SAGAS

Hector Garica-Molina

Kenneth Salem

CS-TR-070-87

January 1987

SAGAS

Hector Garcia-Molina

Kenneth Salem

Department of Computer Science

Princeton University

Princeton, NJ.08544

摘要

一个长时间事务会在相对较长的时间内占用数据库资源，明显的阻碍了较短的和公用的其他事务完成。为了缓解这些问题, 我们提出一个 saga的概念。它是由多个有序的事务组成、并且与其他事务可以交错的一个长时间事务（LLT），数据库管理系统保证成功完成 saga 中的所有事务, 或对部分进行事务补偿。saga的概念 和它的实施相对简单, 但它们有可能显著提高性能。我们分析了与 sagas 相关的各种实施问题，包括如何在不直接支持它们的现有系统上运行它们。我们进行了数据库和 LLT技术讨论, 使 sagas成为LLT解决方案的可能。

1987年1月7日

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.

January 7, 1987

SAGAS

Hector Garcia-Molina

Kenneth Salem

Department of Computer Science

Princeton University

Princeton, N.J. 08544

### 1 简介

### 1 INTRODUCTION

顾名思义，一个执行长生命周期的事务，即使没有其他事务的干扰，也需要大量的时间，可能需要数小时或数天。一个长生命周期事务，或者说 LLT，与大多数其他事务相比, 持续时间较长, 因为它访问了许多数据库对象，它有很长的计算过程，或因用户输入的停顿，或者多种因素的组合。举一个TTL的例子，根据银行交易记录生成每月账目报表, 用来处理保险公司的索赔，这个事务需要收集整个数据库的信息[Gray81a]。

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].

在大多数情况下, LLT存在严重的性能问题。由于它们是事务, 系统必须将它们作为原子操作执行, 从而保持数据库的一致性[date81a, ullm82a]。为了保持事务的原子性，系统通常会锁定事务访问的对象，直到事务它提交，而这通常发生在事务结束时。因此, 试图访问 LLT 锁住的对象的其他事务会被阻塞很长时间。

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.

此外，LLTs可能还会导致事务终止的几率上升。正如 [Gray81b] 中所说的，死锁的频率对于事务的“大小”是非常敏感的，也就是说, 事务访问的对象数。(在 [Gray81b] 的分析中, 死锁频率随事务大小的四倍而增长)。因此, 由于 LLTs 访问了许多对象, 它们可能会导致许多死锁, 相应地，也会有许多中断。从系统崩溃的角度来看，LLTs遇到故障的概率较高 (因为它们的持续时间), 因此更有可能遇到更多的延迟, 更有可能自己被中止。

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.

一般来说，没有解决办法可以消除 LLT带来的问题。即使我们使用不同的锁机制来确保 LLTs的原子性, 长时间的延迟和/高中止率仍然会存在：无论什么机制, 当一个事务需要访问的对象们，已经被 LLT 访问了，那么直到 LLT 提交这个事务才能被提交。

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.

但是, 对于特定的应用程序, 或许可以通过放宽 LLT原子性要求，来缓解这些问题。换言之, 在不牺牲数据库一致性的情况下, 某些 LLT可以在完成之前释放其某些资源, 从而允许其他等待资源的事务可以继续进行。

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.

为了阐释这个想法, 请思考航空公司的订票系统。这个数据库（或这可能实际上是来自不同航空公司的数据库集合）包含航班预定，并且这个事务 T 希望有多个预定。对于本讨论, 让我们假设事务T 是一个 LLT (比如说, 每次预订后, 客户都会暂停输入)。在这个应用程序中，在T 完成之前可能并不需要保留其所有的资源,。例如，T在 F1 上保留一个座位后，它可以立即允许其他事务在同一架飞机上预定座位。换句话说，我们可以把T看做是许多“小事务的集合“，每个小事务T1, T2, ..., Tn 就是预留各个座位。

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 F1, 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" T1, T2, ..., Tn, that reserve the individual seats.

