Increasing Concurrency in Object-Oriented Databases Using Semantic Information: A Survey

Advanced Databases Term Paper December 1, 1993 Professor Shasha
Bob DuCharme ID 041-56-0107 m-rd0107@cs.nyu.edu

1. Introduction

Database research has developed several mechanisms for ensuring consistency of data when concurrently executing processes act on the same data. Various algorithms have been proposed and implemented that group the operations of a process into indivisible units known as transactions and then use locks on data items (which prevent other operations from using a data item while the lock is in effect) and redundant copies of data to coordinate, or "interleave" transactions' operations as they proceed. A typical problem avoided by such a scheme would be the use by a transaction *T* of data that had been stored by an aborted transaction *T'*. This is bad data, and a key goal of concurrency control is to prevent the use of such bad data by scheduling operations so that unfinished transactions have no effect on other transactions [BHG87].

To prevent this, we could block transaction *T* until *T'* has finished, thereby eliminating the possibility of *T* acting on data that turns out to be useless. But what if the use *T'* made of the data in question only meant reading it? If it didn't alter it, then the data used by *T* is good data whether *T'* used it or not, because the latter's use did not change (that is, write to) it. So, there is no need to block transaction *T*.

For this reason, only the simplest concurrency control mechanisms—generally those used for pedantic purposes [U88]—fail to distinguish between operations that write to data and those that merely read it. This distinction between the reading and writing of a data item to minimize the need to hold up transactions is an example of the use of semantic information about specific operations in order to increase concurrency.

One implication of the object-oriented principle of encapsulation is that objected-oriented database systems include the details about allowable operations on data items along with definitions of those data items. These operation definitions provide a rich source of semantic information that can help the system designer to identify situations in which transactions can proceed concurrently. For example, a traditional system would probably prevent simultaneous update by two different transactions of a given employee's salary and address, because at some level lower than the application level, a scheduler would block one process from updating that employee's record while the other was updating the appropriate field. An examination of the operations allowed on the employee object, however, will show that the final state of this object will be the same whether UpdateSalary precedes UpdateAddress, follows it, or executes concurrently with it.

The nature of this examination, and the use made of the resulting information, is what distinguishes these papers from each other. Each looks further than its predecessors for semantic information to take into consideration when deciding which circumstances require the blocking of certain operations until others finish. Part 2 of this paper describes Spector and Schwarz's method [SS84] for analyzing all the possible relations between the operations on an object and then developing a lock table to coordinate their execution. Part 3 describes Roesler and Burkhard's [RB87] use of potential operation results to minimize the number of situations in which transactions must be blocked, and part 4 examines Chrysanthis, Raghuram, and Ramamritham's methodology [CRR91] for extracting further information from an object's data structure by using a graph of its components. Part 5 examines some of the design and runtime implications of the use of semantic information to improve concurrency.

2. Schwarz and Spector, "Synchronizing Shared Abstract Data Types"

Peter Schwarz and Alfred Spector's 1984 paper [SS84] has three key parts: the first, and most important, describes their method for enumerating all possible dependencies between the operations on a given object and how to evaluate this set of *dependency relations* to find ways to improve concurrency between transactions that simultaneously use that object. Because the method described in this first part deals only with the combinations of operations that may take place on one particular object type, the second part describes how to extend this algorithm to assure serializability among concurrent transactions acting on objects of different types. The third part presents an object locking mechanism that makes use of the information derived in the first two parts to implement a concurrency control system more efficient than typical read/write lock schemes.

The authors define a dependency as the relationship between a specific operation of one transaction and a specific operation of another transaction on the same object. By enumerating all the dependency relations possible among operations on an object of a given type, we can then identify which dependency relations may play a role in concurrency problems, which they call the *proscribed* relations.

To characterize a series of `<transaction,operation>` pairs that describe the effects of these operations on a set of objects, with no concern for how a particular implementation may order the operations, they use the term *abstract schedule*. The order in which operations actually execute is known as the *invocation schedule*. While a particular abstract schedule may have many corresponding invocation schedules, a given invocation schedule corresponds to only one abstract schedule. The main point of Schwarz and Spector's paper is that if an abstract schedule is orderable with respect to the union of proscribed members of a given set of dependency relations, all invocation schedules derived from that abstract schedule are serializable. The lower the percentage of those relations that are proscribed, the greater the possible concurrency.

The first step in implementing this for a given set of operations is the creation of the dependency relation list. Its members use the notation D_i: T_j:X $\rightarrow_O T_k$:Y to represent the dependency D_i formed when transaction T_k's operation Y and transaction T_j's operation X are performed on the same object O.

For an object that can be read or written to, there are four possible dependency relations:

D_1: T_j:R $\rightarrow_O T_k$:R	T_j reads an object, then T_k reads it.
D_2: T_j:R $\rightarrow_O T_k$:W	T_j reads an object, then T_k writes to it.
D_3: T_j:W $\rightarrow_O T_k$:R	T_j writes to an object, then T_k reads it.
D_4: T_j:W $\rightarrow_O T_k$:W	T_j writes to an object, then T_k writes to it.

We refer to a specific relation as $<D_i$. For example, if T_n reads an object right after T_m writes to it, then they have the relationship $T_m <_{D_3} T_n$ because their dependency is third on the above list.

