SAGAS

Hector Garica-Molina Kenneth Salem

January 7, 1987

ABSTRACT

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions. To alleviate these problems we propose the notion of a saga. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution. Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly. We analyze the various implementation issues related to sagas, including how they can be run on an existing system that does not directly support them. We also discuss techniques for database and LLT design that make it feasible to break up LLTs into sagas.

INTRODUCTION

As its name indicates, a *long lived transaction* is a transaction whose execution, even without interference from other transactions, takes a substantial amount of time, possibly on the order of hours or days. A long lived transaction, or LLT, has a long duration compared to the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a].

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a]. To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction. As a consequence, other transactions wishing to access the LLT's objects are delayed for a substantial amount of time.

Furthermore, the transaction abort rate can also be increased by LLTs. As discussed in [Gray81b], the frequency of deadlock is very sensitive to the "size" of transactions, that is, to how many objects transactions access. (In the analysis of [Gray81b] the deadlock frequency grows with the fourth power of the transaction size.) Hence, since LLTs access many objects, they may cause many deadlocks, and correspondingly, many abortions. From the point of view of system crashes, LLTs have a higher probability of encountering a failure (because of their duration), and are thus more likely to encounter yet more delays and more likely to be aborted themselves.

In general there is no solution that eliminates the problems of LLTs. Even if we use a mechanism different from locking to ensure atomicity of the LLTs, the long delays and/or the high abort rate will remain: No matter how the mechanism operates, a transaction that needs to access the objects that were accessed by a LLT cannot commit until the LLT commits.

However, for *specific applications* it may be possible to alleviate the problems by relaxing the requirement that an LLT be executed as an atomic action. In other words, without sacrificing the consistency of the database, it may be possible for certain LLTs to release their resources before they complete, thus permitting other waiting transactions to proceed.

To illustrate this idea, consider an airline reservation application. The database (or actually a collection of databases from different airlines) contains reservations for flights, and a transaction T wishes to make a number of reservations. For this discussion, let us assume that T is a LLT (say it pauses for customer input after each reservation). In this application it may not be necessary for T to hold on to all of its resources until it completes. For instance, after T reserves a seat on flight F_1, it could immediately allow other transactions to reserve seats on the same flight. In other words, we can view T as a collection of "sub-transactions" $T_1, T_2, \ldots, T_n$ that reserve the individual seats.

However, we do not wish to submit T to the database management system (DBMS) simply as a collection of independent transactions because we still want T to be a unit that is either successfully completed or not done at all. We would not be satisfied with a DBMS that would allow T to reserve three out of five seats and then

(due to a crash) do nothing more. On the other hand, we would be satisfied with a DBMS that guaranteed that T would make all of its reservations, or would cancel any reservations made if T had to be suspended.

This example shows that a control mechanism that is less rigid than the conventional atomic-transaction ones but still offers some guarantees regarding the execution of the components of an LLT would be useful. In this paper we will present such a mechanism.

Let us use the term *saga* to refer to a LLT that can be broken up into a collection of sub-transactions that can be interleaved in any way with other transactions. Each sub-transaction in this case is a real transaction in the sense that it preserves database consistency. However, unlike other transactions, the transactions in a saga are related to each other and should be executed as a (non-atomic) unit: any partial executions of the saga are undesirable, and if they occur, must be compensated for.

To amend partial executions, each saga transaction T_i should be provided with a compensating transaction C_i. The compensating transaction undoes, from a semantic point of view, any of the actions performed by T_i, but does not necessarily return the database to the state that existed when the execution of T_i began. In our airline example, if T_i reserves a seat on a flight, then C_i can cancel the reservation (say by subtracting one from the number of reservations and performing some other checks). But C_i cannot simply store in the database the number of seats that existed when T_i ran because other transactions could have run between the time T_i reserved the seat and C_i canceled the reservation, and could have changed the number of reservations for this flight.

Once compensating transactions $C_1, C_2, \ldots C_{n-1}$ are defined for saga $T_1, T_2, \ldots, T_n$, then the system can make the following guarantee. Either the sequence

$$T_1, T_2, \ldots, T_n$$

(which is the preferable one) or the sequence

$$T_1, T_2, \ldots, T_j, C_j, C_2, C_1$$

for some $0 <= j < n$ will be executed.