但是, 我们不希望将 T 简单地作为多个独立事务的集合提交到数据库中 (dbms)，因为我们仍然希望 T 是一个单元, 要么全部成功或全部失败。我们不希望事务 T 在数据库中五个座位只保留了三个，然后（由于崩溃）而什么也做不了。另一方面，我们希望DBMS 能够保证 T 将所有预定都成功，或者如果 T 必须停止，那么也将取消所有已预定座位。

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.

这个例子表明了一种控制机制，不那么严格的执行事务的原子性，但仍然提供了一些保证措施，保证LLT是可以实现的。在本文中，我们将提出一个这样的机制。

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.

让我们用 saga 这一词，来形容一个 LLT ，这个 LLT 可以被分解为多个子事务的集合，这些子事务也可以和其他事务交织在一起。在这种情况下，每个子事务都是一个真正的事务，因为它保留了数据库的一致性。但是与其他事务不同的地方，对于 saga 中的事务彼此相关，并且要作为一个（非原子）单元去执行：当saga的任意部分如果发生了意外，那么必须要进行补偿。

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.

若要对发生意外的部分进行修正，则每个事务 Ti 需要提供一个补偿事务 Ci。这个补偿的撤销操作，从语义上来看，补偿事务可以取消 Ti 所执行的任何操作，但不一定要在数据库层面返回到 Ti 执行前的状态。在我们上面举的航公公司订票例子，如果 Ti 是在航班上预订座位，那么 Ci 可以取消这个预定（减少一个预留座位并进行一些检查）。但是 Ci 不能简单的只记录一个座位数量，因为在 Ti 预订座位到 Ci 取消预订座位这段时间内，还有其他事务进行着预订及取消，这可能会导致此航班座位的预订数量发生变化。

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

一旦将 saga 的 T1, T2, ..., Tn 的补偿事务定义为 C1, C2, ..., Cn-1 ，那么 这个系统将会作出如下保证，任何一个序列

T1, T2, ..., Tn

（最好是一个）或者这个序列

T1, T2, ..., Tj, Cj, ..., C2, C1

对于0 ≤ j < n 将被执行。

Once compensating transactions C1, C2, ..., Cn-1 are defined for saga T1, T2, ..., Tn, then the system can make the following guarantee. Either the sequence

T1, T2, ..., Tn

(which is the preferable one) or the sequence

T1, T2, ..., Tj, Cj, ..., C2, C1

for some 0 ≤ j < n will be executed.

Sagas 显然是一种常见的 LLT 类型。当 LLT 由一系列相对有序且独立的步骤组成时，每一步不必关注于全局一致性。例如，在银行中对所有账户进行一些常规的固定操作（例如收益计算），并且一个账户的计算和下一个账户的计算之间交互很少，在办公信息系统中，同具有一些常用 TTLs，拥有相对独立步骤，这些步骤也可以被其他事务所使用。例如，接收一个采购订单涉及到将订单信息录入数据库，更新库存，通知会计记账，打印装运订单等。这个模拟办公 LLTs的程序，可以应对交错的事务。实际上，在采购订单成功前，人们不会实际锁定库存，因此在那些完成之前没必要让计算机程序来锁定库存。

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.

再次说明，我们提出的银行和办公室的LLT例子不仅仅是一些正常事务的集合，他们是一个 sagas。有一个应用程序去”约束”（不能代表数据库的一致性约束）这些活动步骤不应该未完成。这个应用程序需要能够处理所有的账户，或保证购买订单完全处理。如果采购订单未成功完成，那么这些相关记录必须被理顺（例如库存不应该被扣减）。在银行的示例中，可能始终可以一直执行直到完成这个TTL。在这种情况下，可能没必要抵消未完成的LLT。

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.