The presence or absence of the above list's relation D_1 in an abstract schedule will not make the schedule any less serializable. Such dependency relations are designated as *insignificant*. Concurrency problems are caused by a combination of the other three relations; we write the union of this set of proscribed relations as $\{<_{D_2 \cup D_3 \cup D_4}\}$. If an abstract schedule's transactions are orderable with respect to this union—in other words, if there is no T_i such that $T_i <_D T_a <_D T_b <_D \cdots <_D T_i$—then invocation schedules derived from that abstract schedule will not cause any serializability problems.

Before moving on to how this information is used in an implementation of a data type's operations, we should look at how the union of proscribed relations is constructed for a more complex data type. Spector and Schwarz describe this process with three abstract data types: a directory, a typical FIFO queue, and a "weakly" FIFO queue. The latter two demonstrate how their algorithm can be used to determine where to find greater concurrency in a slightly different version of a particular data type—in this case, they show how the weakly FIFO queue, which is not as picky about maintaining the order of entered items (as long as they are all guaranteed to eventually reach the queue's front) allows more concurrency than a strictly FIFO queue.

The directory data type is worth examining more closely, because it demonstrates generalizations that can simplify the process of coming up with a manageable set of dependency relations. Their sample directory object provides a mapping between the text strings that serve as the directory's key and capabilities for arbitrary objects. There are five possible operations that may be performed on it: DirInsert inserts a directory entry, DirDelete deletes one, DirLookup searches for a string and returns its associated capability list, and DirDump returns a list of `<string,capability>` pairs. Fortunately, five different operations don't mean 5^2 dependency relations. They categorize them into three classes:

M The Modify operations, which modify an entry (DirInsert and DirDelete).

L The Lookup operations, which return information about an existing or missing entry. In addition to DirLookup, this also includes DirInsert and DirDelete operations that fail. This demonstrates how operations whose side effects are used by program logic must be examined carefully to determine all of their possible roles in providing semantic information for concurrency control.

D Operations that provide information about more than one entry. The DirDump operation is the only one in this category.

Now the operations have been reduced to three categories, but there are more than 3^2 possible dependency relations if we make use of another kind of semantic information: the parameters passed to these operations. If a given directory entry is modified and the same one is then looked up, stricter control must be maintained over concurrent execution of these two operations than a situation in which one entry is modified and then a different one is looked up. Providing for like and differing parameters gives the $M \rightarrow L$ combination of operations two entries in the list of dependency relations, as it does to $M \rightarrow M$, $L \rightarrow M$, and $L \rightarrow L$.

This brings the total number of dependency relations up to thirteen. In the list, σ represents a key string passed to an operation as a parameter and σ' represents a different argument passed to an operation in the same dependency relation.

D_1: $T_i{:}M(\sigma) \rightarrow T_j{:}M(\sigma')$	T_i modifies an entry and then T_j modifies one with a different key.
D_2: $T_i{:}M(\sigma) \rightarrow T_j{:}M(\sigma)$	T_i modifies an entry and then T_j modifies one with the same key.
D_3: $T_i{:}M(\sigma) \rightarrow T_j{:}L(\sigma')$	T_i modifies an entry and then T_j looks up one with a different key.
D_4: $T_i{:}M(\sigma) \rightarrow T_j{:}L(\sigma)$	T_i modifies an entry and then T_j looks up one with the same key.
D_5: $T_i{:}L(\sigma) \rightarrow T_j{:}L(\sigma')$	T_i looks up an entry and then T_j looks up one with a different key.
D_6: $T_i{:}L(\sigma) \rightarrow T_j{:}L(\sigma)$	T_i looks up an entry and then T_j looks up one with the same key.
D_7: $T_i{:}L(\sigma) \rightarrow T_j{:}M(\sigma')$	T_i looks up an entry and then T_j modifies one with a different key.
D_8: $T_i{:}L(\sigma) \rightarrow T_j{:}M(\sigma)$	T_i looks up an entry and then T_j modifies one with the same key.
D_9: $T_i{:}D \rightarrow T_j{:}M(\sigma)$	T_i dumps the directory's entries and then T_j modifies one.
D_{10}: $T_i{:}D \rightarrow T_j{:}L(\sigma)$	T_i dumps the directory's entries, and then T_j looks one up.
D_{11}: $T_i{:}M(\sigma) \rightarrow T_j{:}D$	T_i modifies an entry and then T_j dumps the directory's entries.
D_{12}: $T_i{:}L(\sigma) \rightarrow T_j{:}D$	T_i looks up an entry and then T_j dumps the directory's entries.
D_{13}: $T_i{:}D \rightarrow T_j{:}D$	T_i dumps the directory's entries and then T_j does the same.

By designating the dependencies where neither operation modifies the directory (D_6, D_{10}, D_{12}, and D_{13}) and those that refer to different key strings (D_1, D_3, D_5, and D_7) as insignificant, this means that abstract schedules that keep a directory object consistent must be orderable with respect to $\{<_{D_2 \cup D_4 \cup D_8 \cup D_9 \cup D_{11}}\}$.

Maintaining the same consistency with abstract schedules that use more than one type is possible with a simple extension of this scheme: Spector and Schwarz define the set of consistent abstract schedules that operate on types Y_1, Y_2, ... Y_n as those that are orderable with respect to the union of the proscribed dependency relations for those types. Types accounted for in this union are known as *cooperative types*.

The final part of the paper describes a mechanism that uses this information to control concurrent execution of multiple abstract types. It is a locking scheme, but unlike the more well-known locking schemes [BHG87] it makes use of more sophisticated semantic information than the simple distinction between read and write locks to allow greater concurrency. This is done by the creation of a set of *type-specific locks.*

The first step is determining the types of locks to use. These *lock classes* may include a parameter for run-time data, to further take advantage of semantic information. To lock data during an unfinished transaction that uses the directory data type, there are three categories of locks, which clearly correspond to the three classes of dependency relations:

- A `DirModify`(σ) lock shows that an unfinished transaction has inserted or deleted a directory entry with a key string of σ.

