Simple Rational Guidance for Chopping Up Transactions

Dennis Shasha
Courant Institute, New York University

Eric Simon and Patrick Valduriez
Projet Rodin, INRIA, Rocquencourt
shasha@cs.nyu.edu, Eric.Simon@inria.fr, Patrick.Valduriez@inria.fr

ABSTRACT

Chopping transactions into pieces is good for performance but may lead to non-serializable executions. Many researchers have reacted to this fact by either inventing new concurrency control mechanisms, weakening serializability, or both. We adopt a different approach.

We assume a user who

- *has only the degree 2 and degree 3 consistency options offered by the vast majority of conventional database systems; and*
- *knows the set of transactions that may run during a certain interval (users are likely to have such knowledge for online or real-time transactional applications).*

Given this information, our algorithm finds the finest partitioning of a set of transactions TranSet with the following property: if the partitioned transactions execute serializably, then TranSet executes serializably. *This permits users to obtain more concurrency while preserving correctness. Besides obtaining more inter-transaction concurrency, chopping transactions in this way can enhance intra-transaction parallelism.*

The algorithm is inexpensive, running in $O(n \times (e + m))$ time using a naive implementation where n is the number of concurrent transactions in the interval, e is the number of edges in the conflict graph among the transactions, and m is the maximum number of accesses of any transaction. This makes it feasible to add as a tuning knob to practical systems. [1]

[1]Supported by U.S. Office of Naval Research #N00014-91-J-1472, U.S. National Science Foundation grants #IRI-89-01699 and #CCR-9103953. Work done while Shasha was at INRIA.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0298...$1.50

1 Motivation

The database research literature has many excellent papers describing new concurrency control methods. These methods aim to help database management system designers to build better concurrency control methods into their systems. However, the fact remains that the vast majority of commercial database systems use two phase locking to enforce the highest degree of isolation (i.e. serializability) and less restrictive locking methods to enforce lower degrees of isolation (e.g. two phase locking for write locks, but immediate release of read locks). So, it is of significant practical interest to find ways to reduce concurrent contention given just those mechanisms, as the first author discovered when writing a book about database tuning.

Performance consultants and tuning guides give a simple way: "Shorten transactions or use less restrictive locking methods whenever you can. Serializability is an overly strict constraint in any case."

Database administrators and users follow this advice. The trouble is that problems can then crop up mysteriously into applications previously thought to be correct.

Example 1 – The Length of a Transaction

Suppose that an application program processes a purchase by adding the value of the item to inventory and subtracting the money paid from cash. The application specification requires that cash never be made negative, so the transaction will roll back (i.e., undo its effects) if subtracting the money from cash will cause the cash balance to become negative.

To improve performance, the application designers divide these two steps into two transactions.

1. The first transaction checks to see whether there is enough cash to pay for the item. If so, the first transaction adds the value of the item to inventory. Otherwise, abort the purchase.
2. The second transaction subtracts the value of the item from cash.

They find that the cash field occasionally became negative. The following scenario shows why. There is \$100 in cash available when a first application program begins to execute. The item to be purchased costs \$75. So, the first transaction commits. Then some other execution of this application program causes \$50 to be removed from cash. When the first execution of the program commits its second transaction, cash will be in deficit by \$25.

End of Example 1 – The Length of a Transaction

So, dividing the application into two transactions can result in an inconsistent database state. Once seen, the problem is obvious, though no amount of sequential testing would have revealed it. Most concurrent testing would not have revealed it either, since the problem occurs rarely.

The above example comes as no surprise to concurrency control afficionados. A little more surprising is how slightly the example must be changed to make everything work well.

Example 2 – Variant on the Length of a Transaction

Suppose that we rearrange the purchase application to check the cash level and decrement it in the first step if the decrement won't make it go negative. In the second step, we add the value of the item to inventory. We make each step a transaction:

1. The first transaction checks to see whether there is enough cash to pay for the item. If so, the first transaction decrements cash by the price of the item. Otherwise, abort the purchase.

2. The second transaction adds the price of the item to inventory.

Using this scheme, cash will never become negative and any execution of purchase applications will appear to execute as if each purchase transaction executed serially.

End of Example 2

Our goal is to help practitioners shorten lock times without sacrificing serializability. We will not propose a new concurrency control algorithm, but will implicitly assume that two phase locking is used.