请注意，saga的概念与嵌套事务的概念有关[Garc83a, Lync83a]。但是, 有两个重要的区别:

（a）一个 saga 嵌套只允许有2层，顶级的 saga 第一层，里面的简单事务为第二层。

（b）在外部层面看不提供完全的原子性。也就是说，某个saga可能看到其他saga的部分结果。（译者注：应该是指违反了事务的隔离性）

Note that the notion of a saga is related to that of a nested transaction [Garc83a, 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还可以被视为在[Garc83a, Lync83a]中描述的机制下运行的特殊类型的事务。这个约定可以使机制更通用，使 sagas 的实现（和理解）更简单，从而使它们更有可能在实践中被使用。

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.

要使我们提出的想法可行, 有两个要素是必要的: DBMS支持sagas 以及 LLTs 可以被分解成有序的事务。在本文的其余部分，我们将更详细地探讨这些要素。在第二至第七章节我们会探讨saga的处理机制如何实现。我们首先讨论应用程序程序员如何定义 sagas, 然后讨论系统如何可以支持他们。我们最初假设补偿事务只能遇到系统故障。稍后, 在第6节中, 我们将研究其他故障 (例如程序错误) 在补偿事务中的影响。

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.

在第8节和第9节中, 我们讨论了 LLT 的设计。我们首先证明, 我们 saga的顺序事务执行模型可以推广到包括并行事务执行, 从而扩大 LLT 的范围。然后, 我们讨论应用程序程序员可以遵循的一些策略, 以便确实编写的 LLT确实是 sagas 并可以利用其机制获益。

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. 用户设施

2. USER FACILITIES

从应用程序程序员的角度来看, 需要一种机制来通知系统一个 saga的开始和结束，每个事务的开始和结束，以及补偿事务。这种机制可能类似于传统系统中用于管理事务的机制 [gray78a]。

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].

特别是，当一个应用程序希望启动一个saga时，它就会向系统发出一个开始saga的命令。接下来是一系列的开始事务、结束事务命令，每一组开始、结束指令，都表示着每个事务的边界。在这些命令之间，应用程序将发出常规的数据库访问命令。在事务中，程序可以选择性的，发起一个由用户倡导的中止事务命令。这将中止当前正在执行的事务，但不会中止 saga。类似地，有一个中止saga的命令首先中止当前正在执行的事务，然后中止整个saga（通过执行补偿事务）。最终，还有一个结束saga的命令，用于提交当前正在执行的事务（如果有）并且完成这个saga。

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 user-initiated 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.

这些命令中的大多数将包括各种参数。 begin-saga命令可以将saga标识符返回给程序。 然后，该标识符可以在saga进行的后续调用中传递给系统。

Abort-transaction命令包含在中止后saga从哪里继续执行的地址作为其参数。End-transaction命令包含当前事务回滚所需补偿事务的标识符，该标识符包括补偿事务的名称与程序入口，以及各种可能会用到的参数。（我们假设每个补偿程序都包含他自己的开始事务和结束事务方法。在补偿事务中，abort-transaction 和 abort-saga 命令不允许执行。）最后，这个 abort-saga 命令可能包含着一个存储点作为参数，如下所述。

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.

请注意, 可以将其补偿事务将来可能需要的参数包含在数据库中的每个事务存储区。在这种情况下, 参数不必由系统传递, 它们可以在补偿事务启动时由其读取。另请注意, 如果end-saga 命令同时结束最后一个事务和saga, 则无需为最后一个事务进行补偿事务。相反, 如果只中止了事务, 那么它必须包括补偿事务的标识。

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.

在某些情况下, 可能需要让应用程序程序员通过存储点命令指示，saga的检查点可能会使用它。可以在事务之间发出此命令。它强制系统保存正在运行的应用程序的状态, 并返回保存点标识符以供将来参考。这样, 保存点就可以帮助减少 saga故障或系统崩溃后的工作量: 系统可以补偿自上次保存点之后执行的事务, 而不是补偿所有未完成的事务, 然后重新启动 saga。

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.