- A `DirLookup`(σ) lock shows that an unfinished transaction has attempted to look up an entry with a key string of σ.

- A `DirDump` lock shows that an unfinished transaction has dumped the directory.

The next step is the building of a *lock compatibility table*, which cross-references locks requested by running transactions with existing held locks in order to determine when locks requests should be granted. The table has a column for each of the lock categories. Instead of three rows, it has five, for the same reason that the directory data type had thirteen possible dependency relations: runtime data is taken into consideration to increase concurrency. To account for the possibility that each pair of operations in $M \rightarrow L$, $M \rightarrow M$, $L \rightarrow M$, and $L \rightarrow L$ dependencies can be invoked using identical or different key strings as parameters, the `DirModify`(σ') and `DirLookup`(σ') rows are added beneath the `DirModify`(σ) and `DirLookup`(σ) rows.

	Lock Held		
Lock requested	**DirModify(σ)**	**DirLookup(σ)**	**DirDump**
DirModify(σ)	No	No	No
DirModify(σ')	OK	OK	No
DirLookup(σ)	No	OK	OK
DirLookup(σ')	OK	OK	OK
DirDump	No	OK	OK

Without the two redundant entries caused by including the `<DirModify(σ'),DirDump>` entry as well as the `<DirModify(σ),DirDump>` entry and the `<DirLookup(σ'),DirDump>` entry as well as the `<DirLookup(σ),DirDump>` entry, the table would have thirteen entries—one for each of the dependency relations.

After an operation defined for a particular object acquires a lock on that object (or some component of it), it holds it until the end of its transaction to prevent the necessity of cascading aborts. The possibility that the two `DirModify` operations (DirDelete and DirInsert) may succeed or fail makes it possible to take advantage of further run-time information to improve concurrency—if either fails, its lock is downgraded to a `DirLookup` lock, which is not as restrictive about how many different operations may proceed while it is in effect.

The paper goes on to construct similar lock compatibility tables for strict FIFO queues and weakly FIFO queues. As we will see, the other papers go on to use even more information to increase potential concurrency, but they build on groundwork laid by Schwarz and Spector: the creation of a table cross-referencing all possible simultaneous operations on an object, taking input parameters into account to minimize the number of situations in which transactions must be held up.

3. Roesler and Burkhard, "Concurrency Control Scheme for Shared Objects: A Peephole Approach Based on Semantics"

Marina Roesler and Walter Burkhard's paper [RB87] describes the concurrency optimization work of Spector[1], Schwarz, and others as dealing with "operation conflict" as opposed to "interaction conflict," which forms the basis of their work. While the others judged conflicts over an object in terms of operations defined for that object and the run-time parameters used with those operations, Roesler and Burkhard incorporate further semantic information into their evaluation of conflicts: the return value of the operations. They also use an alternative to serializability as a criterion for correct cooperation between operations: commutativity, or the demonstration that the relative order of execution of two operations does not affect their final result.

Before each commit in their scheme, an operation is "pseudo-executed" on each object—in other words, the steps are performed to determine the operation's return value, but the object's state is not yet updated. This combination of the potential state transition and return value is known as an *interaction*, identified by an `<operation(parameter),return value>` pair. (The operation can actually have zero or more parameters.) An object's object manager then uses a compatibility function to determine whether the interaction is compatible with the other active transactions using that object. If so, the new operation's transaction is allowed to continue; otherwise, it is blocked. The function uses a compatibility table to compare one active interaction at a time with a potential new interaction.

The paper[2] has three main parts. The first outlines the model of the system and the specification of the objects, the second explains how to construct the table used by the compatibility function, and the third introduces a scheduler that uses the table.

---

[1]Like Skarra and Zdonik's paper "Concurrency Control and Object-Oriented Databases" [SZ89] in "Object-Oriented Concepts, Databases, and Applications," they actually refer to Peter Schwarz's 1984 Carnegie Mellon Ph.D. thesis "Transactions on Typed Objects" and not the Schwarz and Spector paper described earlier. Skarra and Zdonik's description of Schwarz's ideas clearly show that his work with Alfred Spector reflected the concepts and algorithms described in his thesis.

[2]The paper is actually an extended abstract of a longer technical report that Roesler and Burkhard wrote the same year [RB87a]; to shorten it, they omitted proofs and all but the minimum number of tables necessary to demonstrate their construction.

An object’s specification $SPEC_O$ has two parts: the first is a triple representing the possible states, initial state, and possible interactions (I_o) for that object. The other part is the object’s compatibility function, which returns a `yes` or `no` to give the new transaction permission to proceed. It may also return an `x` if the combination won’t happen and its result is therefore moot. For example, a counter’s decrement operation cannot return a value of `false` if the counter’s value is greater than 0, so the pairing of `<value,3>` and `<dec,false>` for a counter object will never happen.

The following table demonstrates in more detail the values returned by a compatibility function for a counter object.