Sagas appear to be a relatively common type of LLT. They occur when a LLT

consists of a sequence of relatively independent steps, where each step does not have to observe the *same* consistent database state. For instance, in a bank it is common to perform a fixed operation (*e.g.*, compute interest) on all accounts, and there is very little interaction between the computations for one account and the next. In an office information system, it is also common to have LLTs with independent steps that can be interleaved with those of other transactions. For example, receiving a purchase order involves entering the information into the database, updating the inventory, notifying accounting, printing a shipping order, and so on. Such office LLTs mimic real procedures and hence can cope with interleaved transactions. In reality, one does not physically lock the warehouse until a purchase order is fully processed. Thus there is no need for the computerized procedures to lock out the inventory database until they complete.

Once again, the bank and office LLTs we have presented are not just collections of normal transactions, they are sagas. There is an application "constraint" (not representable by the database consistency constraints) that the steps of these activities should not be left unfinished. The applications demand that all accounts be processed or that the purchase order is fully processed. If the purchase order is not successfully completed, then the records must be straightened (*e.g.*, inventory should not reflect the departure of the item). In the bank example, it may always be possible to move forward and finish the LLT. In this case, it may not be necessary to ever compensate for an unfinished LLT.

Note that the notion of a saga is related to that of a nested transaction [Mossa, Lync83a]. However, there are two important differences:

(a) A saga only permits two levels of nesting: the top level saga and simple transactions, and
(b) At the outer level full atomicity is not provided. That is, sagas may view the partial results of other sagas.

Sagas can also be viewed as special types of transactions running under the mechanisms described in [Garc83a, Lync83a]. The restrictions we have placed on the more general mechanisms make it much simpler to implement (and understand) sagas, in consequence making it more likely that they be used in practice.

Two ingredients are necessary to make the ideas we have presented feasible: a DBMS that supports sagas, and LLTs that are broken into sequences of transactions. In the rest of this paper we study these ingredients in more detail. In Sections 2 through 7 we study the implementation of a saga processing mechanism. We start by discussing how an application programmer can define sagas, and then how the

system can support them. We initially assume that compensating transactions can only encounter system failures. Later on, in Section 6, we study the effects of other failures (*e.g.* program bugs) in compensating transactions.

In Sections 8 and 9 we address the design of LLTs. We first show that our model of sequential transaction execution for a saga can be generalized to include parallel transaction execution and hence a wider range of LLTs. Then we discuss some strategies that an application programmer may follow in order to write LLTs that are indeed sagas and can take advantage of our proposed mechanism.

2. USER FACILITIES

From the point of view of an application programmer, a mechanism is required for informing the system of the beginning and end of a saga, the beginning and end of each transaction, and the compensating transactions. This mechanism could be similar to the one used in conventional systems to manage transactions [Gray78a].

In particular, when an application program wishes to initiate a saga it issues a *begin-saga* command to the system. This is followed by a series of *begin-transaction*, *end-transaction* commands that indicate the boundaries of each transaction. In between these commands the application program would issue conventional database access commands. From within a transaction, the program can optionally start a userinitiated abort by issuing an *abort-transaction* command. This terminates the current transaction, but not the saga. Similarly, there is an *abort-saga* command to abort first the currently executing transaction and second the entire saga (by running compensating transactions). Finally, there is an *end-saga* command to commit the currently executing transaction (if any) and to complete the saga.

Most of these commands will include various parameters. The *begin-saga* command can return a saga identifier to the program. This identifier can then be passed to the system on subsequent calls made by the saga. An *abort-transaction* command will include as a parameter the address where saga execution is to continue after the abortion. Each *end-transaction* call includes the identification of the compensating transaction that must be executed in case the currently ending transaction must be rolled back. The identification includes the name and entry point of the compensating program, plus any parameters that the compensating transaction may need. (We assume that each compensating program includes its own *begin-transaction* and *end-transaction* calls. *Abort-transaction* and *abort-saga* commands are not allowed within a compensating transaction.) Finally, the *abort-saga* command may include as a parameter a *save-point identifier*, as described below.

