Rules for Transforming Order Dependent Transaction into Order Independent Transaction*

Hamidah Ibrahim
Department of Computer Science, Universiti Putra Malaysia, Malaysia

Abstract: *A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, transactions are required not to violate any database consistency constraints. In most cases, the update operations in a transaction are executed sequentially. The effect of a single operation in a transaction potentially may be changed by another operation in the same transaction. This implies that the sequential execution sometimes does some redundant work. A transaction with a set of update operations is order dependent if and only if the execution of the transaction following the serialibility order as in the transaction produce an output which will be different from the output produced by interchanging the operations in the transaction. Otherwise, the transaction is order independent [8]. In this paper, we present rules that can be applied to generate order independent transaction given an order dependent transaction. An order independent transaction has an important advantage of its update statements being executed in parallel without considering their relative execution orders. With an order independent transaction, we can consider its single updates in an arbitrary order. Furthermore, executing transaction in parallel can reduce the execution time.*

Keywords: *Transaction, parallel processing, transaction decomposition, subtransaction, update operations.*

Received March 2, 2004; accepted June 29, 2004

1. Introduction

Parallel database systems have evolved to cope with the demands forever increasing data storage capacity and data processing performance. Whilst the quantitative requirements of applications are being met by a range of commercial machines and research prototypes, many open issues remain regarding the implementation of efficient mechanisms to help to ensure the quality of data in such systems.

A transaction is a logical unit of work on the database. It may be an entire program, a part of a program or a single command, and it may involve any number of operations on the database. A transaction should always transform the database from one consistent state to another, although we accept that consistency may be violated while the transaction is in progress [3, 4].

[11] has identified three types of fault commonly found in transactions. These faults are:

1. *Inefficient:* Transactions that contain either redundant components which incur unnecessary execution costs, or construct which can be replaced by others which are semantically equivalent but cheaper.
2. *Unsafe:* Transactions do not preserve the consistency of the database.
3. *Unreliable:* Transactions may behave in such a way that their results are either not what the designer has in mind or do not conform to the real world events modeled by the transactions.

Several existing techniques can be applied to overcome the various types of fault as mentioned above. These include transaction optimisation techniques based on high level syntactic or semantic information which can be used to enhance a transaction's efficiency, logic-based techniques for improving integrity checking can be used to reduce both the cost of integrity checking and the complexity of transaction safety verification, techniques for mechanical proof of transaction safety can be used to verify transaction safety automatically, and feedback generated concerning unsafe transactions and changes in the cardinality of the updated relations can be used to help transaction designers amend unsafe transactions and detect unintended results [1, 11].

One particular problem in many advanced applications is the need to support long-lasting transactions. The length of duration of a long-lasting transaction may cause serious performance problems if it is allowed to lock resources until it commits. This may either force other transactions to wait for resources for an unacceptable long time, or it may increase the likelihood of transaction abort. Aborting a long-lasting transaction may have a negative effect on both response time and throughput. If the long transaction has a flat structure, a failure will cause the whole transaction to be undone and possibly reexecuted. This is a very expensive recovery strategy, especially if the failure occurred after executing most

* This work was supported by Ministry of Science, Technology and Innovation (MOSTI) under grant number 04-02-04-0797 EA001.

of the transaction. Decomposing the transaction into a number of subtransactions is one way of dealing with these problems [7].

Although many researchers have investigated the process of decomposing transactions into several subtransactions to increase the performance of the system, but the focus of the research is typically on implementing a decomposition supplied by the database application developer, without really focusing on the decomposition process itself. Examples are [2, 6, 9]. While [8, 10] concentrate on techniques to decompose a transaction into several subtransactions.

[6] has proposed a technique to map an object model to a commercial relational database system using replication and view materialisation and argued that update operations become more complex due to the added redundancy in the mapping of the large classification structures. In order to speed them up, they exploit intra-transaction parallelism by breaking the updates into shorter relational operations. These are executed as ordinary independent parallel transactions on the relational storage server.

[9] has proposed an algorithm which is capable of generating the finest chopping of a set of transactions but his algorithm rely on the following assumptions:

1. A user has access only to user-level tools.
2. A user knows the set of transactions that may run during certain interval.