I	`<inc,true>`	`<dec,true>`	`<dec,false>`	`<val,0>. . .`
I’				
`<inc,true>`	yes	yes	no	no
`<dec,true>`	yes	no	x	x
`<dec,false>`	no	x	yes	yes
`<val,0>`	no	x	yes	yes
`<val,1>`	no	no	x	x
…				

The value operation brings up another point: an object may have operations defined with an infinite number of return values, requiring a compatibility table of infinite size. That’s why the table shown is not the one used by the compatibility function, but only the starting point in constructing one. Nearly half of Roesler and Burkhard’s paper is devoted to the algorithm used to create a finite table CT_O from the infinite table C_O. In addition to being finite, table CT_O must also be *abbreviation-complete*—that is, every entry in C_O must have a corresponding entry in CT_O—and it must be *sound*. This final property means that all `yes` entries in CT_O must correspond to `yes` or `x` entries in C_O.

There are more than one CT_0 tables that can be created from a given C_O table. Roesler and Burkhard define CT_0^1 as being more *refined* than CT_0^2 if it has more `yes` entries, allowing greater concurrency. They do point out that it may prove cost-efficient to settle for less than a perfectly refined CT_0 table.

The crux of the paper is the set of rules for deriving CT_0 from an object's specification $SPEC_O$. The first step is the selection of a *granule* for the object. The granule is a special kind of *atom*, an indivisible portion of the object's state that is accessible by the object's interactions. The granule is the atom that will provide the semantic information that helps the compatibility function judge whether two interactions are compatible. It could be a parameter or an interaction's return value. A constant, not being a part of an object's state, cannot be a granule. For the FifoQueue object used as an example in their paper, an item of the queue is designated as the granule.

For the next step, we group the object's possible interactions into four "families" that generalize about the role of the selected granule in the interactions' parameters and/or return values. The FifoQueue example has the following interactions:

`<enq(i),true>`	The value `i` is added to the queue successfully. (There is no `<enq(i),false>` mentioned in the example.)
`<deq,i>`	A dequeue operation returns the value `i`.
`<deq,no>`	A dequeue operation is unsuccessful.
`<printq,q>`	Print the queue.

The following shows the four families into which the interactions are grouped. In the abbreviations, `gr` represents a granule and ~~`gr`~~ represents "not granule."

`<op(gr),gr>`	An operation that has at least one granule among its parameters returns a granule.
`<op(gr),`~~`gr`~~`>`	An operation that has at least one granule among its parameters returns a value that is not a granule. `<enq(i),true>` is in this family.
`<op,gr>`	An operation with no granules for parameters returns a granule. `<deq,i>` is in this family.
`<op,`~~`gr`~~`>`	An operation with no granules for parameters returns a value that is not a granule. `<deq,no>` and `<printq,q>` are both part of this family.

While it may seem that there is only one interaction in the <op(gr),~~gr~~> group, remember that, just as <val,0>, <val,1>, <val,2> etc. were considered different interactions for the counter object, there is a different <enq(i),true> for each value of i. We need to bring the number of interactions (I_O) down to a finite number (designated $\hat{I}_O$) and therefore bring our infinite compatibility table down to a finite size. To accomplish this, the next step partitions each family into *clusters* of equivalence by designating a class for each group of interactions that agree on operation names, granular parameter values, and granular or constant return values. We then assign an abbreviation to each class.

With the FifoQueue's operations (although not necessarily with all objects), each class happens to correspond to an interaction family. From the <op(gr),~~gr~~> family, we can group operations that agree on operation names and constant return values and refer to them using the abbreviation format <op(g),c>. The <enq(g),true> interaction is the only one in this class. In the <op,gr> family we have operations agreeing on operation names and returning a granular value. We assign this group the abbreviation format <op,g>. <deq,g> is the only interaction for this class. The final family with any FifoQueue operations is <op,~~gr~~>, in which operations agree on operation names and constant return values. This class, which uses the abbreviation format <op,c>, has two operations: <deq,no> and <printq,q>. The family not included among these three, <op(gr),gr>, has no interactions to assign to any class.

The next step uses these abbreviations to create the auxiliary tables *TB*$\hat{I}\hat{I}'$ that cross-reference one set of interactions ($\hat{I}$) against another ($\hat{I}'$). Each table compares all the possible combinations of granule equality and inequality for one pair of interaction classes and assigns a predicate p to each combination.

For example, the table at right compares <op(g),c> with <op(g),c>, which for the FifoQueue means that the table shows the possible states of two concurrent <enq(q)true> operations. The symbols p1 and r1 represent the parameter and return value of $\hat{I}$, and p2 and r2 of $\hat{I}'$, and g and g' represent different granule values.

$\hat{I}$ / $\hat{I}'$	<op(g),c>
<op(g),c>	p1 = p2
<op(g'),c>	p1 ≠ p2

A more complex *TB*$\hat{I}\hat{I}'$ example shows the table comparing <op(gr),gr> with <op(gr),~~gr~~>:

$\hat{I}$ / $\hat{I}'$	<op(g),g>	<op(g),g$_1$>
<op(g),c>	p1 = r1 ∧ p1 = p2	p1 ≠ r1 ∧ p1 = p2
<op(g$_1$),c>	p1 = r1 ∧ p1 ≠ p2	p1 ≠ r1 ∧ p1 ≠ p2 ∧ r1 = p2
<op(g'),c>		p1 ≠ r1 ∧ p1 ≠ p2 ∧ r1 ≠ p2

Once the auxiliary tables are created, we replace each predicate with a yes, no, or x, using the infinite compatibility table C_O to consider the equality and inequality relationships listed for the interactions' parameters and return values. If there are no possible states where a given pair of interactions can coexist with the listed equalities, the corresponding cell of the table gets an x. If they can coexist and commute (that is, if the $\hat{I}$ and $\hat{I}'$ interactions can act on the object in either order and still get the same result, with neither of them producing an undecidable result) then the cell gets a yes; otherwise it gets a no. (Roesler and Burkhard assert that, besides the actual selection of the granule atom, this is the only step in the whole process that cannot be automated, and even this step can be automated for certain classes of objects.) This use of commutativity instead of serializability as the correctness criterion, while not their invention, is an important break from the practice of most other concurrency mechanisms [SS84][BHG87].