Surprisingly, the results are quite strong and compare favorably with some of the semantic concurrency control methods proposed elsewhere. The algorithm is efficient. Given conflict information the algorithm requires time running in $O(n \times (e+m))$ time using a naive implementation where n is the number of concurrent transactions in the interval, e is the number of edges in the conflict graph among the transactions, and m is the maximum number of accesses of any transaction.

2 Assumptions

To use this technique, the database user must have certain knowledge.

- The database system user (here, that means an administrator or a sophisticated application developer) can characterize all the transactions that will run in some interval.

  The characterization may be parametrized. For example, the user may know that some transactions update account balances and branch balances, whereas others check account balances. However, the user may not know exactly which accounts or branches will be updated.

- The goal is to achieve the guarantees of full isolation (degree 3 consistency) — without paying for it.

  That is, the user would like either to use degree 2 consistency (i.e. write locks are acquired in a two phased manner but read locks are released immediately after use) or to chop transactions into smaller pieces. The guarantee should be that the resulting execution be equivalent to one in which each original transaction executes in isolation.

- If a transaction makes one or more calls to rollback, the user knows when these occur.

  Suppose that the user chops up the code for a transaction T into two pieces T_1 and T_2 where the T_1 part executes first. If the T_2 part executes a rollback statement in a given execution after T_1 commits, then the modifications done by T_1 will still be reflected in the database. This is not equivalent to an execution in which T executes a rollback statement and undoes all its modifications. Thus, the user should rearrange the code so rollbacks occur early. We will formalize this intuition below with the notion of rollback-safety.

- If a failure occurs, it is possible to determine which transactions completed before the failure and which ones did not. This will permit the user or system to reexecute those transactions.

Suppose there are n transactions $T_1, T_2, \ldots, T_n$ that can execute within some interval. Let us assume, for now, that each such transaction results from a distinct program. Chopping a transaction will then consist of modifying the unique program that the transaction executes. Because of the form of the chopping algorithm, this assumption will turn out to have no effect on the result.

A *chopping* partitions each T_i into *pieces* $c_{i_1}, c_{i_2}, \ldots c_{i_k}$. That is, every database access performed by T_i is in exactly one piece.

A chopping of a transaction T is said to be *rollback-safe* if either T has no rollback statements or all the rollback statements of T are in its first piece. The first piece must have the property that all its statements execute before any other statements of T. (As we will see, this will prevent a transaction from half-committing and then rolling back.)

A chopping is said to be *rollback-safe* if each of its transactions is rollback-safe.

Two special cases of choppings are of particular interest:

- The transaction T is sequential and the pieces are non-overlapping subsequences of that transaction.

  For example, suppose T updates an account balance and then updates a branch balance. Each update might become a separate piece, acting as a separate transaction.

- The transaction T operates at degree 2 consistency in which read locks are released as soon as reads complete.

  In this case, each read by itself constitutes a piece.[2] All writes together form a piece (because the locks for the writes are only released when T completes).

Execution Rules: (for the pieces of a chopping)

1. When pieces execute, they obey the dependency order imposed by the transaction program text. [3]

2. Each piece will acquire locks according to the two phase locking algorithm and will release them when it ends. It will also commit its changes when it ends.

3. If a piece is aborted due to a lock conflict, then it will be resubmitted repeatedly until it commits.

4. If a piece is aborted due to a rollback statement, then pieces for that transaction that have not begun will not execute.

3 When is a Chopping Correct?

We will characterize the correctness of a chopping with the aid of an undirected graph having two kinds of edges:

1. C edges — C stands for *conflict*. Two pieces p and p' from different original transactions conflict if there is some data item x that both access and at least one modifies.[4] In this case, draw an edge between p and p' and label the edge C.

2. S edges — S stands for *sibling*. Two pieces p and p' are siblings if they come from the same transaction T. In this case, draw an edge between p and p' and label the edge S.

We call the resulting graph the *chopping graph*. (Note that no edge can have both an S and a C label.)

We say that a chopping graph has an *SC-cycle* if it contains a simple cycle that includes at least one S edge and at least one C edge.[5]

We say that a chopping of $T_1, T_2, \ldots, T_n$ is *correct* if any execution of the chopping that obeys the execution rules is equivalent to some serial execution of the original transactions.

"Equivalent" is in the sense of the textbook[1]. That is, every read (resp. write) from every transaction returns (resp. writes) the same value in the two executions and the same transactions roll back. Now, we can prove the following theorem.

Theorem 1: A chopping is correct if it is rollback-safe and its chopping graph contains no SC-cycle.

