Journal of ICT, 3 (2), pp: 87-106

TRANSACTION DECOMPOSITION TECHNIQUE*

*H. Ibrahim

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

*E-mail: hamidah@fsktm.upm.edu.my

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. 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 faults have been identified in transactions, namely: inefficient, unsafe and unreliable. In this paper, we present a technique that can be applied to generate subtransactions to exploit parallelism. In our work, we have identified four types of relationships which can occur in a transaction. They are: redundancy, subsumption, 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.

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

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

1.0 INTRODUCTION

Parallel database systems have evolved to cope with the demands of 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 (Connolly & Begg, 2002; Ibrahim, 2002). To satisfy this goal, a transaction should have the four (ACID) properties, namely: atomicity, consistency, isolation, and durability.

Wang, (1992) has identified three types of faults 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 transactions.

Several existing techniques can be applied to overcome the various types of faults 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. Changes in the cardinality of the updated relations can be used to help transaction designers amend unsafe transactions and detect unintended results (Chakravarthy et al., 1990; Wang, 1992).

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 the transaction being aborted. 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 (ODS, 2004).

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 Christof & Gerhard (1993), Shasha et al., (1995), and Michael et al., (1996). While Sang et al., (1992), and Sushil et al., (1997) concentrate on techniques to decompose a transaction into several subtransactions.

Michael et al., (1996) 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.

Shasha et al., (1995) proposed an algorithm which is capable of generating the finest chopping of a set of transactions but his algorithm relies 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.

Christof & Gerhard (1993) presented 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.

Sushil et al., (1997) introduced the notion of semantic histories which not only listed the sequence of steps forming the history, but also conveyed 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 a transaction. Sushil et al., (1997) 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 introduced the concept of successors sets, but the successor set descriptions are obtained by examining the preconditions with auxiliary variables.

Sang et al., (1992) proposed a technique for partitioning a 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 (i) insert followed by delete (ii) delete followed by insert and (iii) insert followed by change. Also, his technique is not capable of handling more complex transactions 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 fixed; it depends on several factors as highlighted in Section 4; (ii) it does not require additional precondition checks as in Sushil et al., (1997); (iii) most of the previous works only consider transactions with simple update operations such as Sushilet al., (1997), and Sang et al., (1992); and (iv) most of the previous works assumed 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.

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 types of relationships that can be identified between the operations in a transaction. Section 4 presents a technique to generate subtransactions which can be executed in parallel. In Section 5, we analyze the generated subtransactions and we compare them to their respective initial transactions. Conclusion is presented in the final Section, 6.

2.0 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 transition 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:

- $\text{ins}(R(c_1,c_2,\ldots,c_n))$ – inserting a tuple into relation R with values $c_1,c_2,\ldots,c_n$,
- $\text{del}(R(x,\ldots))$ – 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,...,c_n):R(x,c_{n1},...,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 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
```

The 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 Wang, (1992).

```
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 in figure 2:

```
1: Transaction Hire(hiree,comp,jb,sal)
2: Begin
3:    mod(Person(hiree,_,false):Person(hiree,_,true));[1]
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
```

Fig. 2: The Hire Transaction using Our Constructs

[1] The symbol '-' indicates that the value of the column is not necessary.

3.0 IDENTIFYING RELATIONSHIP BETWEEN OPERATIONS

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 (Sang et al., 1992). 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, Sang et al., (1992) has proposed a technique to decompose a transaction into several subtransactions but his technique (i) is limited due to the reasons as described in Section 1; and (ii) does not exploit the possibility that the transaction can be improved (optimised) by removing any redundant or subsumed operation that may occur in the transaction. 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 relationships 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:

i) Redundancy:

A transaction T with n operations $op_1R1(A1),op_2R2(A2),\ldots,op_nRn(An)$ 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.

Example:

```
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));[2]
End
```

The above transaction *T3* contains a redundant operation and since there are no other operations between the redundant operations which change the state of *Placement*, therefore the first operation *del(Placement(h,c,j,s))* can be removed from the transaction.

ii) Subsumption:

A transaction T with n operations $op_1R1(A1),op_2R2(A2),\ldots,op_nRn(An)$ 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.

[2] Modify operation is considered as a sequence of delete followed by an insert operation as in McCaroll, (1995).

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 indicate that performing $op_jRj(Aj)$ will also perform $op_iRi(Ai)$.

Example:

```
Transaction T4(h,c,j,s)
Begin
    del(Placement(h,c,j,s));
    mod(Placement(_,c,_,_):Placement(_,_,_,s+100));
    End
```

The above *del(Placement(h,c,j,s))* operation is subsumed by *mod(Placement(_,c,_,_): Placement(_,_,_,s+100))* since performing *del(Placement(_,c,_,_))* will also delete the tuple <*h,c,j,s*> from *Placement*. Therefore *del(Placement(h,c,j,s))* should be removed from the transaction.

iii) Dependent/Independent:

Given a transaction T with update operations $op_1R1(A1)$, $op_2R2(A2),\ldots,op_nRn(An)$, T is *order dependent* if and only if the execution of the transaction following the serialibility order as in the transaction produces an output which will be different from 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 satisfies the conditions in Table 1.

