Rules and Strategies for Generating Efficient Independent Subtransactions

Hamidah Ibrahim

Department of Computer Science, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia

Summary
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. It is the transaction designer's responsibility to define properly the various transactions so that it preserves the consistency of the database. In the literature, three types of fault have been identified in transactions, namely: inefficient, unsafe and unreliable. In this paper, we present the strategies that can be applied to generate subtransactions to exploit parallelism. In our work, we have identified five types of relationship which can occur in a transaction. They are: redundancy, subsumption, dependent, partly dependent and independent. By analysing these relationships, the transaction can be improved and inefficient transactions can be avoided. Furthermore, generating subtransactions and executing them in parallel can reduce the execution time.
Key words:
Transaction, Subtransactions, Parallel Processing

1. Introduction

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 [2, 4]. To satisfy this goal, a transaction should have the four (ACID) properties, namely: atomicity, consistency, isolation, and durability.

[10] has identified three types of fault commonly found in transactions. These faults are (i) 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, (ii) unsafe – transactions do not preserve the consistency of the database, and (iii) unreliable – transactions may behave in such a way that their results either are not what the designer have in mind or do not conform to the real world events modeled by the transaction.

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 of the transaction. Decomposing the transaction into a number of subtransactions is one way of dealing with these problems [6].

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 [1, 5] and [8]. While [7] and [9] concentrate on techniques to decompose a transaction into several subtransactions.

[5] 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.

[8] 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: (i) a user has access only to user-level tools and (ii) a user knows the set of transactions that may run during certain interval.

[1] 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.

[9] introduced the notion of semantic histories which 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. [9] 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.

[7] 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 sequences (i) insert followed by delete (ii) delete followed by insert and (iii) insert followed by change. Also, his technique is not capable of handling 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 (i) the number of subtransactions and the set of update operations derived by our technique are not fix; it depends on several factors as highlighted in Section 4; (ii) it does not require additional precondition checks as in [9]; (iii) most of the previous works only consider transaction with simple update operations such as [7] and [9]; and (iv) 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. Therefore, we focus on deriving efficient transactions, i.e. transactions that are free from containing redundant and subsumed components that can incur unnecessary execution cost. This is achieved by applying a set of rules to a given transaction which in most cases is an order dependent or partly order dependent transaction. We have also enhanced the work by [7] by introducing complete rules for mapping dependent or partly dependent transaction into transaction where its single updates can be executed in arbitrary order. As a result an equivalent order independent transaction is generated. Here, equivalent means that the state produce by executing the initial transaction (order dependent or partly 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 [7].

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 eliminate redundant and subsumed constructs as well as to transform order dependent or partly order dependent transaction into order independent transaction. Also, the steps to transform a given transaction into an optimized independent transaction are presented. Section 4 presents several strategies that can be adopted to generate subtransactions to be executed in parallel. In Section 5, we analyze the generated subtransactions and we compare them to their respective initial transactions. Conclusions are presented in the final Section, 6.

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.

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($R(x,c_1,\ldots,c_n)$:$R(x,c_{n1},\ldots,c_{nn})$) – updating a tuple of relation *R* whose primary key value is x, and

- 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 executes O1 else executes 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 is used, as given in Figure 1. The example is taken from [10].

```
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)

Transaction  Hire(hiree,comp,jb,sal)
Begin
  modify p in person where p.pid=hiree by [placed=true];
  if all o in offering where o.cid=comp and
o.jid=jb:o.no_of_places=1 then delete o
    else modify o by [no_of_places=no_of_places-1];
  insert [hiree,comp,jb,sal] into placement;
  delete p from application where p.pid=hiree;
  modify c in company where c.cid=comp by [totsal=totsal+sal];
End
```

Fig. 1 The *Job Agency* schema and the *Hire* transaction.