Note that it is possible to have each transaction store in the database the parameters that its compensating transaction may need in the future. In this case, the parameters do not have to be passed by the system; they can be read by the compensating transaction when it starts. Also note that if an *end-saga* command ends both the last transaction and the saga, there is no need to have a compensating transaction for the last transaction. If instead a separate *end-transaction* is used, then it will have to include the identification of a compensating transaction.

In some cases it may be desirable to let the application programmer indicate through the *save-point* command where *saga check points* should be taken. This command can be issued between transactions. It forces the system to save the state of the running application program and returns a *save-point identifier* for future reference. The save points could then be useful in reducing the amount of work after a saga failure or a system crash: instead of compensating for all of the outstanding transactions, the system could compensate for transactions executed since the last save point, and then restart the saga.

Of course, this means that we can now have executions of the type T_1, T_2, C_2, T_2, T_3, T_4, T_5, C_5, C_4, T_4, T_5, T_6. (After successfully executing T_2 the first time, the system crashed. A *save-point* had been taken after T_1, but to restart here, the system first undoes T_2 by running C_2. Then the saga can be restarted and T_2 reexecuted. A second failure occurred after the execution of T_5.) This means that our definition of valid execution sequences given above must be modified to include such sequences. If these partial recovery sequences are not valid, then the system should either *not* take savepoints, or it should take them automatically at the beginning (or end) of every transaction.

The model we have described up to now is the quite general, but in some cases it may be easier to have a more restrictive one. We will discuss such a restrictive model later on in Section 5.

3. SAVING CODE RELIABLY

In a conventional transaction processing system, application code is not needed to restore the database to a consistent state after a crash. If a failure destroys the code of a running transaction, the system logs contains enough information to undo the effects of the transaction. In a saga processing system, the situation is different. To complete a running saga after a crash it is necessary to either complete the missing transactions or to run compensating transactions to abort the saga. In either case it is essential to have the required application code.

There are various possible solutions to this problem. One is to handle application code as system code is handled in conventional systems. Note that even though a conventional DBMS need not save *application* code reliably, it must save system code. That is, a conventional DBMS cannot restart if failure destroys the code required to run the system. Thus, conventional systems have manual or automatic procedures, outside the DBMS itself, for updating and storing backup copies of the system.

In a saga processing system we could then require that application code for sagas be defined and updated in the same fashion. Each new version of a program created would be stored in the current system area, as well as in one or more backup areas. Since the updates would not be under the control of the DBMS, they would not be atomic operations and would probably require manual intervention in case a crash occurs during the update. When a saga starts running, it would assume that all its transactions and compensating transactions have been predefined, and it would simply make the appropriate calls.

Such an approach may be acceptable if sagas are written by trusted application programmers and not updated frequently. If this is not the case, it may be best to handle saga code as part of the database. If saga code is simply stored as one or more database objects, then its recovery would be automatic. The only drawback is that the DBMS must be able to handle large objects, i.e., the code. Some systems would not be able to do this, because their data model does not permit large "unstructured" objects, the buffer manager cannot manage objects that span more than one buffer, or some other reason.

If the DBMS can manage code, then reliable code storage for sagas becomes quite simple. The first transaction of the saga, T_1, enters into the database *all* further transactions (compensating or not) that may be needed in the future. When T_1 commits, the rest of the saga is ready to start. The compensating transaction for T_1, C_1 would simply remove these objects from the database. It is also possible to define transactions incrementally. For example, a compensating transaction C_i need not be entered into the database until its corresponding transaction T_i; is ready to commit. This approach is slightly more complicated but saves unnecessary database operations.

4. BACKWARD RECOVERY

When a failure interrupts a saga, there are two choices: compensate for the executed transactions, *backward recovery*, or execute the missing transactions, *forward recovery*. (Of course, forward recovery may not be an option in all situations.) For

backward recovery the system needs *compensating transactions*, for *forward recovery* it needs *save-points*. In this section we will describe how pure *backward recovery* can be implemented, the next will discuss mixed backward/forward and pure forward recovery.