PROOF.
The proof requires the properties of a serialization graph. Formally, a serialization graph is a directed graph whose nodes are transactions and whose directed edges represent ordered conflicts. That is, $T \rightarrow T'$ if T and T' both access some data item x, one of them modifies x and T accessed x first. Following[1], if the serialization graph resulting from an execution is acyclic, then the execution is equivalent to a serial one. Further the book proves the following fact.

Fact: (†) If all transactions use two phase locking, then all those who commit produce an acyclic serialization graph.

[2] Technically, this needs some qualification. Each read that doesn't follow a write on the same data item constitutes a piece. The reason for the restriction is that if a write(x) precedes a read(x), then the transaction will continue to hold the lock on x after the read completes.

[3] For example, if the transaction updates account X first and branch balance B second, then the piece that updates account X should complete before the piece that updates branch balance B begins.

[4] As has been observed repeatedly in the literature, this notion of conflict is too strong. For example, if the only data item in common between two transactions is one that is only incremented and whose exact value is insignificant, then such a conflict might be ignored. We assume the simpler read-write model only for the purposes of exposition.

[5] Recall that a simple cycle consists of

1. a sequence of nodes $n_1, n_2, \ldots, n_k$ such that no node is repeated and
2. a collection of associated edges: there is an edge between n_i and n_{i+1} for $1 <= i < k$ and an edge between n_k and n_1; no edge is included twice.

Call any execution of a chopping for which the chopping graph contains no SC-cycles an *SC-acyclic execution* of a chopping. We must show that:

1. any SC-acyclic execution yields an acyclic serialization graph on the given transactions $T_1, T_2, \ldots, T_n$. and hence is equivalent to a serial execution of committed transactions; and

2. the transactions that roll back in the SC-acyclic execution would also roll back if properly placed in the equivalent serial execution. We need this to avoid the trivial result that the execution is serializable by being equivalent to a null execution.

For point 1, we proceed by contradiction. Consider an SC-acyclic execution of a chopping of $T_1, T_2, \ldots, T_n$. Suppose there were a cycle in the serialization graph of $T_1, T_2, \ldots, T_n$ resulting from this execution. That is $T_i \rightarrow T_j \rightarrow \ldots \rightarrow T_i$. Identify the pieces of the chopping associated with each transaction that are involved in this cycle: $p \rightarrow p' \rightarrow \ldots \rightarrow p''$. Both p and p'' belong to transaction T_i. Pieces p and p'' cannot be the same, since each piece uses two phase locking by the execution rules and the serialization graph of a set of committed two-phase locked transactions is acyclic by fact (†). Since p and p'' are different pieces in the same transaction T_i, there is an S-edge between them in the chopping graph. Every directed edge in the serialization graph cycle corresponds to a C-edge in the chopping graph since it reflects a conflict. So, the cycle in the serialization graph implies the existence of an SC-cycle in the chopping graph, a contradiction.

For point 2, notice that any transaction T whose first piece p rolls back in the SC-acyclic execution will have no effect on the database, since the chopping is rollback-safe. We want to show that T would also roll back if properly placed in the equivalent serial execution. Suppose that p conflicts with and follows pieces from the set of transactions $W_1, \ldots W_k$. Then place T immediately after the last of those transactions in the equivalent serial execution. In that case, the first reads of T will be exactly those of the first reads of p. Since p rolls back, so will T. □

Theorem 1 shows that the goal of any chopping of a set of transactions should be to obtain a rollback-safe chopping without an SC-cycle.

Chopping Graph Example 1

Suppose there are three transactions that can abstractly be characterized as follows:

```
T1: R(x) W(x) R(y) W(y)

T2: R(x) W(x)

T3: R(y) W(y)
```

Breaking up T1 into

```
T11: R(x) W(x)

T12: R(y) W(y)
```

will result in a graph without an SC-cycle (see Figure 1).

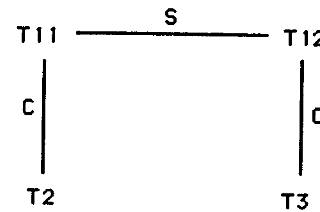

Figure 1:

Chopping Graph Example 2

With the same T2 and T3 as above, breaking up T11 further into

```
T111: R(x)

T112: W(x)
```

will result in an SC-cycle (see Figure 2).