The transaction given in Figure 1 can be rewritten as follows:

```
1: Transaction Hire(hiree,comp,jb,sal)
2: Begin
3:  mod(Person(hiree,_,false):Person(hiree,_,true));
4:  if Offering(comp,jb,1) then del(Offering(comp,jb,1))
      else mod(Offering(comp,jb,no_of_places):
      Offering(comp,jb,no_of_places–1));
5:  ins(Placement(hiree,comp,jb,sal));
6:  del(Application(hiree,_));
7:  mod(Company(comp,_,totsal):
      Company(comp,_,totsal+sal));
8: End
Note: The symbol '_' indicates that the value of the
column is not necessary.
```

Fig. 2 The *Hire* transaction using our constructs.

3. Deriving Independent Transaction

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 [7]. 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 [7]. Therefore it is important to syntactically and semantically analysed the given transaction to identify the relationship among the operations in the transaction before it is being partitioned into subtransactions. This section focuses on the steps of generating an optimized independent transaction. The independent transaction will be the input to the next phase (presented in Section 4) to generate subtransactions so that parallelism can be exploited.

```
Transaction T1(h,c,j,s,n,t1,t2)
Begin
  ins(Placement(h,c,j,s));
  mod(Company(c,n,t1):(c,n,t2));
  del(Placement(h,c,j,s));
End
Transaction T2(c,n,t1,t2)
Begin
  mod(Company(c,n,t1):(c,n,t2));
End
```

We have identified five 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:

A transaction *T* with *n* operations *$op_1R1(A1),op_2R2(A2),\ldots,op_nRn(An)$* is said to be *redundant*, T_R, 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.

Table 1 represents the redundancy rules that can be applied to derive transaction with redundancy being eliminated.

```
Transaction T3(h,c,j,s)
Begin
  del(Placement(h,c,j,s));
  mod(Placement(h,c,j,s):
    (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):(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*. Here, redundancy rule 5 of Table 1 is applied. Consider the following example,

```
  Transaction T4(t)
  Begin
    mod(Company(_,_,t<0):
      (_,_,t=0));
    del(Company(_,_,t<0));
  End
```

Applying redundancy rule 16 of Table 1 will generate the following order independent transaction, *T4'*.

```
  Transaction T4'(t)
  Begin
    mod(Company(_,_,t<0):
      (_,_,t=0));
  End
```

The above *del(Company(_,_,t<0))* operation is removed from transaction *T4* 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):(_,_,t=0))*.

Table 1: Rules for eliminating redundant updates

Rule	Redundant updates	Equivalent non-redundant updates
1.	del(R(T));mod(R(T):(S));	del(R(T))
2.	del(R(t_1,…));mod(R(T):(S));	del(R(t_1,…))
3.	del(R(T));mod(R(t_1,…):(S));	del(R(T)
4.	del(R(t_1,…)); mod(R(t_1,…):(S));	del(R(t_1,…))
5.	del(R(T));mod(R(T):(t_1,S));	del(R(T))
6.	del(R(t_1,…)); mod(R(T):(t_1,S));	del(R(t_1,…))
7.	del(R(T)); mod(R(t_1,…):(t_1,S));	del(R(T))
8.	del(R(t_1,…)); mod(R(t_1,…):(t_1,S));	del(R(t_1,…))
9.	del(R(…,t_i,…)); mod(R(…,t_i,…):(…,s_j,…));	del(R(…,t_i,…))
10.	mod(R(T):(S)); del(R(T));	mod(R(T):(S))
11.	mod(R(t_1,…):(S));del(R(T));	mod(R(t_1,…):(S))
12.	mod(R(t_1,…):(S)); del(R(t_1,…));	mod(R(t_1,…):(S))
13.	mod(R(T):(S));del(R(t_1,…));	mod(R(T):(S))
14.	mod(R(T):(t_1,S));del(R(T));	mod(R(T):(t_1,S))
15.	mod(R(t_1,…):(t_1,S)); del(R(T));	mod(R(t_1,…): (t_1,S)
16.	mod(R(…,t_i,…):(…,s_j,…)); del(R(…,t_i,…));	mod(R(…,t_i,…): (…,s_j,…));

Note for Table 1, 2, 3 and 4: T (S, respectively) is a tuple in the form of <t_1,t_2,…,t_n> (<s_1,s_2,…,s_m>, respectively); S and T are different data items; R(t_1,S) is a tuple in the form of <t_1,s_2,…,s_m>; t_i,… (s_j, respectively) ≡ where condition t_i (s_j, respectively) is true; $t_1 \in$ T and t_1 is the primary key value.

A transaction *T* with *n* operations *$op_1R1(A1)$,$op_2R2(A2)$,…,$op_nRn(An)$* is said to be *subsumed*, *T_S*, 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 T5(j)
  Begin
    mod(Placement(_,_,_,1000):
      (_,_,_,2000));
    mod(Placement(_,_,j,1000):
      (_,_,_,2000));
  End
```