Table 1: Converting Conflicting Updates to Non-conflicting Updates

Conflicting Updates	Equivalent Non-conflicting Updates
ins($R(c_1,c_2,\ldots,c_n)$) del($R(c_1,c_2,\ldots,c_n)$)*	nothing
ins($R(c_1,c_2,\ldots,c_n)$) del($R(\ldots,c_i,\ldots)$)	del($R(\ldots,c_i,\ldots)$)
del($R(c_1,c_2,\ldots,c_n)$)* ins($R(c_1,c_2,\ldots,c_n)$)	nothing
del($R(\ldots,c_i,\ldots)$) ins($R(c_1,c_2,\ldots,c_n)$)	del($R(x,\ldots,c_i,\ldots)$) where $x \neq c_1$
ins($R(c_1,c_2,\ldots,c_n)$) mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)*	ins($R(c_{n1},\ldots,c_{nn})$)
mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)* ins($R(c_1,c_2,\ldots,c_n)$)	ins($R(c_{n1},\ldots,c_{nn})$)
del($R(c_1,c_2,\ldots,c_n)$)* mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)*	mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)
del($R(\ldots,c_i,\ldots)$) mod($R(\ldots,c_i,\ldots):R(\ldots,c_{in},\ldots)$)	mod($R(\ldots,c_i,\ldots):R(\ldots,c_{in},\ldots)$)
mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)* del($R(c_1,c_2,\ldots,c_n)$)	mod($R(c_1,c_2,\ldots,c_n):R(c_{n1},\ldots,c_{nn})$)
mod($R(\ldots,c_i,\ldots):R(\ldots,c_{in},\ldots)$) del($R(\ldots,c_i,\ldots)$)	mod($R(\ldots,c_i,\ldots):R(\ldots,c_{in},\ldots)$)
ins($R(c_1,c_2,\ldots,c_n)$) mod($R(c_{n1},\ldots,c_{nn}):R(c_1,c_2,\ldots,c_n)$)*	not possible
mod($R(c_{n1},\ldots,c_{nn}):R(c_1,c_2,\ldots,c_n)$)* ins($R(c_1,c_2,\ldots,c_n)$)	not possible
del($R(c_1,c_2,\ldots,c_n)$)* mod($R(c_{n1},\ldots,c_{nn}):R(c_1,c_2,\ldots,c_n)$)*	del($R(c_{n1},\ldots,c_{nn})$)
del($R(\ldots,c_i,\ldots)$) mod($R(\ldots,c_{in},\ldots):R(\ldots,c_i,\ldots)$)	del($R(\ldots,c_{in},\ldots)$)
mod($R(c_{n1},\ldots,c_{nn}):R(c_1,c_2,\ldots,c_n)$)* del($R(c_1,c_2,\ldots,c_n)$)*	del($R(c_{n1},\ldots,c_{nn})$)
mod($R(\ldots,c_{in},\ldots):R(\ldots,c_i,\ldots)$) del($R(\ldots,c_i,\ldots)$)	del($R(\ldots,c_{in},\ldots)$)

*Equivalent non-conflicting updates will be derived if the operations specify only the value of the primary key.

$c_i \in \{c_2,\ldots,c_n\}$, $c_{in} \in \{c_{n2},\ldots,c_{nn}\}$
c_1 and c_{n1} are the primary key values

Table 1 shows that any two adjacent conflicting updates can be converted into a set of non-conflicting updates.