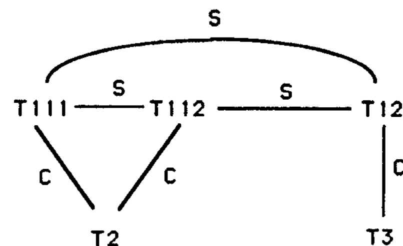

Figure 2:

Chopping Graph Example 3

Now, let us consider an example in which there are three types of transactions:

- A transaction that updates a single depositor's account and the depositor's corresponding branch balance.

- A transaction that reads a depositor's account balance.

- A transaction that compares the sum of the depositors' account balances with the sum of the branch balances.

For purposes of concreteness, consider the following transactions. Suppose that depositor accounts D11, D12, and D13 all belong to branch B1; depositor accounts D21 and D22 both belong to B2. Here are the transactions.

```
T1 (update account): RW(D11) RW(B1)

T2 (update account): RW(D13) RW(B1)

T3 (update account): RW(D21) RW(B2)

T4 (balance): R(D12)

T5 (balance): R(D21)

T6 (comparison): R(D11) R(D12) R(D13) R(B1)
R(D21) R(D22) R(B2)
```

Thus, T6 is the balance comparison transaction. Let us see first whether T6 can be broken up into two transactions.

```
T61: R(D11) R(D12) R(D13) R(B1)

T62: R(D21) R(D22) R(B2)
```

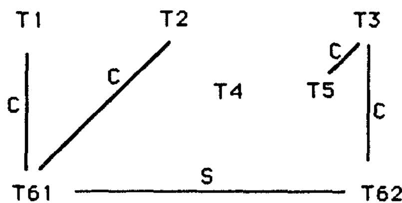

Figure 3:

The lack of an SC-cycle shows that this is possible (see Figure 3). Note that this could be generalized to u updates, b balance transactions and 1 comparison. Each balance transaction would conflict with some update transaction. Each update transaction would conflict with exactly one piece of the branch-by-branch chopping of the comparison transaction. So, there would be no cycles.

Chopping Graph Example 5

Taking the transaction population from the previous example, let us now consider dividing T1 into two transactions giving the following transaction population. Please see Figure 4.

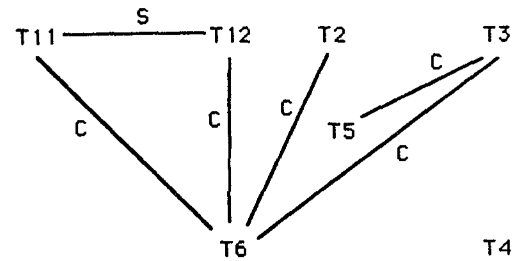

Figure 4:

```
T11: RW(D11)

T12: RW(B1)

T2: RW(D13) RW(B1)

T3: RW(D21) RW(B2)

T4: R(D12)

T5: R(D21)

T6: R(D11) R(D12) R(D13) R(B1) R(D21)
    R(D22) R(B2)
```

This results in an SC-cycle.

Remark about Order-Preservation: The choppings we offer are serializable, but not necessarily order-preserving serializable. Consider the following example:

```
T1: R(A) R(B)

T2: RW(A)

T3: RW(B)
```

The chopping graph remains acyclic if we chop up T1 into the transaction R(A) and the transaction R(B). This would allow the following execution:

```
R(A) RW(A) RW(B) R(B)
```

This is equivalent to

```
T3 T1 T2
```

so is serializable. It is not, however, order-preserving serializable, because T2 executed before T3 yet appears to execute after T3 in the only equivalent serial schedule.

4 Finding the Finest Chopping

On the way to discovering an algorithm, we must answer two particularly worrisome questions:

1. Can chopping a piece into smaller pieces break an SC-cycle?

2. Can chopping one transaction prevent one from chopping another?

Remarkably, the answer to both questions is negative.

Lemma 1: If a chopping is not correct, then any further chopping of any of the transactions will not render it correct.

PROOF.

Let p be a piece of a transaction T to be further chopped and let the result of the chopping be called pieces(p). If p is not in an SC-cycle, then chopping p will have no effect on the cycle. If p is in an SC-cycle, then there are three cases:

1. If there are two C edges touching p from the cycle, then each edge will touch exactly one piece in pieces(p). Since all pieces in pieces(p) are connected by S edges they all belong to an SC-cycle.

2. If there are one C edge and one S edge touching p, then the C edge will be connected to one piece p' of pieces(p). Since p and p' are connected by an S edge, they belong to an SC-cycle.