[2] presents an approach to improve database performance by combining parallelism of multiple independent transactions and parallelism of multiple subtransactions within a transaction without really focusing on the decomposition process.

[10] introduced the notion of semantic histories which do not only list the sequence of steps forming the history, but also convey information regarding the state of the database before and after execution of each step in the history. They have identified several properties which semantic histories must satisfy to show that a particular decomposition correctly models the original collection of transaction. [10] also argued that the interleaving of the steps of a transaction must be constrained so as to avoid inconsistencies and proposed additional preconditions on the auxiliary variables. Although auxiliary variables facilitate analysis, it is expensive to implement them. Also performing additional precondition checks involves extra run time overhead. To avoid implementing auxiliary variables and performing additional precondition checks, they introduce the concept of successors sets, but the successor set descriptions are obtained by examining the preconditions with auxiliary variables.

[8] has proposed a technique for partitioning transaction to reduce the overhead of checking integrity constraints. He has proved that every order dependent transaction can be transformed into equivalent order independent transactions. But in his work he only shows the transformation rules for update operations with the following sequence:

1. Insert followed by delete.
2. Delete followed by insert.
3. Insert followed by change.

Also, his technique is not capable of handling more complex transaction with update operations such as the if construct.

In our research we focus on what constitutes a desirable decomposition and how the developer should obtain such a decomposition. We propose a technique that can be applied to generate subtransactions which will reduce the execution time by exploiting the possibility of executing the transaction in parallel. Our technique differs from the other techniques proposed by other researchers since:

1. The number of subtransactions and the set of update operations derived by our technique are not fix; it depends on several factors such as the number of independent operations, the complexity of independent operations and the location of the relations (for case of distributed database);
2. It does not require additional precondition checks as in [10].
3. Most of the previous works only consider transaction with simple update operations such as [8, 10].
4. Most of the previous works assume that the transaction is efficient without exploring the possibility that an optimized transaction can be obtained by eliminating any redundant or subsumed operation that may occur in the transaction.

In this paper, we focus on deriving efficient transactions, i. e., transactions that are free from containing redundant components which can include unnecessary execution cost. This is achieved by applying a set of rules to a given order dependent transaction. As a result an equivalent order independent transaction is generated. Here, equivalent means that the state produce by executing the initial transaction (order dependent transaction) is the same as executing its order independent transaction. An order independent transaction has an important advantage of its update statements being executed in parallel without considering their relative execution orders as stated in [8].

This paper is organized as follows. In section 2, the basic definitions, notations and examples which are used in the rest of the paper are set out. In section 3, we present the rules that can be applied to transform order dependent transaction into order independent transaction. Conclusions are presented in section 4.

2. Preliminaries

Our approach has been developed in the context of relational databases which can be regarded as consisting of two distinct parts, namely: An intensional part and an extensional part. A database is described by a database schema, D, which consists of a finite set of relation schemas, $<R_1, R_2, \ldots, R_m>$. A relation schema is denoted by R ($A_1, A_2, \ldots, A_n$) where R is the name of the relation (predicate) with n-arity and A_i's are the attributes of R. A database instance is a collection of instances for its relation schemas.

As the real world enterprise changes, the database state, which corresponds to a state of the real world enterprise, must also undergo transitions to reflect those changes. The transition of the database state is carried out by *database transactions*.

A database transaction is one or a sequence of update operations that constitutes some well-defined activity of the enterprise of which the database is model. It is a logical unit of work in the sense that its effect on the database is either committed (i. e., the effects are made permanent) when it is processed successfully in its entirety, or else undone (as if the transaction never executed at all). In our work, only single and conditional operations are considered.

Single operations include insertion, deletion and modification. These operations have the following form:

- ins (R ($c_1, c_2, \ldots, c_n$)): Inserting a tuple into relation R with values $c_1, c_2, \ldots, c_n$.
- del (R (x, …)): Deleting a tuple from relation R with primary key value x.
- del (R (…, <delexp>, …)): Deleting a set of tuples from relation R which satisfy *delexp*.
- mod (R (x, $c_1, \ldots, c_n$): R (x, $c_{n1}, \ldots, c_{nn}$))[1]: Updating a tuple of relation R whose primary key value is x.
- mod (R (…, <modexp>, …): R (…, c_n, c_{n+1}, …)): Updating a set of tuples of relation R which satisfy *modexp*.

where c_i represents any constant, x is the key of relation R, and both *delexp* and *modexp* are constants or simple expressions.