The first *TB*$\hat{I}\hat{I}'$ table shown above, which compared the two <op(g),c> operations, would get turned into the table at right for the FifoQueue's <enq(g),true> interaction.

$\hat{I}$ / $\hat{I}'$	<enq(g),true>
<enq(g),true>	Y
<enq(g'),true>	N

These values are entered because if two enqueue operations try to add the same value to the queue, the order in which they do it doesn't matter, so they can be allowed to proceed concurrently. If the parameter values are not equal, the new one must be held up until the other finishes.

The final step in the creation of the finite CT_O table is the collapse of each auxiliary table *TB*$\hat{I}\hat{I}'$ into a single cell of the CT_O table according to the following rules: if all the entries are x, it gets collapsed to an x entry. If there are no yes entries and at least one no, it's a no. Otherwise, it's a Y_P where the *P* shows the predicate that allows the interactions to proceed concurrently. If possible, multiple Y_P predicates should be simplified into as few entries as possible.

For the FifoQueue, this produces the following finite, sound, abbreviation-complete $CT_{FifoQueue}$ table (because the table is symmetrical, redundant blank entries are not filled in):

$\hat{I}$ $\hat{I}'$	<enq(g),true>	<deq,g>	<deq,no>	<printq,q>
<enq(g′),true>	$Y_{p1=p2}$			
<deq,g′>	Y	N		
<deq,no>	N	x	Y	
<printq,q>	N	N	Y	Y

To implement the use of this table, the paper describes a mechanism where each transaction has a transaction manager that forwards requests for operations to each object's object manager. The object manager, which is part of the definition of the object, keeps the object's current state, its finite compatibility table CT_O, a list of currently active, compatible interactions known as the *active pool*, and a set of queues for the interactions of transactions that are currently not compatible with those in the active pool. This set of queues is known as the *conflict pool*.

The object manager takes each operation that is neither a commit nor an abort and pseudo-executes it on a copy of the object. Once it gets the result, it uses the compatibility function (which uses the compatibility table CT_O) to check whether that `<operation,return value>` pair is compatible with the interactions currently in the active pool. If so, it adds the operation to the active pool and returns the return value to the transaction that called it. If not, it adds the interaction to a queue for that transaction in the conflict pool.

Upon receipt of an abort instruction, the calling transaction's interactions are removed from both pools. A commit instruction causes the object manager to remove the transaction's interactions from the active pool and to re-evaluate the object's current state for each of those interactions. After either a commit or abort, conflict pool interactions may try again to get into the active pool.

Roesler and Burkhard's most significant contribution to the technique of analyzing semantic information in order to identify and minimize potential concurrency problems is the inclusion of an operation's possible results in the analysis. By going beyond an examination of attributes that reveal an object's current state to consider potential future states, their scheme significantly increases potential operation conflicts to take into account and hence increases the number of identifiable situations in which a newly invoked transaction does not have to be held up to wait for currently executing ones.

4. Chrysanthis, Raghuram, and Ramamritham, "Extracting Concurrency from Objects: A Methodology"

In the quest for new semantic information to take into account in order to justify not blocking a newly invoked transaction, Panos Chrysanthis, S. Raghuram, and Krithi Ramamritham [CRR91] derive additional semantic information from objects by creating graphs based on the objects' abstract specification and analyzing any ordering of the objects' components (for example, the members of a queue, stack, or sorted directory). To increase concurrency even further, they break dependencies down into the categories of *abort dependencies* and *commit dependencies*. As we will see, a commit dependency is weaker than an abort dependency, and therefore allows more concurrency.

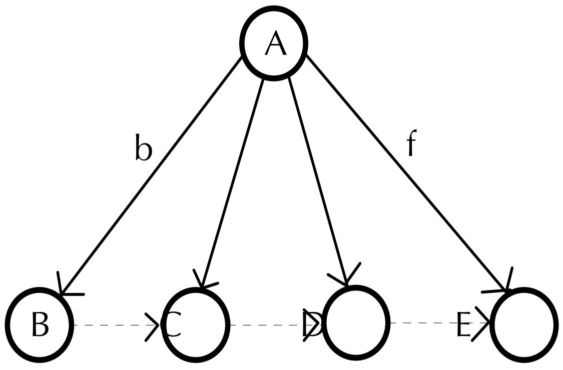
Not including the introduction and summaries, the paper has two key sections. First, they explain the graph used for the object model and the principle of *locality,* which designates which nodes or edges of an object's graph are affected by which operations. They then explain a methodology for developing an object's compatibility table based on the information derived from the graph and the operations defined on that object.

The sample data structure used for an example is a QStack, which combines the properties of a queue and a stack. The operations defined on it are `Enq(e)`, `Deq()`, `Pop()`, `Top()`, `Size()`, `Replace(e1,e2)` (which replaces all `e1` values in the QStack with `e2`) and `XTop()` (which exchanges the first two elements in the back of the QStack).