3. If there are two S edges touching p, then these edges will touch each piece of pieces(p).

□

Lemma 2: Suppose that in some chopping $chop_1$, two pieces, say p and p', of transaction T are in an SC-cycle. Then p and p' will also be in an SC-cycle in chopping $chop_2$ where $chop_2$ is identical to $chop_1$ with regard to transaction T, but in which no other transaction is chopped (i.e., all other transactions are represented by a single piece).

PROOF.

Since p and p' come from T there is an S-edge between them in both $chop_1$ and $chop_2$. Since they are in an SC-cycle, there exists at least one piece p'' of some transaction T' in that cycle. Merging all pieces of T' into a single piece (i.e., T') can only shorten the length of the cycle. The argument applies to every transaction other than T having pieces in the cycle. □

Figure 5 illustrates this lemma. putting the three pieces of T3 into one will not make the chopping of T1 OK, Nor will chopping T3 further.

These two lemmas lead directly to a systematic method for chopping transactions as finely as possible.

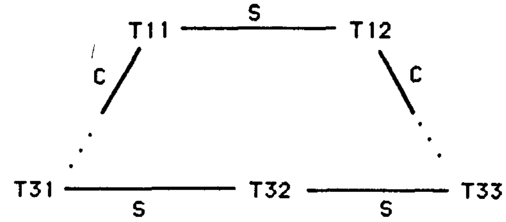

Figure 5:

Consider again the set of transactions that can run in this interval $\{T_1, T_2, \ldots, T_n\}$. We will take each transaction T_i in turn. We call $\{c_1, c_2, \ldots, c_k\}$ a *private chopping* of T_i, denoted private(T_i), if

1. $\{c_1, c_2, \ldots, c_k\}$ is a rollback-safe chopping of T_i; and

2. there is no SC-cycle in the graph whose nodes are $\{T_1, \ldots, T_{i-1}, c_1, c_2, \ldots, c_k, T_{i+1}, \ldots, T_n\}$.

   That is, the graph of all other transactions plus the chopping of T_i.

Theorem 2: The chopping consisting of $\{private(T_1), private(T_2) \ldots, private(T_n)\}$ is rollback-safe and has no SC-cycles.

PROOF.

- Rollback-safe: the chopping is rollback-safe because all its constituents are rollback-safe.

- No SC-cycles: if there were an SC-cycle that involved two pieces of private(T_i,) then Lemma 2 implies that the cycle is still present even if all other transactions are not chopped. But that contradicts the definition of private(T_i).

□

Theorem 2 implies that if we can discover a fine-granularity private(T_i) for each T_i, then we can just take their union. Formally, the *finest chopping* of T_i (whose existence we will prove) is

- a private chopping of T_i;

- if piece p is a member of this private chopping, then there is no other private chopping of T_i containing p_1 and p_2 where p_1 and p_2 partition p and neither is empty.

That is, we would have the following algorithm:

```
procedure chop (T_1, ..., T_n)
    for each T_i
        Fine_i := finest chopping of T_i
    end for;
    the finest chopping is
        {Fine_1, Fine_2, ..., Fine_n}
```

We now give an algorithm to find the finest private chopping of T.

Algorithm FineChop:
initialization:
  if there are rollback statements then
    p_1 := all database writes of T that may occur before or concurrently with any rollback statement in T
  else
    p_1 := set consisting of the first database access
  end
  $P := \{\{x\} \mid x$ is a database access not in $p_1\}$;
  $P := P \cup \{p_1\}$;

merging pieces:
  construct the connected components of the graph induced by C edges on all transactions besides T and on the pieces in $P = \{p_1, ..., p_r\}$;

update P based on the following rule:
  for each connected component, if $p_{e1}, p_{e2}, ..., p_{ek}$ are k pieces of P such that $1 < e1 < e2 < ... < ek < r$, then put all accesses of $p_{e1}, p_{e2}, ..., p_{ek}$ into p_{e1} and then remove $p_{e2}, ..., p_{ek}$.

  call the resulting partition FineChop(T)

Figure 6 shows an example of a fine-chopping of transaction T5 given a certain set of conflicts. Since there are no rollback statements, each piece starts off being a single access. Assuming no rollback statements, T5 can be "fine- chopped" into $\{\{a\}, \{b, d, e, f\}, \{c\}\}$. If $\{b, d, e, f\}$ were subdivided further, there would be an SC-cycle in the chopping graph.

Note on Efficiency: The expensive part of the algorithm is finding the connected components of the graph induced by C on all transactions besides T and the pieces in P. We have assumed a naive implementation in which the connected components are recomputed for each transaction T at a cost of $O(e + m)$ time in the worst case, where e is the number of C edges in the transaction graph and m is the size of P. Since there are n transactions, the total time is $O(n(e + m))$.