Conditional operation (control structure) has the following format: If C then O1 else O2 where C is a database state referring to relations and O1 and O2 are update operations. The operational interpretation of the above construct is: If C is true then execute O1 else execute O2.

The structure of database transactions adopted by us is composed of two sections, namely: The parameter section and the transaction body as shown below:

```
Transaction Transaction_Name (Parameter)
    Begin
        Transaction Body;
    End
```

Parameter contains parameters used by the operations in a transaction while the transaction body consists of one or more of the update mechanisms as discussed above.

Throughout this paper, the same example *Job Agency* database schema is used, as given in Figure 1. The example is taken from [11].

```
Person (pid, pname, placed);
Company (cid, cname, totsal);
Job (jid, jdescr);
Placement (pid, cid, jid, sal);
Application (pid, jid);
Offering (cid, jid, no_of_places);
```

Figure 1. The job agency schema.

3. Order Dependent and Order Independent Transactions

In most cases, the update operations in a transaction are executed sequentially. The effect of a single operation in a transaction potentially may be changed by another operation in the same transaction. This implies that the sequential execution sometimes does some redundant work [8]. For example the transaction *T1* below is equivalent to *T2* since they produce the same database states. This occurs when there are at least two single updates which conflict with each other. Here two update operations are said to conflict if they operate on the same data item.

```
Transaction T1 (h, c, j, s, n, t1, t2)
    Begin
        ins (Placement (h, c, j, s));
        mod (Company (c, n, t1): Company (c, n, t2));
        del (Placement (h, c, j, s));
    End

Transaction T2 (c, n, t1, t2)
    Begin
        mod (Company (c, n, t1): Company (c, n, t2));
    End
```

As mentioned in section 1, [8] has proposed a technique to decompose a transaction into several subtransactions but his technique is limited due to the reasons as described in section 1. We have improved his technique and this is discussed below.

To exploit parallelism within transaction operations, the operations of the transaction need to be syntactically and semantically analysed to identify the relationship among them. We have identified four types of relationship between operations of a transaction based on the information presented in the operations, i. e., the types of update operations, the relations involved and the values specified in the operations. These relationships are presented below:

[1] Modify operation is considered as a sequence of delete followed by an insert operation as in [5].

A transaction *T* with *n* operations $op_1R1\ (A1)$, $op_2R2\ (A2)$, …, $op_nRn\ (An)$ is said to be *redundant* if there exists at least an operation that occurs more than once in the same transaction. This operation should be eliminated if there are no other operations which change the state of the relation that is involved in the redundant operation.

Definition 1: An operation $op_iRi\ (Ai)$ is said to be *redundant* if there exists at least an operation $op_jRj\ (Aj)$ where $op_i = op_j \in \{ins, del, mod\}$, $Ri = Rj$ and $Ai = Aj$. If $op_i = op_j$, $Ri = Rj$ and $Ai = Aj$, then the transaction *T* contains duplicate operations and therefore redundancy occurs.

Transaction T3 (h, c, j, s)
Begin
del (Placement (h, c, j, s));
mod (Placement (h, c, j, s):
Placement (h, c, j, s + 100));
End

The above transaction *T3* contains redundant operation and since there are no other operations between the redundant operations which change the state of *Placement*, therefore the second operation *mod (Placement (h, c, j, s): Placement (h, c, j, s + 100))* can be removed from the transaction. This is because the modify operation is no longer required as the tuple to be modified does not exist in the relation *Placement*. Table 1 represents some of the redundancy rules that can be applied, namely: Rules 7, 8, 9, 10, 11, and 12.

A transaction *T* with *n* operations $op_1R1\ (A1)$, $op_2R2\ (A2)$, …, $op_nRn\ (An)$ is said to be *subsumed* if there exists at least an operation whose effect is the same as performing another operation in the same transaction. Similar to redundant operation, this operation should be eliminated if there are no other operations which change the state of the relation that is involved in the operation.

