.
- **Horizontal write set (HWS)** of a given operation is a the set of tuples in which some attribute value(s) may be changed (or created) by the operation.

All attributes which appear in `P` belong to VRS of the operation. If `O` is `retrieve` operation, all attributes which appear in `R` also belong to VRS; if `O` is `replace` then all attributes which appear at the right hand side of assignments in `V` also belong to VRS. If `O` is either `delete` or `append`, then all attributes of the relation, which appears in `R`, belong to VWS; if `O` is `replace` then only attributes which appear at the left hand side of assignments in `V` belong to VWS.

Horizontal sets are generally denoted as $\{t_R \mid \alpha\}$, representing the set of tuples in relation `R` which satisfy the condition α.

Now we define two types of conflicts.

Definition 5.

- Two queries **conflict vertically** if vertical sets from two queries are not disjoint and at least one of them is a write set.
- Two queries **conflict horizontally** if horizontal sets from two queries are not disjoint and at least one of them is a write set.

If two queries conflict both vertically and horizontally then there is some type of *dependence* between these two queries. If there is either no conflict or only one type of conflict then there is no dependence — which implies that the order of execution of the queries will not affect the results obtained.

In addition, we consider the relationship between predicates.

Definition 6.

A predicate α is **independent** of predicate β in relation R *iff* two sets $\{t_R \mid \alpha\}$ and $\{t_R \mid \beta\}$ are disjoint (i.e., $\{t_R \mid \alpha\} \, || \, \{t_R \mid \beta\}$).

Note that the definition of independence between α and β is similar to $(\alpha \wedge \beta)$ is unsatisfiable in [11]. However, there is an important difference between these two definitions. For instance, let $\alpha \equiv (R.a \leq 30 \wedge R.b = S.c \wedge S.d \leq 3000)$ and $\beta \equiv (R.a \leq 40 \wedge R.b = S.c \wedge S.d \geq 6000)$. $(\alpha \wedge \beta)$ is unsatisfiable according to [11] and α is independent of β in relation S. However, α is not independent of β in relation R [6].

To demonstrate the application of the above concepts, consider the queries:

```
Q1: replace(EMP, EMP.name = 'Lisa',
            EMP.position  = 'Manager')
Q2: replace(EMP, EMP.salary < 45,
        EMP.salary = EMP.salary * 1.07)
```

The first query, `Q1`, generates the sets:

VRS ≡ {`EMP.name`}

VWS ≡ {`EMP.position`}

HRS ≡ {t_{EMP}| `EMP.name = 'Lisa'`}

HWS ≡ {t_{EMP}| `EMP.name = 'Lisa'`}

The second query, `Q2`, generates the sets:

VRS ≡ {`EMP.salary`}

VWS ≡ {`EMP.salary`}

HRS ≡ {t_{EMP}| `EMP.salary < 45`}

HWS ≡ {t_{EMP}| `EMP.salary < 45`}

When the regions of the intersection of four sets from the first query, `Q1`, and the intersection of four sets from the second query, `Q2`, are displayed in the same table (figure 3), we can easily see that these regions do not overlap.

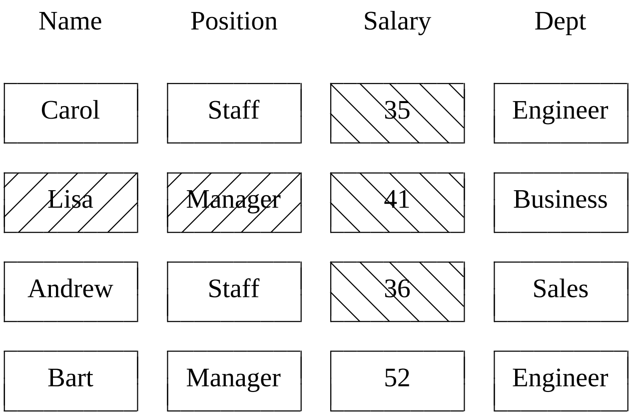

Figure 3: Independence of Sample Queries

In the following two sections, we discuss the method to make use of the above information.

3.2.2. Use of Informations from Horizontal Sets

In this section, we describe how potential dependence relationships from basic dependence analysis can be removed using informations from horizontal sets. The method for removing potential dependence relationships using informations from vertical sets is described in next section.

If there is no dependence between two queries, Q1 and Q2, the execution order will not affect the result. Therefore, before removing potential dependences between queries, it should be clear that the execution order of both <Q1, Q2> and <Q2, Q1> produce the same result. In this section, we use an independence test to find if two predicates are independent in a specific relation. The algorithm to test the independence of given predicates is presented in [6].

Append Query

The append query adds new tuples to a relation. If append follows another query, Q2, Q2 cannot access new tuples which will be added by append. Hence, if an execution order <Q1, Q2>, where Q1 is append, accesses disjoint horizontal sets, then another execution order <Q2, Q1> will also access disjoint horizontal sets. Consider the following 2 queries:

```
Q1: append(EMP, ('John', 'trainee', 25,
                                'business'))
Q2: delete(EMP, EMP.dept = 'appliance')
```

The basic dependence analysis suggests that there may be true and output dependences between Q1 and Q2. However, when the independence property is tested after the appended tuple is converted to predicate form, we find that there is no such dependence between Q1 and Q2. This is because (EMP.name = 'John' /\ EMP.position = 'trainee' /\ EMP.salary = 25 /\ EMP.dept = 'business') from Q1 and (EMP.dept = 'appliance') from Q2 are independent in EMP.