The above *mod(Placement(_,_,j,1000):(_,_,_,2000))* operation is subsumed by *mod(Placement(_,_,_,1000):(_,_,_,2000))* since performing *mod(Placement(_,_,_,1000):(_,_,_,2000))* will also modify the tuple *<_,_,j,1000>* of *Placement*. Therefore *mod(Placement(_,_,j,1000):(_,_,_,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*, *T_D*, 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. A transaction *T* is *partly order dependent*, *T_{PD}*, if and only if *T* contains at least two partly conflicting updates operations. Two update operations are said to partly conflict if one of the update operation is operating on a data item which is part of a set of data items operate by the other update operation. Otherwise *T* is *order independent*, *T_I*.

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*, *Ai = Aj* and satisfy the conditions in Table 2.

Definition 4: An operation *$op_iRi(Ai)$* is said to be *partly dependent* on operation *$op_jRj(Aj)$* if and only if *$op_i \neq op_j$*, *Ri = Rj*, *$Ai \subset Aj$* and satisfy the conditions in Table 3.

Definition 5: An operation *$op_iRi(Ai)$* is said to be *independent* if and only if for all operations in transaction *T*, *$op_jRj(Aj)$* where $j = 1,…,n$ and $j \neq i$, (i) *$op_i \neq op_j$* and *$Ri \neq Rj$*

or (ii) $op_i = op_j$ and $Ri \neq Rj$ or (iii) $op_i = op_j$, $Ri = Rj$ and $Ai \neq Aj$.

As dependent/partly dependent operations occur only when the relations in both operations are the same therefore (i) and (ii) above are proved. Also, dependent/partly dependent operations require that both type of operations are different, therefore (iii) is also proved.

```
Transaction **T6**(h,c,j,s)
Begin
  ins(Placement(h,c,j,s));
  del(Placement(h,c,j,s));
End
Transaction **T7**(hiree,h,c,j,s)
Begin
  ins(Placement(h,c,j,s));
  del(Application(hiree,_));
End
```

Transaction *T6* is order dependent while *T7* 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 [7], every order dependent transaction can be transformed into equivalent order independent transaction. But this is not true as discuss at the end of this section.

Tables 2 and 3 present the rules to convert dependent and partly dependent operations (conflicting and partly conflicting updates) into equivalent independent operations (non-conflicting updates). 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 or partly order dependent transaction.

```
Transaction **T8**(p,n)
Begin
  ins(Person(p,n,false));
  mod(Person(p,n,false):
     (p,n,true));
End
```

Applying rule 6 of Table 2 will generate the following order independent transaction, *T8'*.

```
Transaction **T8'**(p,n)
Begin
  ins(Person(p,n,true));
End
Transaction **T9**(t)
Begin
   mod(Company(_,_,t<0):
      (_,_,t=0));
   del(Company(_,_,t=0));
End
```

Applying rule 5 of Table 3 will generate the following order independent transaction, *T9'*.