Definition 2: An operation $op_iRi\ (Ai)$ is said to be *subsumed* when there exists at least an operation $op_jRj\ (Aj)$ where $op_i = op_j \in \{ins, del, mod\}$, $Ri = Rj$ and $Ai \subset Aj$. If $op_i = op_j$, $Ri = Rj$ and $Ai \subset Aj$, this indicates that performing $op_jRj\ (Aj)$ will also perform $op_iRi\ (Ai)$.

Transaction T4 (j)
Begin
mod (Placement (_, _, _, 1000):
Placement (_, _, _, 2000));
mod (Placement (_, _, j, 1000):
Placement (_, _, _, 2000));
End

The above *mod (Placement (_, _, j, 1000): Placement (_, _, _, 2000))* operation is subsumed by *mod (Placement (_, _, _, 1000): Placement (_, _, _, 2000))* since performing *mod (Placement (_, _, _, 1000): Placement (_, _, _, 2000))* will also modify the tuple <_, _, j, 1000>. Therefore *mod (Placement (_, _, j, 1000): Placement (_, _, _, 2000))* should be removed from the transaction.

Given a transaction *T* with update operations $op_1R1\ (A1)$, $op_2R2\ (A2)$, …, $op_nRn\ (An)$, *T* is *order dependent* if and only if the execution of the transaction following the serialibility order as in the transaction produce an output which will be different than the output produced by interchanging the operations in the transaction. A transaction *T* is *order dependent* if and only if *T* contains at least two conflicting update operations. Otherwise *T* is *order independent*.

Definition 3: An operation $op_iRi\ (Ai)$ is said to be *dependent* on operation $op_jRj\ (Aj)$ if and only if $op_i \neq op_j$, $Ri = Rj$ and satisfy the conditions in Table 1.

Definition 4: An operation $op_iRi\ (Ai)$ is said to be *independent* if and only if for all operations in transaction *T*, $op_jRj\ (Aj)$ where $j = 1, \ldots, n$ and $j \neq i$,

1. $op_i \neq op_j$ and $Ri \neq Rj$ or
2. $op_i = op_j$ and $Ri \neq Rj$ or
3. $op_i = op_j$, $Ri = Rj$ and $Ai \neq Aj$

As dependent operations occur only when the relations in both operations are the same therefore 1 and 2 above are proved. Also, dependent operations require that both type of operations are different, therefore 3 is also proved.

Transaction T5 (h, c, j, s)
Begin
ins (Placement (h, c, j, s));
del (Placement (h, c, j, s));
End

Transaction T6 (hiree, h, c, j, s)
Begin
ins (Placement (h, c, j, s));
del (Application (hiree, _));
End

Transaction *T5* is order dependent while *T6* is order independent. An order independent transaction has an important advantage of its update statements being executed in parallel without considering their relative execution orders. With an order independent transaction we can consider its single updates in an arbitrary order. As proved in [8], every order dependent transaction can be transformed into equivalent order independent transaction. But this is not true as discussed at the end of this section.

In the following, we present the rules to convert dependent operations (conflicting updates) into equivalent independent operations (non-conflicting updates). Here, we use the symbol +R to indicate the new state of the database after inserting a tuple (set of tuples) into relation R and the symbol –R to indicate the new state of the database after deleting a tuple (set of tuples) from relation R. Therefore, +R (c_1, c_2, …, c_n)

(-R (c_1, c_2,…, c_n), respectively) means a new database state after inserting (deleting, respectively) the tuple <c_1, c_2, …, c_n> into (from) relation R. R (…, c_i,…) (R (…, c_{in}, …), respectively) is a set of tuples satisfying a condition(s) over domain c_i (domain c_{in}, respectively) and <c_1, c_2,…, c_n> ⊂ R (…, c_i, …) (<c_{n1}, …, c_{nn}> ⊂ R (…, c_{in}, …), respectively). With this, the following assumptions are true:

- *Assumption 1*: +R (c_1, c_2, …, c_n) +R (c_1, c_2, …, c_n) is not possible since once a tuple <c_1, c_2, …, c_n> has been inserted into R, the same tuple <c_1, c_2, …, c_n> cannot be inserted into R at a later time as relational model does not allow duplicate copies.
- *Assumption 2*: -R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n) ≡ -R(c_1, c_2, …, c_n).
- *Assumption 3*: -R (c_1, c_2,…, c_n) [-R (c_1, c_2, …, c_n) +R (…)] ≡ -R (c_1, c_2, …, c_n). [-R (c_1, c_2, …, c_n) +R (…)] refers to a modify operation. This is true since when the tuple <c_1, c_2, …, c_n> is deleted, the modify operation is no longer necessary since the tuple to be modified does not exist anymore.
- *Assumption 4*: +R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n) ≡ 0 which means that the database is in its initial state, i. e., no changes has occur. This is true since inserting a tuple <c_1, c_2, …, c_n> and later on deleting the same tuple <c_1, c_2, …, c_n> from the same relation R, will bring back the new state of the database to its initial state.

Rule 1: dependent operations:
ins (R (c_1, c_2, …, c_n)); del (R (c_1, c_2, …, c_n));
equivalent independent operation: nothing
Proof: +R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n) ≡ 0

Rule 2: dependent operations:
ins (R (c_1, c_2, …, c_n)); del (R (…, c_i, …));
equivalent independent operation:
del (R (X ≠ c_1, …, c_i, …))
Proof: +R (c_1, c_2, …, c_n) -R (…, c_i, …)
= +R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n)
-R (X ≠ c_1, …, c_i, …)
= -R (X ≠ c_1, …, c_i, …)

Rule 3: dependent operations:
del (R (c_1, c_2, …, c_n)); ins (R (c_1, c_2, …, c_n));
equivalent independent operation: nothing
Proof: -R (c_1, c_2, …, c_n) +R (c_1, c_2, …, c_n) ≡ 0

Rule 4: dependent operations:
del (R (…, c_i, …)); ins (R (c_1, c_2, …, c_n));
equivalent independent operation:
del (R (X ≠ c_1, …, c_i, …))
Proof: -R (…, c_i, …) +R (c_1, c_2, …, c_n)
= -R (c_1, c_2, …, c_n) -R (X ≠ c_1, …, c_i, …) +R (c_1, c_2, …, c_n)
= -R (X ≠ c_1, …, c_i, …)

Rule 5: dependent operations:
ins (R (c_1, c_2, …, c_n));
mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));
equivalent independent operation:
ins (R (c_{n1},…, c_{nn}))
Proof: +R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n) +R (c_{n1}, …, c_{nn})
= +R (c_{n1}, …, c_{nn})

Rule 6: dependent operations:
mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));
ins (R (c_1, c_2, …,c_n));
equivalent independent operation:
ins (R (c_{n1}, …, c_{nn}))
Proof: -R (c_1, c_2, …, c_n) +R (c_{n1}, …, c_{nn}) +R (c_1, c_2, …, c_n)
= +R (c_{n1}, …, c_{nn})

Rule 7: dependent operations:
del (R (c_1, c_2, …, c_n)); mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));
equivalent independent operation:
del (R (c_1, c_2, …, c_n))
Proof: -R (c_1, c_2,…, c_n) [-R (c_1, c_2,…, c_n) +R (c_{n1},…, c_{nn})]
= -R (c_1, c_2, …, c_n) (refer to Assumption 3)

Rule 8: dependent operations:
del (R (…, c_i, …)); mod (R(…, c_i, …): R (…, c_{in}, …));
equivalent independent operation:
del (R (…, c_i, …))
Proof: -R (…, c_i,…) [-R (…, c_i, …) +R (…, c_{in}, …)]
= -R (…, c_i, …) (refer to Assumption 3)

Rule 9: dependent operations:
mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));
del (R (c_1, c_2,…, c_n));
equivalent independent operation:
mod (R (c_1, c_2, …, c_n): R (c_{n1},…, c_{nn}))
Proof: -R (c_1, c_2, …, c_n) +R (c_{n1}, …, c_{nn}) -R (c_1, c_2, …, c_n)
= -R (c_1, c_2, …, c_n) +R (c_{n1}, …, c_{nn})

Rule 10: dependent operations:
mod (R (…, c_i, …): R (…, c_{in}, …));
del (R (…, c_i, …));
equivalent independent operation:
mod (R (…, c_i, …): R (…, c_{in}, …))
Proof: -R(…, c_i, …) +R (…, c_{in}, …) -R (…, c_i, …)
= -R (…, c_i, …) +R (…, c_{in}, …)