Delete Query

The delete query removes tuples from a relation. Hence, the set of tuples in the relation after the delete operation will be a subset of the set of tuples in that relation before the delete operation. Therefore, if an execution order <Q1, Q2>, where Q2 is a delete query, accesses different horizontal sets, then another execution order <Q2, Q1> will also access different horizontal sets. Consider the following two queries:

```
Q1: retrieve(EMP.all, EMP.salary < 30
          /\ EMP.position = 'programmer')
Q2: delete(EMP, EMP.salary > 70 /\
             PRODUCT.item = 'HDTV' /\
              EMP.dept = PRODUCT.dept)
```

Since two of the predicates from Q1 and Q2 are independent in EMP, there is no dependence between queries Q1 and Q2.

Replace Query

Resolving the dependence relationship between queries, where one of them is replace is more complicated because a modified tuple may be either examined or changed by the other query. Consider the following queries:

```
Q1: replace(EMP, EMP.salary ≥ 35,
         EMP.salary = EMP.salary * 0.8)
Q2: retrieve(EMP.all, EMP.salary < 30 /\
          EMP.position = 'programmer')
```

Even though EMP.salary ≥ 35 from Q1 is independent of EMP.salary < 30 /\ EMP.position = 'programmer' from Q2 in EMP, there is a dependence relationship between two queries. Consider a tuple $t \equiv$ ('John', 'engineer', 35, 'engineer'). Since t satisfies the condition of Q1, t will be replaced by $t' \equiv$ ('John', 'engineer', 28, 'engineer'). Since t' satisfies the condition of Q2, it will be retrieved. Hence, there exists dependence between Q1 and Q2 because two queries may access the same data.

Therefore, if replace query is involved in a dependence relationship, independence test may be required both before and after the modification. Consider a sequence of two queries $Q1 \equiv O_1(R_1, P_1, V_1)$ and $Q2 \equiv O_2(R_2, P_2, V_2)$. Let $V_j \equiv \{v_1, \ldots, v_n\}$ where j is either 1 or 2, v_i is an assignment to an attribute in relation R_j and n is less than or equal to the number of attributes in R_j. Let $P_i(V_j)$ be the new predicate which replaces the left hand side of the replace assignment which appears in P_i with right hand side of replace assignment in V_j. For example, if $P_i \equiv$ (EMP.salary < 30 /\ EMP.position = 'programmer') and $V_j \equiv$ (EMP.salary = EMP.salary * 0.8), then $P_i(V_j) \equiv$ (EMP.salary * 0.8 < 30 /\ EMP.position = 'programmer').

If there is only true dependence between Q1 and Q2, and O_1 is replace then relation R_1 will be updated by replace. Therefore, if P_1 is independent of P_2 in R_1, and P_1 is independent of $P_2(V_1)$ in R_1 then there is no dependence between Q1 and Q2.

If there is only anti-dependence between Q1 and Q2, and O_2 is replace then relation R_2 will be updated by replace. Therefore, if P_1 is independent of P_2 in R_2, and P_2 is independent of $P_1(V_2)$ in R_2

then there is no dependence between above two queries.

If there exist both true dependence and anti-dependence between Q1 and Q2, and O_1 and O_2 are both `replace` then the following 4 tests:

a) P_1 is independent of P_2 in R_1,

b) P_1 is independent of $P_2(V_1)$ in R_1,

c) P_1 is independent of P_2 in R_2, and

d) P_2 is independent of $P_1(V_2)$ in R_2

must be true. However, if R_1 and R_2 are the same relation (i.e., output dependence exists), then test (c) can be omitted because (a) and (c) are the same tests.

3.2.3. Use of Informations from Vertical Sets

Now we show how VRS and VWS can be used to resolve potential dependence relationships which are suggested by the basic dependence analysis. Consider the following example:

```
Q1: retrieve((EMP.name, EMP.dept),
              EMP.position = 'manager')
Q2: replace(EMP, EMP.salary < 20,
               salary = salary * 1.1)
```

In the above example, VRS of Q1 is {EMP.name, EMP.dept, EMP.position} and VWS of Q2 is {EMP.salary}. Since VRS of Q1 and VWS of Q2 are disjoint sets, there is no dependence between two queries.

Note that resolving potential dependence relationship using vertical set is limited among `retrieve` and `replace` queries because, in general, `append` and `delete` affect all attributes of a relation.

3.2.4. Dependence Analysis Algorithm

The basic dependence analysis reveals potential dependence relationships among queries. The following algorithm summarizes the advanced dependence analysis to obtaining precise dependence relationships between two queries.