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

Table 2: Rules for eliminating dependent updates

Rule	Dependent updates	Equivalent independent updates
1.	ins(R(T));del(R(T));	nothing
2.	ins(R(T));del(R(t_1,…));	nothing
3.	del(R(T));ins(R(T));	nothing
4.	ins(R(T));mod(R(T):(S));	ins(R(S))
5.	ins(R(T));mod(R(t_1,…):(S));	ins(R(S))
6.	ins(R(T));mod(R(T):(t_1,S));	ins(R(t_1,S))
7.	ins(R(T)); mod(R(t_1,…):(t_1,S));	ins(R(t_1,S))
8.	mod(R(T):(S));ins(R(T));	ins(R(S))
9.	mod(R(t_1,…):(t_1,S)); del(R(t_1,…));	del(R(t_1,…));
10.	mod(R(T):(t_1,S)); del(R(t_1,…));	del(R(t_1,…));
11.	del(R(T));mod(R(S):(T));	del(R(S))
12.	del(R(T)); mod(R(…,s_j,…):(T));	del(R(…,s_j,…))
13.	mod(R(S):(T));del(R(T));	del(R(S))
14.	mod(R(S):(T));del(R(t_1,…));	del(R(S))
15.	mod(R(…,s_j,…):(T)); del(R(t_1,…));	del(R(…,s_j,…))
16.	mod(R(…,s_j,…):(T)); del(R(T));	del(R(…,s_j,…))

Table 3: Rules for eliminating partly dependent updates

Rule	Partly dependent updates	Equivalent independent updates
1.	ins(R(T)); del(R(…,t_i,…));	del(R(…,t_i,…)
2.	ins(R(T)); mod(R(…,t_i,…):(t_1,S));	mod(R(…,t_i,…): (t_1,S));
3.	mod(R(S):(T)); del(R(…,t_i,…));	del(R(S)) del(R(…,t_i,…))
4.	mod(R(…,s_j,…):(T)); del(R(…,t_i,…));	del(R(…,s_j,…)) del(R(…,t_i,…))
5.	mod(R(…,s_j,…): (…,t_i,…)); del(R(…,t_i,…));	del(R(…,s_j,…)) del(R(…,t_i,…))

```
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 4 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 4: Company_Status transaction rules

Condition 1	Condition 2	Operations	Rule applied: independent operations
True	True*	del(Company(c,_,_)) ins(Company(c,_,_))	Rule 3 of Table 2: nothing
True	False	del(Company(c,_,_))	no changes
False	True	ins(Company(c,_,_))	no changes
False	False	nothing	nothing

* Note that if Condition 1 (Company(c,n,totsal<0)) is true then definitely Condition 2 (Placement(_,c,_,_) and not Company(c,_,_)) is false. Identifying contradiction between conditions in the if constructs is not the focus of this paper.

So far we have shown that given conflicting and partly conflicting updates (dependent and partly dependent operations) (i) equivalent independent operations can be derived; (ii) it is equivalent to not performing at all the conflicting updates (stated by nothing); or (iii) it is not possible to perform the updates (as shown by Table 5). These rules are based on the term conflicting updates which means that two update operations operate on the same data item or based on the term partly conflicting updates which means that one of the update operation is operating on a data item which is part of a set of data items operate by the other update operation. Other sequences of update operations which are syntactically correct but are not included in the tables since (i) semantically they do not make sense; (ii) no single equivalent independent operation can be derived as shown by transaction T10; and (iii) no equivalent independent operation can be derived as shown by transaction T11.

Table 5: Rules for eliminating dependent updates

Dependent updates	Equivalent independent updates
mod(R(T):(t_1,S));ins(R(T));	nothing

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

Consider the following example,

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

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

Having discussed the above definitions, the steps to derive an independent transaction, T_I are as follows:

```
INPUT:  transaction $T$
OUTPUT:  $T_I$
BEGIN
    Apply Definition 1 to $T$,
      if $T$ is $T_R$ then $T' = T_R$ – redundant operations
      else $T' = T$.
    Apply Definition 2 to $T'$,
      if $T'$ is $T_s$ then $T' = T_s$ – subsumed operations.
    Apply Definition 3 to $T'$,
      if $T'$ is $T_D$ then
      $T'$ = apply the dependent rules to $T_D$.
    Apply Definition 4 to $T'$,
      if $T'$ is $T_{PD}$ then
      $T'$ = apply the partly dependent rules to $T_{PD}$.
    $T_I = T'$
END
```

4. Strategies for Deriving Subtransactions

Once the relationships among the operations have been detected and the transaction has been optimised (eliminate redundancy, subsumption and convert dependent or partly dependent operations to independent operations), the next step is to group the operations (independent) in the transaction into several groups or subtransactions. Here, several strategies can be applied so that each processor will be given the same number/complexity of operations. We assume that each processor has the same capability (speed).

i) Number of independent operations – in general if there are *m* processors and *n* number of independent operations, then each processor will be given approximately *n*/*m* operations if $n \geq m$ and 1 operation if $n < m$ (not all processors will involve in processing the transaction). Then assigning the update operations to the processors will be based on: any *n*/*m* operations to the first processor, then any *n*/*m* operations from the balance of operations to the second processor and so on. The number of ways each processor is given the set of update operations is given by the following equation:

$$\prod_{i=0}^{n-r/r}\left[\frac{n-ir!}{r!(n-(i+1)r)!}\times\frac{1}{p-i}\right] \qquad (1)$$

where *n* is the number of update operations, *p* is the number of available processors and *r* = *n*/*p*. Equation (1) is for case where *r* is an integer value.

Example: If a transaction, T, consists of 4 update operations denoted as 1, 2, 3, 4 and there are 2 processors then there are 3 possible number of ways each processor is given the set of update operations. These alternatives are shown below:

Alternative A: {1, 2}, {3, 4}
Alternative B: {1, 3}, {2, 4}
Alternative C: {1, 4}, {2, 3}

The following table shows some of the possible number of alternatives, *a*, that can be generated given the various numbers of update operations, *n*, and processors, *p*.

n	4	6	6	8	8	9	10	10
p	2	2	3	2	4	3	2	5
a	3	10	15	35	105	280	126	945

This strategy is easy to implement and although each processor is given approximately the same number of update operations but not necessarily that each processor will execute the same complexity of update operations. To generate all possible alternatives and later choose the best alternative is time consuming. This is because the number of possible alternatives increases as the number of update operations and the number of available processors increase.

Example: Referring to the example given in Figure 2, if there are two processors, then each processor will be given approximately 5/2 operations. The transaction can be split as follows (note: other alternatives are also possible).

Alternative 1: ST1{3, 6, 7} and ST2{4, 5}
Alternative 2: ST1{3, 4, 5} and ST2{6, 7}
Alternative 3: ST1{5, 6} and ST2{3, 4, 7}

For simplicity purposes we have simplify the presentation of the transaction. Here STi is the name of the subtransaction and the numbers in the set represent the operations as in Figure 2.

ii) Complexity of independent operations – although strategy i) will allocate on average the same number of independent operations to all processors, but the actual workload each processor will perform might be different due to the complexity of the operations. For this, we proposed complexity weight to be given to each operation based on its complexity, as follows:

	Complexity weight (CW)
(a) Modify multiple tuples	4
(b) Delete multiple tuples	3
(c) Modify single tuple	2
(d) Insert/Delete single tuple	1

(e) Control structure (if construct) is the average of performing the operations specified in it.

The total complexity for a transaction, $TC = \sum_{i=1}^{n} \text{CW}(\text{op}_i)$ where *n* is the number of independent operations (op). Based on the total complexity, each processor will be given a number of operations with total complexity of *TC*/*m* where *m* is the number of processors.

This strategy is easy to implement and each processor is given approximately the same complexity of update operations and therefore the time spend to execute a subtransaction by each processor is more or less the same. Although each processor might be given different number of update operations but this is not a critical factor that influences the performance of the parallel system.

Example: Referring to the example given in Figure 2, the total complexity, *TC*, is 7.5. If there are two processors, then each processor will be given a set of operations with total complexity of 7.5/2 = 3.7 (approximately 4). The transaction can be split as follows:

Alternative 4: ST1{3, 4} and ST2{5, 6, 7}
Alternative 5: ST1{3, 5, 6} and ST2{4, 7}
Alternative 6: ST1{3, 7} and ST2{4, 5, 6}