Before describing the graph model, the paper defines three concepts that characterize an operation's potential interaction with other operations: the concepts of *observer, modifier,* and *modifier-observer*. An observer returns information about an object's state. A modifier modifies the state, but does not return any information about it. A modifier-observer modifies the state and returns something from that state—for example, `Deq()`. An operation's membership in one of these categories can vary, depending on the state of the object being acted upon; for example, a successful `Enq()` is a modifier-observer, because it changes the QStack's state and returns an `ok` to indicate its success, and an unsuccessful `Enq()` is merely an observer, returning a `nok`. An operation is a modifier-observer if there is any possible state of the object in which the operation is a modifier-observer, and it is a modifier if there are no states in which the operation is a modifier-observer, but at least one in which it is a modifier. If neither of these hold true, the operation is an observer.

The idea of "modifying" and "observing" sounds similar to the concepts of "writing" and "reading." The paper's authors maintain that they are broader, more flexible terms, because there is no way to distinguish whether a read or write affects a whole object or just a part of it. The identification of which part of an object is being modified and/or observed by a given operation, and its relation to any part being modified and/or observed by another operation is a crucial step in their evaluation of possible concurrency problems. To perform this identification, they use a graph that makes it possible to keep track of an object's structure and content.

The graph, which can be constructed from an object's abstract specifications, is a rooted graph with vertices representing an object's components, its *composed-of* edges that begin at the root and eventually reach every vertex, and its *ordering* edges, which show the relative ordering of vertices at a particular level.



The graph of an object's instance is dynamic, representing its state at a given time. For example, this graph shows a QStack object that currently has four items. Solid arrows show the composed-of edges and dotted arrows show ordering edges. The `b` and `f` on the graph show that two composed-of edges of the QStack are used as pointers to the back and front of the stack.

Subsets of the graph's vertices known as *localities* are used to identify an operation's effect on an object. The locality L_o of an operation o is the set of vertices that have been affected by o, whether by insertion, deletion, change, or observation, as well as the vertices connected to ordering edges that have been changed or observed by o.

The locality L_o has two important subsets: the *structure locality* L_o^s and the *content locality* L_o^c. L_o^s contains the vertices involved in an operation's structural effect on an object's graph: vertices that are inserted, deleted, connected to edges that are changed or observed, and vertices whose presence is observed by o. L_o^c contains vertices involved in an operation's changes on an object's content. This includes vertices that are added, removed, or whose content is changed or observed. Note the difference in the kind of observed vertices included in the two groups: an operation that checks whether a particular vertex has a sibling is checking for the sibling's presence—a structural observation that would put the sibling in the L_o^s set. An operation that checks for the value stored in a vertex, on the other hand, is observing the vertex's contents, which would include the vertex in the L_o^c set.

The distinction between modification and observation makes it possible to break down the content locality into the *content-observation* locality L_o^{co} and the *content-modification* locality L_o^{cm}. Similarly, the structure locality is divided between the *structure-observation* locality L_o^{so} and the *structure-modification* locality L_o^{sm}. Depending on the state of the object, a given operation can belong to all four groups; for example, a failed `Deq()` operation returns a `nok`, indicating that that QStack is empty, which qualifies it as a structure observation. A successful `Deq()` removes a vertex, which constitutes a modification to the QStack's structure and content. It also returns the vertex's value, making it a content observer as well.

Once these distinctions between subsets of localities are made, the paper asserts that operations limited to an object's structure cannot form dependencies with operations limited to an object's contents. The distinction therefore becomes useful when we build the compatibility tables.

An object may have other objects as components. An object component that is not another object is said to have a *simple data value*. Imagine constructing a hierarchical graph of an object and of all of its subcomponent objects so that the outer vertices of the graph were all simple data values. The collection of an object *ob*'s set of simple data value vertices and those of all of its component object's simple data values is designated as V_{ob}^{simple}. Knowing the vertices in V_{ob}^{simple} enables us to determine whether an operation o on an object ob is a *global* operation. If the locality L_o of that operation contains all the vertexes in V_{ob}^{simple}, it's considered a global operation. (A non-global operation is called just that—calling it a "local" operation would be pretty confusing with all this talk of "localities.")

We now have all the information necessary to put together an object's compatibility table. Other concurrency schemes typically have entries of either `yes`, meaning that the two operations intersecting at that table cell commute and can therefore proceed concurrently, or a `no`, indicating that they conflict. The method used by Chrysanthis et al. uses `ND` for "no dependency" instead of `yes`, and they refine `no` into two categories: `AD` for "abort dependency" and `CD` for "commit dependency."

An abort dependency means that the second of two conflicting transactions can commit only if the first one commits. If transaction *T* begins execution but then its conflicting predecessor *T'* aborts, then *T* must also abort, giving this dependency its name. For example, if *T* reads data written by *T'* and *T'* then aborts, then the data causing the conflict is no longer any good, requiring *T* to abort.

With a commit dependency, the second of two conflicting transactions can commit only after the first one commits or aborts. This ensures serializability, because if transaction *T'* starts before transaction *T*, enforcing commit dependency prevents *T* from committing before its predecessor. A commit dependency is not as strong as an abort dependency, because it can only postpone a commit, as opposed to actually preventing a commit. An abort dependency is actually a stronger form of a commit dependency, because it too prevents the second of two conflicting transactions from committing until the first one has resolved. The difference is the action that an abort dependency takes upon the first transaction's abort.

So, instead of an entry of no in a compatibility table to show that two operations do not conflict, we will see either AD or CD. The refinement into the two possibilities allows more concurrency because of the weaker nature of the commit dependency. For "no dependency," we could enter ND, but those entries are left blank to make the tables easier to read.