Within the DBMS, a *saga execution component* (SEC) manages sagas. This component calls on the conventional *transaction execution component* (TEC), which manages the execution of the individual transactions. The operation of the SEC is similar to that of the TEC: the SEC executes a series of transactions as a unit, while the TEC executes a series of actions as an (atomic) unit. Both components require a log to record the activities of sagas and transactions. As a matter of fact, it is convenient to merge both logs into a single one, and we will assume that this is the case here. We will also assume that the log is duplexed for reliability. Note that the SEC needs no concurrency control because the transactions it controls can be interleaved with other transactions.

All saga commands and database actions are channeled through the SEC. Each saga command (*e.g.*, *begin-saga*) is recorded in the log before any action is taken. Any parameters contained in the commands (*e.g.*, the compensating transaction identification in an *end-transaction* command) are also recorded in the log. The *begin-transaction* and *end-transaction* commands, as well as all database actions, are forwarded to the TEC, which handles them in a conventional way [Gray78a].

When the SEC receives an *abort-saga* command it initiates *backward recovery*. To illustrate, let us consider a saga that has executed transactions T_1 and T_2, and that halfway through the execution of T_3 issues an *abort-saga* command to the SEC. The SEC records the command in the log (to protect against a crash during roll back) and then instructs the TEC to abort the current transaction T_3. This transaction is rolled back using conventional techniques, *e.g.*, by storing the "before" values (found in the log) back into the database.

Next the SEC consults the log and orders the execution of compensating transactions C_2 and C_1. If the parameters for these transactions are in the log, they are extracted and passed in the call. The two transactions are executed just like other transactions, and of course, the information as to when they begin and commit is recorded in the log by the TEC. (If there is a crash during this time, the system will then be able to know what work remains to be done.) When C_1 commits, the saga terminates. An entry is made in the log, similar to the one created by the *end-saga* command.

The log is also used to recover from crashes. After a crash, the TEC is first invoked to clean up pending transactions. Once all transactions are either aborted or committed, the SEC evaluates the status of each saga. If a saga has corresponding

begin-saga and *end-saga* entries in the log, then the saga completed and no further action is necessary. If there is a missing *end-saga* entry, then the saga is aborted. By scanning the log the SEC discovers the identity of the last successfully executed and uncompensated transaction. Compensating transactions are run for this transaction and all preceeding ones.

5. FORWARD RECOVERY

For forward recovery, the SEC requires a reliable copy of the code for all missing transactions plus a *save-point*. The save point to be used may be specified by the application or by the system, depending on which aborted the saga. (Recall that a *save-point identifier* can be included as a parameter to the *abort-saga* command.) In the case of a system crash, the recovery component can specify the most recent save point for each active saga.

To illustrate the operation of the SEC in this case, consider a saga that executes transactions T_1, T_2, a *save-point* command, and transaction T_3. Then during the execution of transaction T_4 the system crashes. Upon recovery, the system must first perform a *backward recovery* to the *save-point* (aborting T_4 and running C_3). After ensuring that the code for running T_3, T_4, ... is available, the SEC records in the log it decision to restart and restarts the saga. We call this backward/forward recovery.

As mentioned in Section 2, if *save-points* are automatically taken at the beginning of every transaction, then pure forward recovery is feasible. If we in addition prohibit the use of *abort-saga* commands, then it becomes unnecessary to ever perform backward recovery.[1] (*Abort-transaction* commands would still be acceptable.) This has the advantage of eliminating the need for compensating transactions, which may be difficult to write in some applications (see Section 9).

In this case the SEC becomes a simple "persistent" transaction executor, similar to persistent message transmission mechanisms [Hamm80a]. After every crash, for every active saga, the SEC instructs the TEC to abort the last executing transaction, and then restarts the saga at the point where this transaction had started.

We can simplify this further if we simply view a saga as a file containing a sequence of calls to individual transaction programs. Here there is no need for explicit begin or end saga nor begin or end transaction commands. The saga begins with the first call in the file and ends with the last one. Furthermore, each call is a transaction. The state of a running saga is simply the number of the transaction that is executing.