Rule 11: dependent operations:
ins (R (c_1, c_2, …, c_n));
mod (R (c_{n1}, …, c_{nn}) :R (c_1, c_2, …, c_n));
equivalent independent operation:
not possible
Proof: +R (c_1, c_2, …, c_n) -R (c_{n1}, …, c_{nn}) +R (c_1, c_2, …, c_n)
= +R (c_1, c_2, …, c_n) +R (c_1, c_2,…, c_n) -R (c_{n1}, …, c_{nn})
= not possible (refer to Assumption 1)

Rule 12: dependent operations:
mod (R (c_{n1}, …, c_{nn}): R(c_1, c_2, …, c_n));
ins (R (c_1, c_2, …, c_n));
equivalent independent operation:
not possible
Proof: -R (c_{n1}, …, c_{nn}) +R (c_1, c_2, …, c_n) +R (c_1, c_2,…, c_n)
= not possible (refer to Assumption 1)

Rule 13: dependent operations:
del (R (c_1, c_2, …, c_n));
mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2, …, c_n));
equivalent independent operation:
del (R (c_{n1}, …, c_{nn}))
Proof: -R (c_1, c_2, …, c_n) -R (c_{n1}, …, c_{nn}) +R (c_1, c_2, …, c_n)
= -R (c_1, c_2,…, c_n) +R (c_1, c_2, …, c_n) -R (c_{n1}, …, c_{nn}) = -R (c_{n1}, …, c_{nn})

Rule 14: dependent operations:
mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2, …, c_n));
del (R (c_1, c_2, …, c_n));
equivalent independent operation:
del (R (c_{n1}, …, c_{nn}))
Proof: -R (c_{n1}, …, c_{nn}) +R (c_1, c_2,…, c_n) -R (c_1, c_2, …, c_n)
= -R (c_{n1}, …, c_{nn})

Rule 15: dependent operations:
mod (R (…, c_{in}, …): R (…, c_i, …));
del (R (…, c_i, …));
equivalent independent operation:
del (R (…, c_{in}, …)); del (R (…, c_i, …))
Proof: [-R (…, c_{in}, …) +R (…, c_i, …)] -R (…, c_i, …)
= -R (…, c_{in}, …) -R(…, c_i, …)

Table 1 summarizes the rules that we have presented above. In the following we will show through examples how the rules that we have presented can be applied to generate order independent transaction given an order dependent transaction.

```
Transaction T7 (p, n)
    Begin
      ins (Person (p, n, false));
      mod (Person (p, n, false): Person (p, n, true));
    End
```

Applying rule 5 will generate the following order independent transaction, *T7'*.

```
Transaction T7' (p, n)
    Begin
      ins (Person (p, n, true));
    End

Transaction T8 (t)
    Begin
      mod (Company (_, _, t < 0): Company (_, _, t
        = 0));
      del (Company (_, _, t < 0));
    End
```

Applying rule 10 will generate the following order independent transaction, *T8'*.

```
Transaction T8'(t)
    Begin
      mod (Company (_, _, t < 0): Company(_, _, t
        = 0));
    End
```

The above *del (Company (_, _, t < 0))* operation is removed from transaction *T8* since there is no tuple in the relation *Company* that will satisfy the condition $t < 0$ as these tuples have been modified by the operation *mod (Company (_, _, t < 0): Company (_, _, t = 0))*.

```
Transaction T9 (t)
    Begin
      mod (Company (_, _, t < 0): Company (_, _, t
      = 0));
      del (Company(_, _, t = 0));
    End
```

Applying rule 15 will generate the following order independent transaction, *T9'*.

```
Transaction T9'(t)
    Begin
      del (Company (_, _, t < 0));
      del (Company (_, _, t = 0));
    End

Transaction Company_Status (c, n, totsal)

    Begin
      If Company (c, n, totsal < 0) then
        del (Company (c, _, _));

      If Placement (_, c, _, _) and not Company (c,
        _, _)
        then ins (Company (c, _, _));
    End
```

Here, a truth table as shown in Table 2 is derived based on the truth values of the conditions specified in the if construct. For each possibility, an equivalent independent operation is generated.