当然，这意味着我们现在可以执行 T1, T2, C2, T2, T3, T4, T5, C5, C4, T4, T5, T6。(第一次成功执行T2后，系统崩溃。然后使用T1执行后的保存点，但是要在这里重新启动，系统首先通过运行C2取消T2带来的影响。然后saga可以重新启动并重新执行T2。在执行T5之后发生了第二次失败。)这意味着我们必须修改上面给出的有效执行序列的定义，以包含这类序列。如果这些部分恢复序列无效，那么系统要么不采用保存点，要么在每个事务的开始(或结束)时自动采用保存点。

Of course, this means that we can now have executions of the type T1, T2, C2, T2, T3, T4, T5, C5, C4, T4, T5, T6. (After successfully executing T2 the first time, the system crashed. A save-point had been taken after T1, but to restart here, the system first undoes T2 by running C2. Then the saga can be restarted and T2 re executed. A second failure occurred after the execution of T5.) 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 save-points, or it should take them automatically at the beginning (or end)of every transaction。

我们到目前为止所描述的模式是相当普遍的, 但在某些情况下, 可能容易有一个更严格的模式。我们将在第5节后面讨论这种限制性模型。

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. 可靠地保存代码

3. SAVING CODE RELIABLY

在一个传统事务处理系统, 不需要应用程序代码就可以在崩溃后将数据库还原到一致的状态。如果一个正在运行的事务代码遭到破坏而终止, 系统日志中包含足够的信息来撤消事务的影响。在 saga 系统中，情况就不一样了，要在崩溃后完成正在运行的saga, 必须完成尚未完成的事务或运行补偿事务以中止该saga。在这两种情况下, 都必须有所需的应用程序代码。

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.

有各种各样的可能的解决这个问题。一种是在传统系统中处理系统代码时处理应用程序代码。请注意, 即使传统的DBMS不需要可靠地保存应用程序代码, 它也必须保存系统代码。也就是说, 如果故障破坏了运行系统所需的代码, 则传统的 DBMS无法重新启动。因此, 传统系统有手动或自动程序, 在DBMS本身的外部, 用于更新和存储系统的备份副本。

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 a failure destroys the code required to run the system. Thus, conventional systems have manual or automatic procedures, out- side the DBMS itself, for updating and storing backup copies of the system.

在saga处理系统中, 我们可以要求以相同的方式定义和更新saga的应用程序代码。创建的程序的每个新版本都将存储在当前系统区域以及一个或多个备份区域中。由于更新不在DBMS的控制之下, 因此它们不是原子操作, 并且可能需要手动干预, 以防在更新过程中发生崩溃。当一个saga开始运行时, 它将假定它的所有事务和补偿事务都已预定义, 它只会进行适当的调用。

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.

如果 sagas 是由受信任的应用程序程序员编写的, 并且不是经常更新, 则这种方法可能是可以接受的。如果不是这种情况, 最好将saga代码作为数据库的一部分来处理。如果saga代码只是作为一个或多个数据库对象存储, 则其恢复将是自动的。唯一的缺点是 dbms 必须能够处理大型对象（即代码）。有些系统不会能够做到这一点, 因为他们的数据模型不允许大的非结构化的对象, 缓冲区管理器不能管理跨越多个缓冲区的对象, 或其他一些原因。

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.

如果 dbms 可以管理代码, 那么 sagas 的可靠代码存储就变得非常简单。saga的第一个事务 T1 将所有进一步的事务输入数据库 (补偿或不补偿) 这在未来可能是必要的。当 T1提交, saga的其余部分已准备好开始。T1的补偿事务C1只需从数据库中删除这些对象也可以定义增量事务。例如, 在相应的事务Ti准备提交之前, 不需要将补偿事务Ci输入到数据库中。此方法稍微复杂一些, 但节省了不必要的数据库操作。

If the DBMS can manage code, then reliable code storage for sagas becomes quite simple. The first transaction of the saga, T1, enters into the database all further transactions (compensating or not) that may be needed in the future. When T1 commits, the rest of the saga is ready to start. The compensating transaction for T1, C1 would simply remove these objects from the database It is also possible to define transactions incrementally. For example, a compensating transaction Ci need not be entered into the data base until its corresponding transaction Ti is ready to commit. This approach is slightly more complicated but saves unnecessary database operations.