Note that each subtransaction has the total complexity of 3.5 or 4.

iii) The location of the relation (for case of distributed database) – this also plays an important factor to decide how to decompose a transaction. Subtransactions can be derived based on the relations involved in the transactions. Those relations that are allocated at the same site and are specified in the transaction can be grouped in the same subtransaction. This is to minimise data transfer across the network.

Example: Assuming that *Person*, *Company* and *Offering* are allocated at site 1 and *Placement*, *Application* and *Job* at site 2 then the transaction can be split as follows:

Alternative 7: ST1{5, 6} and ST2{3, 4, 7}

Note that ST1 (ST2, respectively) can be executed locally at site 1 (2, respectively).

This strategy focuses on a way to reduce the amount of data transferred across the network which is important in a distributed database. Each processor is not necessarily given the same number and complexity of update operations. The above strategies can be integrated, for example if each operation has the same complexity then the first strategy can be applied.

5. Evaluation

In this section, we will compare the initial transaction against the subtransactions that are derived using different strategies as presented in Section 4. Our discussion will be based on the following parameters which can indirectly represent the performance of the system during the execution of the transactions/subtransactions. These parameters are:

i) *TC* provides an estimate of the total complexity of the transaction/subtransaction, which is related to the type of update operations. This measurement indirectly indicates the workload a processor is given, i.e. the more complex the update operation the more time is required by the processor to execute the operation. It is based on the formula given in Section 4.

ii) *n* is the number of update operations (independent operations) in a transaction/subtransaction.

iii) *S* provides a rough measurement of the amount of nonlocal access necessary to perform the update operations. This is measured by analyzing the number of sites that might be involved in executing the transaction/subtransaction.

Table 6 summarises the estimation of the complexity, the number of update operations and the number of sites involved during the execution of Transaction *Hire* and its subtransactions that are derived using different strategies as presented in Section 4. Referring to Table 1, we can conclude that:

i) Applying strategy i) alone, each processor will be given approximately the same number of update operations but not necessarily the same complexity of update operations (compare Alternative 1 and Alternative 3). Also, in most cases the number of sites involved (if the number of update operations > 1) is more than 1.

ii) Applying strategy ii) alone, each processor will be given approximately the same complexity of update operations. The number of update operations is not important as long as each processor is given approximately the same workload. But again only sometimes the derived subtransactions can be executed locally.

Table 6: Estimation of the complexity of the transaction and subtransactions, the number of update operations and the number of sites involved – 2 processors

Transaction/ subtransaction	*TC*		*n*		*S*					
					Worst case[1]		Best case[2]		Moderate case[3]	
Hire	7.5		5		5		1		2	
Strategy i)	ST1	ST2	ST1	ST2	ST1	ST2	ST1	ST2	ST1	ST2
Alternative 1	5	2.5	3	2	3	2	1	1	2	2
Alternative 2	4.5	3	3	2	3	2	1	1	2	2
Alternative 3	2	5.5	2	3	2	3	1	1	1	1
Strategy ii)	ST1	ST2	ST1	ST2	ST1	ST2	ST1	ST2	ST1	ST2
Alternative 4	3.5	4	2	3	2	3	1	1	1	2
Alternative 5	4	3.5	3	2	3	2	1	1	2	1
Alternative 6	4	3.5	2	3	2	3	1	1	1	2
Strategy iii)	ST1	ST2	ST1	ST2					ST1	ST2
Alternative 7	2	5.5	2	3	-	-	-	-	1	1

Note: [1] where each relation is allocated at different sites of the network
[2] where each site of the network has a copy of all the relations
[3] where *Person*, *Company* and *Offering* are allocated at site 1 and *Placement*, *Application* and *Job* at site 2

iii) Applying strategy iii) alone, each subtransaction can be executed locally but not necessarily that each processor will be given the same complexity of update operations (the same workload) (refer to Alternative 7).

Therefore, the best strategy will be to combine the above three strategies, where each subtransaction can be executed locally and each processor is given approximately the same complexity and the same number of update operations. Based on the results presented in Table 1, it is clear that executing the subtransactions by several processors can reduce the execution time as each processor will be given *TC/m* complexity of the transaction where *m* is the number of processors available. Also each processor will handle approximately *n*/*m* number of update operations instead of *m* update operations. Apart from that, for the case of distributed database we can always minimize the number of sites involved in executing the subtransactions and therefore reduce the amount of data transferred across the network. Below is the algorithm which applies the above strategies to generate subtransactions:

```
INPUT:  transaction $T_I$
OUTPUT:  subtransactions, $STi$

Let Uj(R) denotes the relation R as specified in the update operation Uj and $S_k(R)$ denotes the site $S_k$ where R is located
BEGIN
  FOR each update operation, Uj, in $T_I$ DO
    IF $Uj(R) \cap S_k(R) \neq \{\}$, THEN $STk \leftarrow Uj(R)$
  FOR each $STm$ and $STn$ where $m \neq n$ DO
    BEGIN
      Let $OP = STm \cap STn$
        $STm = STm - OP$

        $STn = STn - OP$
      IF OP = {} THEN exit()
      IF OP = {op} THEN
      BEGIN
        Calculate TC for $STm$, TC($STm$)
        Calculate TC for $STn$, TC($STn$)
        IF TC($STm$) > TC($STn$) THEN $STn = STn \cup$ {op}
        ELSE $STm = STm \cup$ {op}
      END
      IF OP ={op1, op2, …, opj}, THEN
      BEGIN
        Calculate TC for OP, TC(OP)
        ATC = TC(OP)/2
        Get CW(op1)
        TempTC = CW(op1)
        TempST = {op1}
        FOR i = 2 to j DO
        BEGIN
          Get CW(opi)
          TempTC = TempTC + CW(opi)
          IF TempTC > ATC THEN exit()
          TempST = TempST ∪ {opi}
        END
        $STm = STm \cup$ TempST
        $STn = STn \cup$ {OP – TempST}
      END
    END
END
```

6. Conclusion

Designing efficient, safe and reliable transactions is a difficult task. This paper presents technique that can improve and produce efficient transaction by detecting the relationships between the operations in the transaction. Redundant and subsumed operations will be detected and removed from the transaction. Also, dependent operations are converted into equivalent independent operations. Since the independent operations can be executed in arbitrary order, therefore the transaction can be divided into several smaller transactions (subtransactions). Several strategies to split a transaction into subtransactions have been presented in this paper. These strategies are based on the number of independent operations, the complexity of the independent operations and the physical location of the relations which are involved in the transaction. Executing the subtransactions by several processors can reduce the execution time.

References

[1] Christof, H. and Gerhard, W. “Inter- and Intra-Transaction Parallelism in Database Systems”, Proceedings of the 14th Speedup Workshop on Parallel and Vector Computing, Zurich (Switzerland), 1993.

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

[3] Ibrahim, H. “Rules for Deriving Efficient Independent Transaction”, Proceedings of the 2nd IEEE International Conference on Information & Communication Technologies: from Theory to Applications (ICTTA’06), Damascus (Syria), 24-28 April 2006, pages 1061-1066.

[4] Ibrahim, H. “Extending Transactions with Integrity Rules for Maintaining Database Integrity”, Proceedings of International Conference on Information and Knowledge Engineering (IKE’02), Edited by Hamid R. Arabnia, Youngsong Mun and Bhanu Prasad, Computer Science Research, Education and Application Tech. (CSREA) Press, Las Vegas (USA), 24-27 June 2002, pages 341-347.

[5] 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”, Proceedings of the 22nd Very Large Databases (VLDB) Conference, Bombay (India), 1996, pages 1-12.

[6] Open Distributed Systems (ODS) Group. “A Reader in Transaction Processing”, http://www.cs.uit.no/forskning/ODS/ODSProjects/adtrans/ReaderTrans.html.

[7] Sang, H.L., Lawrence J.H., Myoung, H.K., and Yoon-Joon L. “Enforcement of Integrity Constraints against Transactions with Transition Axioms”, 16th. Annual International Computer Software and Applications, 1992, pages 162-167.

[8] 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, 1995, pages 325-363.

[9] Sushil, J., Indrakshi, R., and Paul, “A. Implementing Semantic-Based Decomposition of Transactions”, CAiSE 1997, 1997, pages 75-88.

[10] Wang, X.Y. “The Development of a Knowledge-Based Transaction Design Assistant”, PhD Thesis, Department of Computing Mathematics, University of Wales College of Cardiff, Cardiff (UK), 1992.

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

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 databases, transaction processing, and knowledge-based systems.