Table 1 presents the conflicting updates (dependent operations) in which equivalent independent operations can be derived, it is equivalent to not performing at all the conflicting updates (stated by nothing) or it is not possible to perform the updates as this will violate the assumption(s) given above. These rules are based on the term conflicting updates which means that two update operations operate on the same data item. Other sequences of update operations which are syntactically correct but are not included in the table since:

1. Semantically they do not make sense.
2. No single equivalent independent operation can be derived as shown by rule 16 and example T10.

3. No equivalent independent operation can be derived as shown by rule 17 and example T11.

Table 1. Converting conflicting updates to non-conflicting updates.

Rule	Conflicting Updates	Equivalent Non-Conflicting Updates
1	ins (R (c_1, c_2, …, c_n)); del (R (c_1, c_2, …, c_n));*	nothing
2	ins (R (c_1, c_2, …, c_n)); del (R (…, c_i, …));	del (R (x, …, c_i, …)) where x ≠ c_1
3	del (R (c_1, c_2, …, c_n));* ins (R (c_1, c_2, …, c_n));	nothing
4	del (R (…, c_i, …)); ins (R (c_1, c_2, …, c_n));	del (R (x, …, c_i, …)) where x ≠ c_1
5	ins (R (c_1, c_2, …, c_n)); mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));*	ins (R (c_{n1}, …, c_{nn}))
6	mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));* ins (R (c_1, c_2, …, c_n));	ins (R (c_{n1}, …, c_{nn}))
7	del (R (c_1, c_2, …, c_n));* mod (R (c_1, c_2,…, c_n): R (c_{n1}, …, c_{nn}));*	del (R (c_1, c_2, …, c_n))
8	del (R (…,c_i,…)); mod (R (…, c_i, …): R (…, c_{in}, …));	del (R (…, c_i, …))
9	mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}));* del (R (c_1, c_2, …, c_n));	mod (R (c_1, c_2, …, c_n): R (c_{n1}, …, c_{nn}))
10	mod (R (…, c_i, …): R (…, c_{in}, …)); del (R (…, c_i, …));	mod (R (…, c_i, …): R (…, c_{in}, …))
11	ins (R (c_1, c_2, …, c_n)); mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2, …, c_n));*	not possible
12	mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2,…, c_n));* ins (R (c_1, c_2, …, c_n));	not possible
13	del (R (c_1, c_2, …, c_n));* mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2, …, c_n));*	del (R (c_{n1}, …, c_{nn}))
14	mod (R (c_{n1}, …, c_{nn}): R (c_1, c_2, …, c_n));* del (R (c_1, c_2, …, c_n));*	del (R (c_{n1}, …, c_{nn}))
15	mod (R (…, c_{in}, …): R (…, c_i, …)); del (R (…, c_i, …));	del (R (…, c_{in}, …)) del (R (…, c_i, …))

Table 2. Truth table.

Condition 1: Company (c, n, totsal < 0)	Condition 2: Placement (_, c, _, _) and not Company (c, _, _)	Operations	Rule Applied: Independent Operations
True	True	del (Company (c, _ , _)) ins (Company (c, _, _))	Rule 3: nothing
True	False	del (Company (c, _, _))	del (Company (c, _, _))
False	True	ins (Company (c, _, _))	Ins (Company (c, _, _))
False	False	nothing	nothing

Note that if condition 1 is true then definitely condition 2 is false. Identifying contradiction between conditions in the if constructs is not the focus of this paper.

* Equivalent non-conflicting updates will be derived if the operations specify only the value of the primary key. ci ∈ {c2, …, cn}, cin ∈ {cn2, …, cnn}, c1 and cn1 are the primary key values.