Algorithm: DEPENDENCE ANALYSIS

Input:

a sequence of two queries $O_i(R_i, P_i, V_i)$ and $O_j(R_j, P_j, V_j)$ which may have dependence relationship(s), and type of dependence relationship(s) from basic dependence analysis.

Output:

''YES'' if dependence relationship exist,

''NO'' if dependence relationship does not exist

Comment:

P_i and P_j are assumed to be conjunctive predicates. Also append query is expected to pass predicate instead of appended tuple(s).

Method:

if *two queries have non conflicting vertical sets* **then**
  return*(NO);*
if *output dependence exists* **then**
  if $DEPEND(P_i, P_j, R_i) = YES$ **then**
    return*(YES);*
else
  if *true dependence exists* **then**
    if $DEPEND(P_i, P_j, R_i) = YES$ **then**
      return*(YES);*
  if *anti-dependence exists* **then**
    if $DEPEND(P_i, P_j, R_j) = YES$ **then**
      return*(YES);*
if O_i *is append* **or** O_j *is append* **then**
  return*(NO);*
if O_i *is replace* **and** *true dependence exists* **then**
  if $P_j = P_j(V_i)$ **then**
    return*(NO);*
  if $DEPEND(P_i, P_j(V_i), R_i) = YES$ **then**
    return*(YES);*
if O_j *is replace* **and** *anti-dependence exists* **then**
  if $P_i = P_i(V_j)$ **then**
    return*(NO);*
  if $DEPEND(P_i(V_j), P_j, R_j) = YES$ **then**
    return*(YES);*
return*(NO);*

Procedure: DEPEND(P_i, P_j, R)

/* See [6] for independence test */

if P_i *is independent of* P_j *in R* **then**
  return*(NO);*
return*(YES);*

3.3. Incremental Data Dependence Analysis

The dependence analysis can be incrementally applied to database transactions. The only requirement for incremental dependence analysis is that for a new transaction, the complete (query sequence of the) transaction is used by the dependence analysis to guarantee serializability.

Suppose that the system receives a sequence of transactions $<T_1, T_2, T_3, ..., T_n>$. The dependence analysis is applied to generate dependence relationships among transactions. This analysis can remove all dependence relationships between T_1 and other transactions as soon as T_1 is committed. Suppose that at later time one additional transaction arrives, making the sequence $<T_2, T_3, T_4, ..., T_n, T_{n+1}>$. All existing dependence relationships are still valid; only the dependence relationships between T_{n+1} and other transactions need to be analyzed.

4. Conclusions

We have introduced the data dependence analysis which can be used by the global scheduler for multilevel-secure databases which use the replicated architecture. The primary goal of data dependence analysis is to precisely determine if the portions of relations affected by various operations overlap without the knowledge of actual data or physical layout of the data in each untrusted backend database. If there is no overlap, the query processing will permit the operations to be executed out of sequence or concurrently.

Using dependence analysis permits efficient transaction management while permitting the use of off-the-shelf database systems as the backends in the replicated architecture multilevel database systems. We believe that dependence analysis plays more important role if parallel machines are used as the backend computers. This is the case connecting to host hcig.itd (128.60.2.61), port 530 connection open because the more transactions are submitted simultaneously, the better the chances are to find useful parallelism.

We believe the data dependence analysis may have broad application in non-multilevel database. For example, the data dependence analysis can be applied to batch jobs or database programming because entire contents of a transaction is known beforehand. Also this technique can be used by the transaction preanalysis technique in real-time database [1].

References

[1] Buchmann, A., et el. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. Proceedings of Conference on Data Engineering (1989).

[2] Date, C. J. An introduction to database systems. (Addison Wesley, 1986).

[3] Denning, D. Commutative filters for reducing inference threats in multilevel database systems. Proceedings of the IEEE symposium on Security and Privacy (1985).

[4] Froscher, J. N., and Meadows, C. Achieving a trusted database management systems using parallelism. in Database Security II: Status and Prospects (North-Holland 1989)

[5] Hunt, H., and Rosenkrantz, D. The complexity of testing predicate locks. Proceedings of ACM SIGMOD International Conference on Management of Data (1979)

[6] Kang, M. H. Optimization and parallelization of database queries. Ph.D. Dissertation, Purdue University (1991).

[7] Kang, M. H., Froscher, J. N., and Costich, O. A practical transaction model and untrusted transaction manager for multilevel-secure database systems. IFIP WG 11.3 Sixth Working Conference on Database Security (1992).

[8] Kuck, D. J. The structure of computers and computations, Vol. 1 (Wiley, 1978).

[9] Lunt, T., et el. The seaview security model. IEEE Transaction on Software Engineering, 16, 6 (1990).

[10] Padua, D. A., and Wolfe, M. J. Advanced compiler optimizations for supercomputers. Communications of the ACM, 29, 12 (1986).

[11] Rosenkrantz, D., and Hunt, H. Processing conjunctive predicates and queries. Proceedings of the Conference on Very Large Data Bases (1980).

[12] Wolfe, M., and Banerjee, U. Data dependence and its application to parallel processing. International Journal of Parallel Programming, 16, 2 (1987).