[1] In this case we must also assume that every sub-transaction in the saga will eventually succeed if it is retried enough times.

This means that the system can take *save-points* after each transaction with very little cost.

Such pure forward recovery methods would be useful for simple LLTs that always succeed. The LLT that computes interest payments for back accounts may be an example of such a LLT. The interest computation on an individual account may fail (through an *abort-transaction* command), but the rest of the computations would proceed unaffected.

Using operating systems terminology, the transaction file model described above could be called a simple EXEC or SCRIPT. The idea of a persistent SCRIPT would also be useful in an operating system to ensure that a collection of commands were successfully executed (assuming that each command executed as a transaction). For example, a typical text processing and printing job consists of several steps (*e.g.*, in UNIX, equation processing, troffing, printing). Each step produces one or more files that are used by the following steps. A persistent SCRIPT would allow a user to start a long text processing job and go home, confident that the system would complete it.

6. OTHER ERRORS

Up to this point we have assumed that the user-provided code in compensating transactions does not have bugs. But what happens if a compensating transaction cannot be successfully completed due to errors (*e.g.*, it tries to read a file that does not exist, or there is a bug in the code)? The transaction could be aborted, but if it were run again it would probably encounter the same error. In this case, the system is stuck: it cannot abort the transaction nor can it complete it. A similar situation occurs if in a pure forward scenario a transaction has an error.

One possible solution is to make use of software fault tolerant techniques along the lines of recovery blocks [Ande81a, Horn74a]. A recovery block is an alternate or secondary block of code that is provided in case a failure is detected in the primary block. If a failure is detected the system is reset to its pre-primary state and the secondary block is executed. The secondary block is designed to achieve the same end as the primary using a different algorithm or technique, hopefully avoiding the primary's failure.

The recovery block idea translates very easily into the framework of sagas. Transactions are natural program blocks, and rollback capability for failed transactions is provided by the TEC. The saga application can control recovery block execution. After it aborts a transaction (or is notified that its transaction has been aborted),

the application either aborts the saga, tries an alternative transaction, or retries the primary. Note that compensating transactions can be given alternates as well to make aborting sagas more reliable.

The other possible solution to this problem is manual intervention. The erroneous transaction is first aborted. Then it is given to an application programmer who, given a description of the error, can correct it. The SEC (or the application) then reruns the transaction and continues processing the saga.

Fortunately, while the transaction is being manually repaired the saga does *not* hold any database resources (i.e., locks). Hence, the fact that an already long saga will take even longer will not significantly affect performance of other transactions.

Relying on manual intervention is definitely not an elegant solution, but it is a practical one. The remaining alternative is to run the saga as a long transaction. When this LLT encounters an error it will be aborted in its entirety, potentially wasting much more effort. Furthermore, the bug will still have to be corrected manually and the LLT resubmitted. The only advantage is that during the repair, the LLT will be unknown to the system. In the case of a saga, saga will continue to be pending in the system until the repaired transaction is installed.

7. IMPLEMENTING SAGAS ON TOP OF AN EXISTING DBMS

In our discussion of saga management we have assumed that the SEC is part of the DBMS and has direct access to the log. However, in some cases it may be desirable to run sagas on an existing DBMS that does not directly support them. This is possible as long as the database can store large unstructured objects (i.e., code and *save-points*). However, it involves giving the application programmer more responsibilities and possibly hurting performance.

There are basically two things to do. First, the saga commands embedded in the application code become subroutine calls (as opposed to system calls). (The subroutines are loaded together with the application code.) Each subroutine stores within the database all the information that the SEC would have stored in the log. For example, the *begin-saga* subroutine would enter an identification for the saga in a database table of active sagas. The *save-point* subroutine would cause the application to save its state (or a key portion of its state) in a similar database table. Similarly, the *end-transaction* subroutine enters into some other table(s), the identification of the ending transaction and its compensating transaction before executing an *end-transaction* system call (to be processed by the TEC).