Note on Shared Code: Suppose that T_i and T_j result from the same program P. Since the chopping is implemented by changing P, transactions T_i and T_j must be chopped in the same way. This may seem surprising at first,

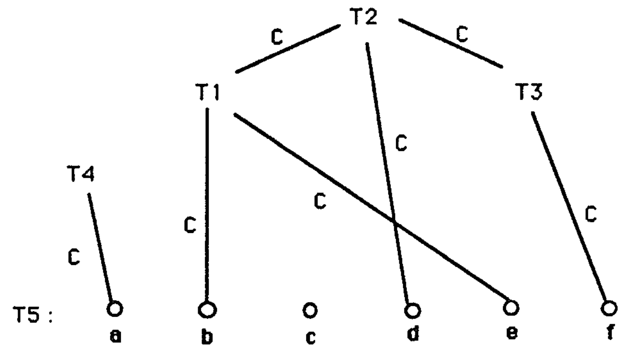

Figure 6:

but the above algorithm will give the result that FineChop(T_i) = FineChop(T_j) even if they did not share code. The reason is that the two transactions are treated symmetrically by the algorithm. When FineChop(T_i) runs, T_j is treated as unchopped and similarly for T_j. Thus, shared code does not change this result at all.

Theorem 3: FineChop(T) is the finest chopping of T.

PROOF.
We must prove two things: FineChop(T) is a private chopping of T and it is the finest one.

- FineChop(T) is a private chopping of T:
  1. Rollback-safety: by inspection of the algorithm. The initialization step creates a rollback-safe partition. The merging step can only cause p_1 to become larger.
  2. No SC-cycles: any such cycle would involve a path through the conflict graph between two distinct pieces from *FineChop*(T). The merging step would have merged any two such pieces to a single one.

- No piece of FineChop(T) can be further chopped:

  Suppose p is a piece in FineChop(T). Suppose there were a private chopping *TooFine* of T that partitions p into two non-empty subsets q and r. Since p contains at least two accesses, the accesses of q and r could come from two different sources.

  1. Piece p is the first piece, i.e., p_1, and q and r each contain accesses of p_1 as constructed in the initialization step. In that case, p_1 contains one or more rollback statements. So, one of q or r may commit before the other rolls back by construction of p_1. This would violate rollback safety.

2. The accesses in q and r result from the merging step. In that case, there is a path consisting of C-edges through the other transactions from q to r. This implies the existence of an SC-cycle for chopping *TooFine*.

□

5 Applying these Results to Typical Database Systems

For us, a typical database system will be one running SQL. Our main problem is to figure out what conflicts with what. Because of the existence of bind variables, it will be unclear whether a transaction that updates the account of customer :x will access the same record as a transaction that reads the account of customer :y. So, we will have to be conservative.

We can use the tricks of typical predicate locking schemes as pioneered in System R and then elaborated in [2, 3]. For example, if two statements on relation account are both conjunctive (only AND's in the qualification) and one has the predicate

```
AND name LIKE 'T%'
```

whereas the other has the predicate

```
AND name LIKE 'S%'
```

they clearly will not conflict at the logical data item level. (This is the only level that matters since that is the only level that affects the return value to the user.) Detecting the absence of conflicts between two qualifications is the province of compiler writers. We offer nothing new.

The only new idea we have to offer is that we can make use of information in addition to simple conflict information. For example, if there is an update on the account table with a conjunctive qualification and one of the predicates is

```
AND acctnum = :x
```

then, if acctnum is a key, we know that the update will access at most one record. This will mean that a concurrent reader of the form

```
SELECT ...
FROM account
WHERE ...
```

will conflict with the update on at most one record, a single data item, so can execute at level 2 isolation.

In fact, even if many updates of this form are concurrent with the reader, the reader can be chopped in this way.

We will label each transaction with the label with values "1" or "many" for each relation that it accesses. For example, the update above has the label "1, account," whereas the read access has the label "many, account." We will label the conflict edges similarly. So, the conflict edge will have the label "1, account." Of course, there can be many labels on each conflict edge.