4. 逆向恢复

4. BACKWARD RECOVERY

当saga故障中断时, 有两种选择: 补偿已执行的事务向后恢复，或执行还未执行的事务转而恢复正常。 (当然, 在所有情况下, 正向恢复可能不是一种选择。)对于向后恢复，系统需要补偿事务, 对于正向恢复, 系统需要保存点。在本节中, 我们将介绍如何实现纯逆向恢复, 后面再讨论混合逆向正向和纯正向恢复。

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.

在DBMS中, saga执行组件 (SEC) 管理着sagas。此组件调用常规事务执行组件 (TEC), 该组件管理各个事务的执行。SEC的运作类似于 TEC的操作: SEC作为一个单元执行一系列事务, 而TEC作为一个 (原子) 单元执行一系列操作。这两个组件都需要一个日志来记录 sagas 和事务的活动。事实上, 将这两个日志合并为一个日志是很方便的, 我们将假设这里的情况就是这样。我们还将假设日志是双重的可靠性。请注意, SEC不需要并发控制, 因为它控制的事务可以与其他事务交错。

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.

所有saga命令和数据库操作都通过 SEC 进行。在执行任何操作之前, 每个 saga 命令 (例如 begin-saga) 都会记录在日志中。命令中包含的任何参数 (例如, 停止事务命令中的补偿事务标识) 也记录在日志中。开始事务和停止事务命令以及所有数据库操作, 被转发到 TEC, 它以传统的方式处理它们 [gray78a]

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]

当 SEC 收到中止传奇命令时, 它将启动逆向恢复。为了说明这一点, 让我们考虑一个已经执行交易 T1 和 T2 的传奇, 并且在T3执行的中途向 SEC发出一个中止saga的命令。SEC在日志中记录该命令 (以防止回滚过程中崩溃), 然后指示 TEC中止当前事务 T3。使用常规技术回滚此事务, 例如, 通过将 "之前" 值 (在日志中找到) 存储回数据库。

When the SEC receives an abort-saga command it initiates backward recovery. To illustrate, let us consider a saga that has executed transactions T1 and T2, and that halfway through the execution of T3 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 T3. This transaction is rolled back using conventional techniques, e.g., by storing the “before" values (found in the log) back into the database.

接下来，SEC会查询日志，并命令执行补偿事务C2和C1，如果这些事务的参数在日志中，则会使用这些参数。这两个补偿事务执行方式就像其他事务一样，当然，关于他们何时开始和提交的信息记录到日志中取决于TEC。（如果在此期间出现崩溃，系统将能够知道还有哪些工作要做。）当C1提交后，这个saga将会终止。日志中会记录一个信息，类似于由结束saga的命令创建的信息。

Next the SEC consults the log and orders the execution of compensating transactions C2 and C1. 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 C1 commits, the saga terminates. An entry is made in the log, similar to the one created by the end-saga command.

该日志还用于从崩溃中恢复。崩溃后, 首先调用 TEC 来清理挂起的事务。一旦所有事务中止或提交, SEC 将评估每个saga的状态。如果一个saga有相应的开始记录和结束记录在日志中，那么这个saga是完整的，并且必须不能再改变它。如果缺少结束记录, 则该saga将被中止。通过扫描日志, sec 发现了最后一次成功执行并且没有补偿的事务。那么要补偿这个事务和之前的所有事务。

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. 正向恢复

5. FORWARD RECOVERY

对于正向恢复，SEC要求所有缺失的交易都有可靠的代码副本和一个保存点。这个保存点会被用于应用程序或者系统，具体取决于是哪个中止了saga。（回忆一下，存储点标识符可以作为参数传递到 abort-saga 命令中。）在一个系统崩溃的情况中，恢复组件可以为每个saga指定最近期的保存点。

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.