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 $j \neq i$,

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

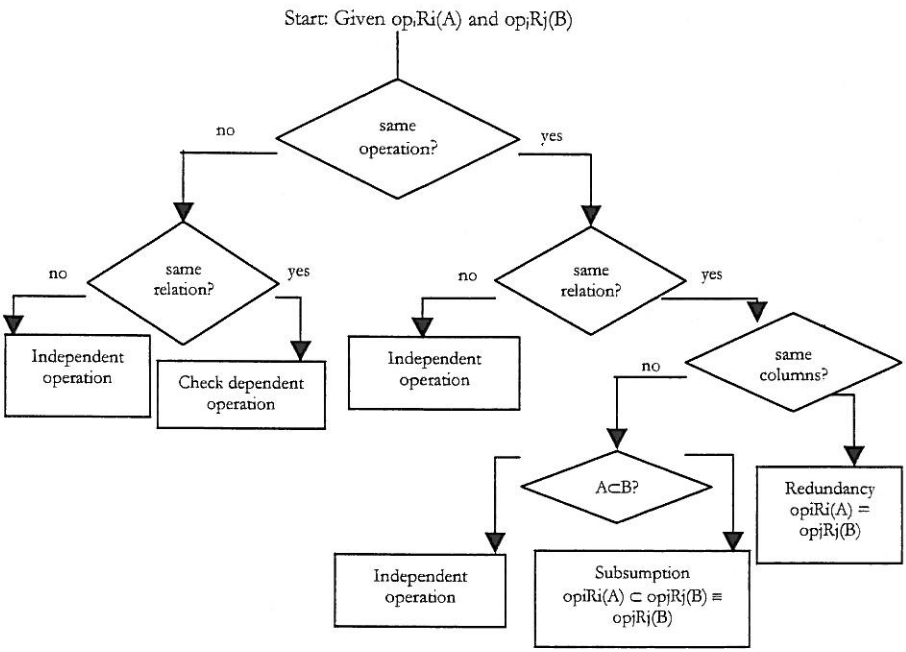

Fig. 3: Flowchart for Identifying Redundancy, Subsumption, Dependent and Independent Operations

Example:

```
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 its relative execution orders. With an order independent transaction we can consider its single updates in an arbitrary order. Table 1 can be used to transform any order dependent transaction into an equivalent order independent transaction.

Figure 3 summarises the types of relationships that can be identified between operations in a transaction by comparing the type of operation, the relations specified in the operations as well as the values.

4.0 TECHNIQUE FOR DERIVING SUBTRANSACTIONS

Given an order independent transaction T with update operations $op_1R1(A1),op_2R2(A2),\ldots,op_nRn(An)$, its single update can be considered in an arbitrary order. The transaction T can be partitioned into m disjoint subtransactions, $T1,T2,\ldots,Tm$ according to the arguments of the data item operated by the update. In our work this is performed by the *subtransaction_generating* technique. This technique consists of the following procedures:

4.1 redundancy_procedure

This procedure will check whether redundancy occurs in a given transaction based on Definition 1. The redundant operation, if it exists, will be eliminated from the transaction as it does not affect the state of the database.

4.2 subsumption_procedure

This function will check whether an operation is subsumed by another operation based on Definition 2. Similar to redundancy_procedure, this function will eliminate any operation whose effect is the same as performing its subsumed pair.

4.3 dependent_and_independent_procedure

Based on Definition 3 and 4, this procedure will detect dependent operations and by referring to Table 1, it will convert the dependent operations into independent operations. A truth table is constructed if control structure is part of the transaction.

Example:

```
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
```

Table 2: Truth Table

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

[3] Note that if the 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.

The above truth table is derived based on the truth values of the conditions specified in the if construct. For each possibility, an equivalent independent operation will be generated.

Once the relationships among the operations have been detected and the transaction has been optimised[4] (eliminate redundancy, subsumption and convert 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 be involved 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.

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:[5]

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 simplified 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.

[4] The word optimised here refers to a transaction which does not contain redundant components or subsumption operations as used by many authors (Wang, 1992 ; Ibrahim, 2002).

[5] Other alternatives are also possible.

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
(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, $TC = \sum_{i=1}^{n}$ complexity weight(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.

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 in deciding 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).

The above strategies can be integrated, for example if each operation has the same complexity then the first strategy can be applied.

5.0 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 2 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 2, 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 operation 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.

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).

Table 3: 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

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 2, 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.

6.0 CONCLUSION

Designing efficient, safe and reliable transactions is a difficult task. This paper presents a technique that can improve and produce an 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 an 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.

7.0 REFERENCES

Chakravarthy, U.S., Grant, J., and Minker, J. (1990). Logic-based Approach to Semantic Query Optimization. *ACM TODS, Vol. 15*, No. 2, pp. 162-207.

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

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

Ibrahim, H. (2002). 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, pp. 341-347.

McCaroll, N.F. (1995). Semantic Integrity Enforcement in Parallel Database Machines. *PhD Thesis*, Department of Computer Science, University of Sheffield, Sheffield, UK.

Michael, R., Moira, C. N., and Hans-Jorg, S. (1996). 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), pages 1-12.

ODS. (2004). Open Distributed Systems (ODS) Group. A Reader in Transaction Processing.http://www.cs.uit.no/forskning/ODS/ODSProjects/adtrans /ReaderTrans.html.

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

Shasha, D., Llirbat, F., Simon, E., and Valduriez, P. (1995). Transaction Chopping: Algorithms and Performances Studies. *Journal of ACM Transaction Database Systems*, Vol. 20, No. 3, pages 325-363.

Sushil, J., Indrakshi, R., and Paul, A. (1997). Implementing Semantic-Based Decomposition of Transactions. *CAiSE 1997*, pages 75-88.

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