The commands to store saga information (except *save-point*) in the database must always be performed within a transaction, else the information may be lost in a crash. Thus, the saga subroutines must keep track of whether the saga is currently executing a transaction or not. This can easily be achieved if the begin-transaction subroutine sets a flag that is reset by the *end-transaction* one. All database storage actions would be disallowed if the flag is not set. Note that the subroutine approach only works if the application code never makes system calls on its own. For instance, if a transaction is terminated by an *end-transaction* system call (and not a subroutine call), then the compensating information will not be recorded and the transaction flag will not be reset.

Second, a special process must exist to implement the rest of the SEC functions. This process, the *saga daemon* (SD) would always be active. It would be restarted after a crash by the operating system. After a crash it would scan the saga tables to discover the status of pending sagas. This scan would be performed by submitting a database transaction. The TEC will only execute this transaction after transaction recovery is complete; hence the SD will read consistent data. Once the SD knows the status of the pending sagas, it issues the necessary compensating or normal transactions, just as the SEC would have after recovery. Care must be taken not to interfere with sagas that started right after the crash, but before the SD submitted its database query.

After the TEC aborts a transaction (*e.g.*, because of a deadlock or a user initiated abort), it may simply kill the process that initiated the transaction. In a conventional system this may be fine, but with sagas this leaves the saga unfinished. If the TEC cannot signal the SD when this occurs, then the SD will have to periodically scan the saga table searching for such a situation. If found, the corrective action is immediately taken.

A running saga can also directly request services from the SD. For instance, to perform an *abort-saga*, the *abort-saga* subroutine sends the request to the SD and then (if necessary) executes an *abort-transaction*.

8. PARALLEL SAGAS

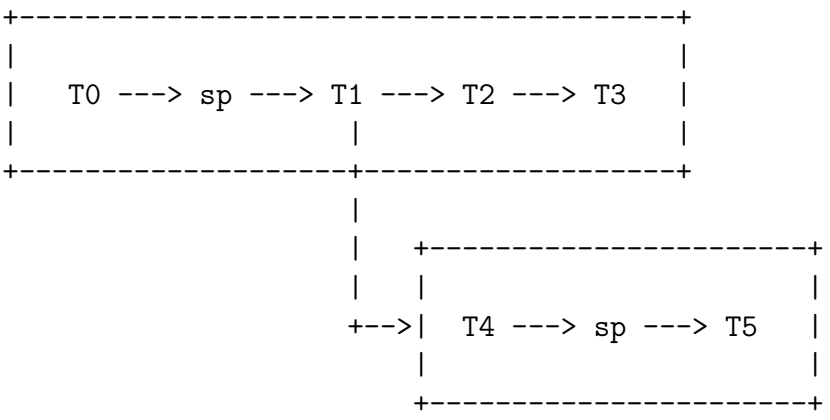
Our model for sequential transaction execution within a saga can be extended to include parallel transactions. This could be useful in an application where the transactions of a saga are naturally executed concurrently. For example, when processing a purchase order, it may be best to generate the shipping order and update accounts receivable at the same time.

We will assume that a saga process (the parent) can create new processes (children) with which it will run in parallel, with a request similar to a fork request in UNIX. The system may also provide a join capability to combine processes within a saga.

Backward crash recovery for parallel sagas is similar to that for sequential sagas. Within each process of the parallel saga, transactions are compensated for (or undone) in reverse order just as with sequential sagas. In addition, all compensations in a child process must occur before any compensations for transactions in the parent that were executed before the child was created (forked). (Note that only transaction execution order *within a process* and fork and join information constrain the order of compensation. If T_1 and T_2 have executed in parallel processes and T_2 has read data written by T_1, compensating for T_1 does not force us to compensate for T_2 first.)

Unlike backward crash recovery, *backward recovery* from a saga failure is more complicated with parallel sagas because the saga may consist of several processes, all of which must be terminated. For this, it is convenient to route all process fork and join operations through the SEC so it can keep track of the process structure of the saga. When one of the saga processes requests an *abort-saga*, the SEC kills all processes involved in the saga. It then aborts all pending transactions and compensates all committed ones.

Forward recovery is even more complicated due to the possibility of "inconsistent" *save-points*. To illustrate, consider the saga of Figure 8.1. Each box represents a process; within each box is the sequence of transactions and *save-points* (sp) executed by the process. The lower process was forked after T_1 committed. Suppose that T_3 and T_5 are the currently executing transactions and that *save-points* were executed before T_1 and T_5.