Now, we say that a transaction T that may contain binding variables is ***bind-chop-safe*** if one of the following two conditions holds:

1. Transaction T conflicts with a single other transaction and their conflict edge has a single label decorated with a 1. Or,

2. Transaction T may conflict with many other transactions. However for any cycle c (consisting of conflict edges only) that touches T, there must be a relation R such that

   - every edge e in c has label (1,R) as its only label; and
   - every transaction T' other than T in c associates 1 with R (i.e. has label (1,R)), though T' may have other labels; T may have (many, R) as part of its label.

Figure 7) portrays transactions with their decorated conflict edges. T1 can be executed at degree 2, yet will appear to execute at degree 3.

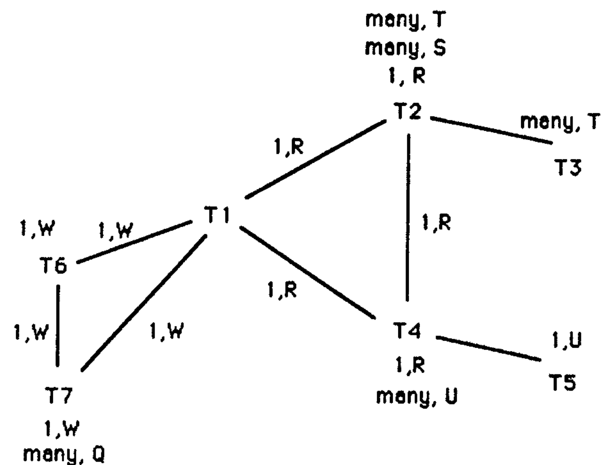

Figure 7:

Theorem 4: If transaction T is bind-chop-safe, then T can be chopped into a set of pieces such that

- each piece holds at least one database access;
- the first piece holds all rollback statements; and

- no two pieces access the same data item.

(In particular, if T is a read-only query, then it can run at degree 2 isolation.)

PROOF.
Suppose there were a cycle in the SC-graph as a result of chopping T and some values of the bind variables. Such a cycle must connect two pieces p_1 and p_2 of the chopping by a path of conflicts through other transactions. Any such path must correspond to a cycle c consisting of edges labeled (1,R) and nodes labeled (1,R) for some relation R. That implies that the conflict carried by this path is on a single data item x. However p_1 and p_2 access different data items by construction. This contradiction implies that no such cycle is possible. □

6 Related Work

There is a rich body of work in the literature on the subject of chopping up transactions or changing concurrency control mechanisms, some of which we review here, although the work is not strictly comparable.

The reason is that this paper is aimed at database users rather than database implementors. Database users normally cannot change the concurrency control algorithms of the underlying system, but must use two phase locking and its variants. Even if users could change the concurrency control algorithms, they probably should avoid doing so as the bugs that might result can easily corrupt a system.

The literature offers many good ideas however. Here is a brief summary of some of the major contributions.

Farrag and Ozsu[4] consider the possibility of chopping up transactions by using "semantic" knowledge and a new locking mechanism. For example, consider a hotel reservations system that supports a single transaction Reserve. Reserve performs the following two steps:

1. Decrement the number of available rooms or roll back if that number is already 0.

2. Find a free room and allocate it to a guest.

If reservation transactions are the only ones running, then the authors assert that each reservation can be broken up into two transactions, one for each step.

Hector Garcia-Molina[5] suggested using semantics by partitioning transactions into classes. Transactions in the same class can run concurrently, whereas transactions in different classes must synchronize. He proposes using semantic notions of consistency to allow more concurrency than serializability would allow and using counterstep transactions to undo the effect of transactions that should not have committed.

Nancy Lynch[6] generalized Garcia-Molina's model by making the unit of recovery different from the unit of locking (this is also possible with the checkout/checkin model offered by some object-oriented database systems).

Rudolf Bayer[7] showed how to change the concurrency control and recovery subsystems to allow a single batch transaction to run concurrently with many short transactions.

Meichun Hsu and Arvola Chan[8] have examined special concurrency control algorithms for situations in which data is divided into raw data and derived data. The idea is that the recency of the raw data is not so important in many applications, so updates to that data should be able to proceed without being blocked by reads of that data.

Some commercial systems such as Oracle use this scheme as well of allowing reads to view old data. That facility would remove the necessity to use the algorithms in this paper for read-only transactions.

Patrick O'Neil[9] takes advantage of the commutativity of increments to release locks early even in the case of writes.

Ouri Wolfson[10] presents an algorithm for releasing certain locks early without violating serializability based on an earlier theoretical condition given by Yannakakis[11]. He assumes that the user has complete control over the acquisition and release of locks. The setting here is a special case: the user can control only how to chop up a transaction or whether to allow reads to give up their locks immediately. As mentioned above, we have restricted the user's control in this way for the simple pragmatic reason that systems restrict the user's control in the same way.