Given that an abort dependency is stronger than a commit dependency and a commit dependency is stronger than a "no dependency," the stronger function makes it possible to safely pick the appropriate dependency of the two that could be associated with the modifier-observer combination. This allows us to construct simpler tables, because not as many classes of operations must be compared to create the *template tables*.

We use template tables to create compatibility tables for a given object's operations. The table on the right shows the dependencies created by the four interactions possible between observers and modifiers.

	O	M
O		AD
M	CD	CD

If it's possible to identify whether an object's observation and modification operations affect its content and/or structure, we can use the following three tables, which refine the information shown in the table above.

We use the initials CO, SO, CSO, CM, SM, and CSM to represent content observers, structure observers, etc. up through content/structure modifiers. The columns of each table represent the possible operations currently being performed on an object, and the rows represent potential new operations that may wish to execute concurrently with those.

(O,M)	SM	CM	CSM
SO	AD		AD
CO		AD	AD
CSO	AD	AD	AD

(M,M)	SM	CM	CSM
SM	CD		CD
CM		CD	CD
CSM	CD	CD	CD

(M,O)	SO	CO	CSO
SM	CD		CD
CM		CD	CD
CSM	CD	CD	CD

Once an object's operations have been classified into the CO, SO, CSO, CM, SM and CSM categories, we can use these tables to find the appropriate entries for each cell of the object's preliminary compatibility table. It's a preliminary table because the consideration of additional information about the operations makes it possible to "weaken" some of the cells, allowing further concurrency in the object's operations.

Two important sources of information are operation outcomes and localities. For example, the QStack's Deq and Push operations are both Modifier-Observers. The stronger of the potential dependencies between the two of them is an abort dependency (AD). However, a Push that returns nok is an observer, not a modifier, so we don't always need the stronger dependency. The AD in the compatibility table's cell cross-referencing Push with Deq therefore gets replaced with two (dependency,condition) pairs: (CD,Push_{out}=nok) and (AD,Push_{out}=ok). Similar steps are taken for the rest of the compatibility table, taking operation inputs into consideration as well as outcomes in order to identify situations in which concurrency can be increased.

The last step in refining the compatibility table takes non-global locality into account. For example, in the graph of QStack, f identifies the composed-of edge affected by the structural modifier Deq and b identifies the edge modified by Push. In any QStack having more than one member, f and b will be two separate edges, and concurrent execution of Push and Deq will cause no trouble. So the table cell cross-referencing Push and Deq can now have its second (dependency, condition) pair replaced by two more, giving the cell these three pairs: (CD,Push_{out}=nok), (AD,f=b), and (ND,f≠b). Considering how often f will typically not equal b, this last refinement to the table cell clearly adds a great deal of concurrency to the compatibility table.

Unlike the other papers, [CRR91] presents no implementation of their concurrency scheme. The paper's conclusion points out, with a touch of pride, that the principles upon which they base their work are general enough to make their concurrency scheme viable in a range of implementations. A typical one would include an object manager as part of an object’s implementation to control access to the object by looking up potential dependencies in the compatibility table before allowing new operation executions to proceed.

5. Conclusion

A key tenet of the object-oriented approach is the advantage of dealing with data and operations at a higher level of abstraction. The increased concurrency available by using semantic information to schedule operations on an object-oriented database demonstrates a benefit of this, because it allows the developer to evaluate operations in terms that come closer to their actual functions instead of requiring him or her to break these operations down into their component reads and writes.

The tradeoff of working at a higher level is the same one that drives some people to still work in assembly language: it’s computationally more expensive. Older concurrency algorithms ask "Are the executing operations reading, writing, or both? Does the new one that wants to join them plan to do reads, writes, or both?" Algorithms that take advantage of additional semantic information do so by comparing many other aspects of currently executing operations to possible new ones. The more semantic information that they use, the more comparisons must be done, and the more CPU cycles are required to determine whether the new transaction really needs to be held up or not. This is compounded by the need to check much of the most valuable semantic information, such as potential transaction results or passed parameters, at run-time whenever a transaction is invoked.

However, CPU cycles get cheaper every year, and in this case they are being spent on a good cause: increased efficiency in the overall system. When a scheduling algorithm considers whether to allow transaction *T* to proceed concurrently with *T’*, a quick glance at *T* and *T’* may show that their concurrent execution might cause problems, while a closer examination may show that there won’t in fact be any problem. In this case,*T* can be allowed to continue and the overall system proceeds more quickly despite the extra cycles used to analyze the relationship between *T* and *T’*.

Another problem with object-oriented systems will require more than cheaper CPU cycles to ameliorate. The efficiency of the concurrency schemes described in this paper depend heavily on the quality of the database’s design. Any of these concurrency schemes, when implemented with a badly designed set of objects and operations, will result in a mess. While proponents love to cite the ease of use and re-use of object-oriented systems, the initial design and creation of these systems are not easy to implement well. There is currently no clear, relatively simple series of steps comparable to the normalization process used in designing relational systems [U88] that start with a list of data attributes and their relationships and turns this list into a reasonably efficient object-oriented database schema. Improved concurrency is a good example of the benefits of object-oriented systems that will be available when further progress is made on design issues.

Questions

1. *Schema evolution* [N89], or modification to class definitions, is an important issue in the maintenance of object-oriented systems. Imagine a Queue object with Enqueue, Dequeue, and Peek operations defined. All of the compile-time compatibility table operations have been performed, with their results stored where they can be used to maximize concurrency. Now, we add a LookUp(i) operation to check for the existence of a queue member with the value i and return a `Yes` or `No`. What would be necessary to take this new operation into account with systems using each of the three concurrency schemes described in this paper?