Rule 16: dependent operations:
ins (R (c_1, c_2, …, c_n));
mod (R (…, c_i, …): R (…, c_{in}, …));
equivalent independent operation:
ins (R (c_1, …, c_{in},…));
mod (R (…, c_i, …): R (…, c_{in}, …));

Proof: +R (c_1, c_2, …, c_n) -R (…, c_i, …) +R (…, c_{in}, …)
= +R (c_1, c_2, …, c_n) -R (c_1, c_2, …, c_n)
-R (X ≠ c_1, c_2, …, c_n) +R (c_1, …, c_{in}, …)
+R (X ≠ c_1, …, c_{in},…)
= -R (X ≠ c_1, c_2, …, c_n) +R (c_1, …, c_{in}, …)
+R (X ≠ c_1, …, c_{in}, …)

Consider the following example,

Transaction T10 (p, c, j)
Begin
ins (Placement (p, c, j, 1000));
mod (Placement (_, _, _,1000): Placement (_, _, _, 2000));
End

The above transaction *T10* is equivalent to the following transaction *T10'* which consists of independent operations.

Transaction T10'(p, c, j)
Begin
ins (Placement (p, c, j, 2000));
mod (Placement (_, _, _, 1000): Placement (_, _, _, 2000));
End

Rule 17: dependent operations:
del (R (…, c_i, …));
mod (R (…, c_{in}, …): R (…, c_i, …));

Consider the following example,

Transaction T11 (t)
Begin
del (Company (_, _, t < 0));
mod (Company(_, _, t = 0): Company(_, _, t < 0));
End

No equivalent order independent transaction can be derived for the above transaction T11.

4. Conclusion

Designing efficient, safe and reliable transactions is a difficult task. This paper presents rules that can be applied to transform a given order dependent transaction into order independent transaction. The rules can improve the transaction by indirectly detecting redundant and subsumed operations which are then removed from the transaction. Since

independent operations in a transaction can be executed in arbitrary order, this implies that the transaction's update statements can be executed in parallel without considering their relative execution orders. This can reduce the execution time.

References

[1] Chakravarthy U. S., Grant J., and Minker J., "Logic-Based Approach to Semantic Query Optimization," *ACM TODS*, vol. 15, no. 2, pp. 162-207, 1990.

[2] Christof H. and Gerhard W., "Inter- and Intra-Transaction Parallelism in Database Systems," *in Proceedings of the 14th Speedup Workshop on Parallel and Vector Computing*, Zurich, Switzerland, 1993.

[3] Connolly T. M. and Begg C. E., *Database Systems: A Practical Approach to Design, Implementation and Management*, Addison-Wesley, 2002.

[4] Ibrahim, H., "Extending Transactions with Integrity Rules for Maintaining Database Integrity," *in Proceedings of the International Conference on Information and Knowledge Engineering (IKE'02)*, in Arabnia H. R., Mun Y., and Prasad B. (Eds), Computer Science Research, Education and Application Technical (CSREA) Press, Las Vegas, USA, pp. 341-347, 2002.

[5] McCaroll N. F., "Semantic Integrity Enforcement in Parallel Database Machines," *PhD Thesis*, University of Sheffield, Sheffield, UK, 1995

[6] Michael R., Moira C. N., and Hans-Jorg S., "Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System," *in Proceedings of the 22nd Very Large Databases (VLDB) Conference*, Bombay, India, pp. 1-12, 1996.

[7] ODS Group, "*A Reader in Transaction Processing*," http://www.cs.uit.no/forskning/ODS/ODSProjects/adtrans/ReaderTrans.html.

[8] Sang H. L., Lawrence J. H., Myoung H. K., and Yoon-Joon L., "Enforcement of Integrity Constraints against Transactions with Transition Axioms," *in Proceedings of the 16th. Annual International Computer Software and Applications*, pp. 162-167, 1992.

[9] Shasha D., Llirbat F., Simon E., and Valduriez P., "Transaction Chopping: Algorithms and Performances Studies," *Journal of ACM Transaction Database Systems*, vol. 20, no. 3, pp. 325-363, 1995.

[10] Sushil J., Indrakshi R., and Paul A., "Implementing Semantic-Based Decomposition of Transactions," *in Proceedings of CAiSE'1997*, pp. 75-88, 1997.

[11] Wang X. Y., "The Development of a Knowledge-Based Transaction Design Assistant," *PhD Thesis*, University of Wales College of Cardiff, Cardiff, UK, 1992.

Hamidah Ibrahim is currently an associate professor at the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. She obtained her PhD in computer science from the University of Wales Cardiff, UK in 1998. Her current research interests include distributed databases, transaction processing, and knowledge-based systems.