Bernstein, Shipman and Rothnie[12] introduced the idea of conflict graphs in an experimental system called SDD-1 in the late 1970's. Their system divided transactions into classes such that transactions within a class executed serially whereas transactions between classes could execute without any synchronization.

Marco Casanova's thesis[13] extended the SDD-1 work by representing each transaction by its flowchart and by generalizing the notion of conflict. A cycle in his graphs indicated the need for synchronization if it included both conflict and flow edges.

Shasha and Snir[14] explored graphs that combine conflict, program flow, and atomicity constraints in a study of the correct execution of parallel shared memory programs that have critical sections. The graphs used here are a special case of the ones used in that article.

7 Conclusion

We propose a simple, efficient algorithm to partition transactions into the smallest pieces possible with the

following property:

- If the small pieces execute serializably, then the transactions will appear to execute serializably.

This permits database users to obtain more concurrency and intra-transaction parallelism without requiring any changes to database system locking algorithms. The only information required is some characterization of the transactions that can execute during a certain interval and the location of any rollback statements within the transaction. We sketch the application of our algorithm to SQL-based systems.

Several interesting problems remain open:

- How would this approach work with the concurrency control methods offered by some of the object-oriented systems such as optimistic methods, sagas, and the tree-locking approaches of the Kedem and Silberschatz school?
- Suppose that an administrator asks how best to partition transaction populations into time windows, so that the transactions in each window can be chopped as much as possible. For example, a good heuristic is to put global update transactions in a partition by themselves while allowing point updates to interact with global reads. What precise guidance could theory offer?
- What is a good architecture for incorporating chopping among the tuning knobs for database management system?

8 Acknowledgments

We would like to thank Gerhard Weikum for his astute comments regarding order-preserving serializability and intra-transaction parallelism, Victor Vianu for a bus ride discussion concerning transitive closure, Rick Hull for a discussion about partitioned accesses. We would also like to thank Fabienne Cirio for applying her artistry to make the figures of the manuscript.

References

[1] *Concurrency Control and Recovery in Database Systems* P. A. Bernstein, V. Hadzilacos, and N. Goodman Addison-Wesley, 1987.

[2] "Interval Hierarchies and their Application to Predicate Files" K. C. Wong and M. Edelberg ACM transactions on Database Systems, September 1977, vol. 2, no. 3, pp. 223-232

[3] "A Predicate Oriented Locking Approach for Integrated Information Systems," P. Dadam, P. Pistor, and H-J. Schek IFIP Congress, Paris, 1983, published by North-Holland, 1983.

[4] "Using Semantic Knowledge of Transactions to Increase Concurrency" Abdel Aziz Farrag and M. Tamer Ozsu ACM transactions on Database Systems, December 1989, vol. 14, no. 4, pp. 503-525

[5] "Using Semantic Knowledge for Transaction Processing in a Distributed Database" Hector Garcia-Molina ACM Transactions on Database System, June. 1983, vol. 8, no. 2, pp. 186-213.

[6] "Multi-level Atomicity — a new correctness criterion for database concurrency control" Nancy Lynch ACM Transactions on Database System, Dec. 1983, vol. 8, no. 4, pp. 484-502.

[7] "Consistency of Transactions and Random Batch" R. Bayer ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 397-404

[8] "Partitioned Two-Phase Locking" M. Hsu and A. Chan ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 431-446

[9] "The Escrow Transactional Mechanism" Patrick O'Neil ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 405-430

[10] "The Virtues of Locking by Symbolic Names" Ouri Wolfson Journal of Algorithms 1987 8, pp. 536-556, 1987

[11] "A Theory of Safe Locking Policies in Database Systems," Mihalis Yannakakis JACM 29(3), pp. 718-740, (1982).

[12] "Concurrency Control in a System for Distributed Databases (SDD-1)" P. A. Bernstein, D. W. Shipman and J. B. Rothnie ACM Transactions on Database Systems, March 1980, vol. 5, no. 1, pp. 18-51.

[13] *The Concurrency Control Problem for Database Systems* Marco Casanova Springer-Verlag Lecture Notes in Computer Science no. 116, 1981

[14] "Efficient and Correct Execution of Parallel Programs that Share Memory" D. Shasha and M. Snir ACM Transactions on Programming Languages and Systems, vol. 10, no. 2, pp. 282-312, April, 1988