2. Queues provide a typical example of the value of semantic information in controlling concurrency, because while simpler concurrency schemes view concurrent enqueue and dequeue operations as concurrent writes to the same data structure and therefore prevent them, a more sophisticated scheme could see that different queue items are being affected and that the enqueue and dequeue operations would not get in each other's way. For a queue that holds less than two items, interleaving of enqueue and dequeue instructions could pose a problem, and should therefore not be allowed to proceed concurrently. If the number of items currently in a queue is private information within the queue class of objects and a concurrency control scheme uses that value as semantic information to decide whether a given pair of enqueue and dequeue operations should be allowed to proceed concurrently, does this violate the object-oriented principle of encapsulation?

3. Which concurrency scheme has the highest run-time cost? Which has the lowest?

4. As relational database systems try to compete in the specialized markets currently using object-oriented systems, one new feature incorporated into some relational systems is user-defined operations [S92]. These systems allow the use of application-specific operations from within SQL statements. What circumstances would allow these systems to take advantage of the increased concurrency availability in object-oriented database systems?

Answers

1. Using Schwarz and Spector's scheme, you would first add a dependency relation to the list that compared LookUp with each existing operation, and an extra one comparing LookUp(i') with Enqueue(i) (that is, in addition to the dependency relation of LookUp and Enqueue acting with the same parameter values, you would have another where they used different parameter values). A new lock class must be defined, and then a new row and column must be calculated for the lock compatibility table.

For a system using Roesler and Burkhard's scheme, the two possible results `<LookUp(i),Yes>` and `<LookUp(i),No>` must be assigned to an interaction family. They both belong in the family `<op(gr),`~~`gr`~~`>`, as does `<Enqueue(i),Yes>`, and the same auxiliary table can be used to determine the predicates to consider. So the main work here is the next step: figuring out which predicates become `yes`, `no`, or `x` when comparing `<LookUp(i),Yes>` and `<LookUp(i),No>` with other interactions. Like the steps before the conversion of predicates into `yes`, `no`, and `x`, the final step—the collapse of the LookUp operation's *TBÎÎ'* tables into individual entries for each of the two new interactions' rows and columns of the Queue's finite *CT* table—can be automated.

This ease of adding the new operation with Boesler and Burkhard's scheme is contingent on the simple, obvious choice of granule for the Queue: the queue item. For a more complex class, the choice of granule would strongly affect the amount of trouble necessary to incorporate a new operation into an object's concurrency control.

Using the scheme of Chrysanthis et al., the LookUp operation would be classified as a content observer of the queue, so adding the new row and column for it to the preliminary compatibility table would be simple using the template tables. The weakening of the new entries, or the consideration of operation outcomes and localities to allow for the possibility of more concurrency, must then be done manually to each cell in the new row and column.

2. Encapsulation is maintained, because in the first two papers and any reasonable implementation of the third [CRR91], concurrency implementation is part of the definition of the class, not an outside process reading the object's private variables.

3. Roesler and Burkhard's scheme, with its pseudo-execution of operations and conflict pool of blocked transaction queues, has more run-time work to do than either of the other two. While all the schemes consider an operation's run-time parameters, Chrysanthis et al.'s consideration of locality (a property that, given the different results of concurrently executing Enqueue and Dequeue operations on an empty queue versus a queue with four items, must be constantly re-evaluated at runtime) makes Spector and Schwarz's runtime evaluation of just the operation parameters the least costly at runtime—and, not coincidentally, the least capable of maximizing concurrency.

4. The object-oriented principle of encapsulation makes it possible to determine all of the operations that may be legally performed on an object. A relational system that allows the cross-referencing of possible operation characteristics would need a catalog of the operations available and preferably some mechanism to ensure that no operations outside of the "registered" ones can be performed on the data.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[CRR91] Panos Chrysanthis, S. Raghuram, and Krithi Ramamritham. Extracting Concurrency from Objects: A Methodology. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. SIGMOD Record Volume 20, Issue 2 June 1991 pages 108 - 117.

[N89] Oscar Nierstrasz. A Survey of Object-Oriented Concepts. In *Object-Oriented Concepts, Databases, and Applications.* Won Kim and Frederick H. Lochovsky, editors. ACM Press, New York, September 1989.

[RB87] M. Roesler and W. A. Burkhard. Concurrency Control Scheme for Shared Objects: A Peephole Approach Based on Semantics. In *Proceedings of the 7th International IEEE Conference on Distributed Sys tems*. September 1987 pages 224-231.

[RB87a] M. Roesler and W.A. Burkhard. Semantics-Based Concurrency Control Scheme for Shared Objects: A Peephole Approach. Technical Report C-091, CS&E Dept., UCSD, Jan 1987.

[S92] Dennis E. Shasha. *Database Tuning: A Principled Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[SS84] Peter M. Spector and Alfred Z. Schwarz. Synchronizing Shared Abstract Types. In *ACM Transactions on Computer Systems* Vol 2, No. 3, August 1984 pages 223-250.

[SZ89] Andrea H. Skarra and Stanley B. Zdonik. Concurrency Control and Object-Oriented Databases. In *Object-Oriented Concepts, Databases, and Applications.* Won Kim and Frederick H. Lochovsky, editors. ACM Press, New York, September 1989.

[U88] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems Volume I: Classical Database Systems.* Computer Science Press, Rockville, Maryland 1988.