At this point the system fails. The top process will have to be restarted before T_1. Therefore, the *save-point* made by the second process is not useful. It depends on the execution of T_1 which is being compensated for.

This problem is known as cascading roll backs. It problem has been analyzed in a scenario where processes communicate via messages [Rand78a]. There it is possible to analyze *save-point* dependencies to arrive at a consistent set of *save-points* (if it exists). The consistent set can then be used to restart the processes. With parallel sagas, the situation is even simpler since *save-point* dependencies arise only through forks and joins, and transaction and *save-point* order within a process.

To arrive at a consistent set of *save-points*, the SEC must again be informed of process forking and joining. The information must be stored on the log and analyzed at recovery time. The SEC chooses the latest *save-point* within each process of the saga such that no earlier transaction has been compensated for. (A transaction is earlier than a *save-point* if it would have to be compensated for after a transaction that had executed in place of that *save-point*). If there is no such *save-point* in a process, that entire process must be rolled back. For those processes with *save-points*, the necessary backward recoveries can be conducted and the processes restarted.

9. DESIGNING SAGAS

The saga processing mechanisms we have described will only be of use if application programmers write their LLTs as sagas. Thus the following questions immediately arise: How can a programmer know if a given LLT can be safely broken up into a sequence of transactions? How does the programmer select the break points? How difficult is it to write compensating transactions? In this section we will address some of these issues.

To identify potential sub-transactions within a LLT, one must search for natural divisions of the work being performed. In many cases, the LLT models a series of real world actions, and each of these actions is a candidate for a saga transaction. For example, when a university student graduates, several actions must be performed before his or her diploma can be issued: the library must check that no book are out, the controller must check that all housing bills and tuition bills are checked; the students new address must be recorded; and so on. Clearly, each of these real world actions can be modeled by a transaction.

In other cases, it is the database itself that is naturally partitioned into relatively independent components, and the actions on each component can be grouped into a saga transaction. For example, consider the source code for a large operating system. Usually the operating system and its programs can be divided into components like the scheduler, the memory manager, the interrupt handlers, etc. A LLT to add a tracing facility to the operating system can be broken up so that each transaction adds the tracing code to one of the components. Similarly, if the data on employees can be split by plant location, then a LLT to give a cost-of-living raise to all employees can be broken up by plant location.

Designing compensating transactions for LLTs is a difficult problem *in general*. (For instance, if a transaction fires a missile, it may not be possible to undo this action.) However, for many practical applications it may be as simple (or difficult) as writing the transactions themselves. In fact, Gray notes in [Gray81a] that, transactions often have corresponding compensating transactions within the application transaction set. This is especially true when the transaction models a real world action that can be undone, like reserving a rental car or issuing a shipping order. In such cases, writing either a compensating or a normal transaction is very similar: the programmer must write code that performs the action and preserves the database consistency constraints.

It may even be possible to compensate for actions that are harder to undo, like sending a letter or printing a check. For example, to compensate for the letter, send a second letter explaining the problem. To compensate for the check, send a stoppayment message to the bank. Of course, it would be desirable not to have to compensate for such actions. However, the price of running LLTs as regular transactions may be so high that one is forced to write sagas and their compensating transactions.

Also recall that pure forward recovery does not require compensating transactions (see Section 5). So if compensating transactions are hard to write, then one has the choice of tailoring the application so that LLTs do not have user initiated aborts. Without these aborts, pure forward recovery is feasible and compensation is never needed.

As has become clear from our discussion, the structure of the database plays an important role in the design of sagas. Thus, it is best not to study each LLT in isolation, but to design the entire database with LLTs and sagas in mind. That is, if the database can be laid out into a set of loosely-coupled components (with few and simple inter-component consistency constraints), then it is likely that the LLT will naturally break up into sub-transactions that can be interleaved.

Another technique that could be useful for converting LLTs into sagas involves storing the temporary data of an LLT in the database itself. To illustrate, consider a LLT L with three sub-transactions T_1, T_2, and T_3. In T_1, L performs some actions and then withdraws a certain amount of money from an account stored in the database. This amount is stored in a temporary, local variable until during T_3 the funds are placed in some other account(s). After T_1 completes, the database is left in an inconsistent state because some money is "missing," i.e., it cannot be found in the database. Therefore, L cannot be run as a saga. If it were, a transaction that needed to see all the money (say an audit transaction) could run sometime between T_1 and T_3 and would not find all the funds. If L is run as a regular transaction, then the audit is delayed until L completes. This guarantees consistency but hurts performance.

However, if instead of storing the missing money in local storage L stores it in the database, then the database would be consistent, and other transactions could be interleaved. To achieve this we must incorporate into the database schema the "temporary" storage (*e.g.*, we add a relation for funds in transit or for pending insurance claims). Also, transactions that need to see all the money must be aware of this new storage. Hence it is best if this storage is defined when the database is first designed and not added as an afterthought.

Even if L had no T_2 transaction, writing the missing funds in the database may be convenient. Notice that in this case L would release the locks on the temporary storage after T_1, only to immediately request them again in T_3. This may add some overhead to L, but in return for this transactions that are waiting to see the funds will be able to proceed sooner, after T_1. This is analogous to having a person with a huge photocopying job periodically step aside and let shorter jobs through. For this the coveted resources, i.e., the coping machine or the funds, must be temporarily released.

We believe that what we have stated in terms of money and LLT L holds in general. The database and the LLTs should be designed so that data passed from one sub-transaction to the next via local storage is minimized. This technique, together with a well structured database, can make it possible to write LLT's as sagas.

10. CONCLUSIONS

We have presented the notion of saga, a long lived transaction that can be broken up into transactions, but still executed as a unit. Both the concept and its implementation are relatively simple, but in its simplicity lies its usefulness. We believe that a saga processing mechanism can be implemented with relatively little effort, either as part of the DBMS or as an added-on facility. The mechanism can then be used by the large number of LLTs that are sagas to improve performance significantly.

ACKNOWLEDGMENTS

Bruce Lindsay provided several useful suggestions, including the name "saga." Rafael Alonso and Ricardo Cordon also contributed a number of ideas.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014–85-C-0456 and N00014–85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

REFERENCES

Ande81a. Anderson, T. and P. A. Lee, *Fault Tolerance, Principles and Practice*, Prentice-Hall International, London, 1981.

Date81a. Date, C. J., *An Introduction to Database Systems, (3rd Edition)*, Addison-Wesley, Reading, MA, 1981.

Garc83a. Garcia-Molina, Hector, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 186–213, June 1983.

Gray78a. Gray, Jim, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. G. Seegmuller, pp. 393–481, Springer-Verlag, 1978.

Gray81a. Gray, Jim, "The Transaction Concept: Virtues and Limitations," *Proceedings of the Seventh Int'l. Conference on Very Large Databases*, pp. 144–154, IEEE, Cannes, France, Sept., 1981.

Gray81b. Gray, Jim, Pete Homan, Ron Obermarck, and Hank Korth, "A Straw Man Analysis of Probability of Waiting and Deadlock," IBM Research Report RJ3066 (38112), IBM Research Laboratory, San Jose, California, Feb., 1981.

Hamm80a. Hammer, Michael and David Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Transactions on Database Systems*, vol. 5, pp. 431–466, December, 1980.

Horn74a. Horning, J. J., H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," in *Lecture Notes in Computer Science 16*, ed. C. Kaiser, Springer-Verlag, Berlin, 1974.

Lync83a. Lynch, Nancy, "Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control," *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 484–502, December, 1983.

Mossa. Moss, J. Elliot B., "Nested Transactions: An Introduction," unpublished, U.S. Army War College.

Rand78a. Randell, B., P. A. Lee, and P. C. Treleaven, "Reliability in Computing System Design," *Computing Surveys*, vol. 10, no. 2, pp. 123–165, ACM, June, 1978.

Ullm82a. Ullman, Jeffrey D., *Principles of Database Systems, (2nd Edition)*, Computer Science Press, Rockville, MD, 1982.