为了说明SEC在这种情况下的操作，请考虑一个执行了事务T1，T2，并有一个保存点，然后又执行了事务T3。然后在执行T4时系统崩溃了。恢复后，系统必须首先执行一个逆向恢复到保存点（中止T4并且运行补偿C3）。确保代码运行T3、T4、等等后续事务是可用的之后，SEC在日志中记录了它决定重启saga，我们叫这种为混合恢复。

To illustrate the operation of the SEC in this case, consider a saga that executes transactions T1, T2, a save-point command, and transaction T3. Then during the execution of transaction T4 the system crashes. Upon recovery, the system must first perform a backward recovery to the save-point (aborting T4 and running C3). After ensuring that the code for running T3,T4, ... is available, the SEC records in the log it decision to restart and restarts the saga. We call this backward/forward recovery.

如第2节所述, 如果在每次交易开始时自动获取保存点, 则纯正向恢复是可行的。如果我们同时禁止使用 abrt-saga 命令, 那么就没有必要执行逆向恢复。(abort-transaction命令仍然是可以接受执行的。 这样做的好处是消除了补偿事务, 在某些应用程序中可能很难编写 (见第9节)。

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 back- ward recovery. (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).

在这种情况下，SEC成为一个简单的“持久”事务执行器，类似于持久性消息传输机制[Hamm80a]。 每次崩溃后，对于每个活动的saga，SEC指示TEC中止最后执行的事务，然后在此事务开始时重新启动 saga。

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

如果我们只将saga视为包含对各个事务程序的一系列调用的文件，我们可以进一步简化这一过程。 这里不需要显式的开始或结束 saga，也不需要开始或结束事务命令。 saga从文件中的第一个调用开始，到最后一个调用结束。 此外，每次调用都是一次事务。 正在运行的saga的状态只是正在执行的事务的编号。 这意味着系统可以在每次交易后以很少的成本获取保存点。

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

这种纯的正向恢复方法对于总能成功的简单LLT非常有用。 计算账户利息可能是此类LLT的一个例子。 单个帐户的利息计算可能失败（通过中止事务命令），但其余计算将不受影响

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.

使用操作系统术语，上述事务文件模型可以称为简单的EXEC或SCRIPT。 持久性SCRIPT的想法在操作系统中也很有用，以确保成功执行命令集合（假设每个命令作为事务执行）。 例如，典型的文本处理和打印作业包括几个步骤（例如，在UNIX中，方程处理，转发，打印）。 每个步骤都会生成一个或多个由以下步骤使用的文件。 持久性SCRIPT将允许用户启动长文本处理作业然后回家，相信系统会将它完成。

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.

在这种情况下, 我们还必须假设, 在saga中的每一个子事务如果它被重试足够多的次数，最终将会成功,。

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

6. 其他错误

6. OTHER ERRORS

到目前为止，我们假设用户提供的补偿事务代码中没有错误。 但是如果由于错误而无法成功完成补偿事务（例如，它尝试读取不存在的文件，或者代码中存在错误）会发生什么？ 事务可能会中止，但如果它再次运行，它可能会遇到相同的错误。 在这种情况下，系统卡住：它不能中止事务，也不能完成它。 如果在纯正向补偿方案中事务有错误，同样会出现类似情况。

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.

一种可能的解决方案是使用类似恢复块[Ande81a，Horn74a]的软件容错技术。 恢复块是在主块中检测到故障的情况下提供的备用或辅助代码块。 如果检测到故障，则系统重置为其原始状态，并执行辅助块。 辅助块旨在使用不同的算法或技术实现与主要逻辑相同的结果，希望规避掉主流程的故障。

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 preprimary 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.

恢复块的想法很容易转化为saga框架。事务是自然的程序块, 由 TEC提供失败事务的回滚功能。saga应用程序可以控制恢复块的执行。 中止事务 (或通知其事务已中止) 后, 应用程序要么中止saga, 要么尝试另一个事务, 要么重试主事务。请注意, 补偿事务也可以为其提供备用事务, 以使中止 sagas 更可靠。

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.

解决此问题的另一个可能的方式是手动干预。错误的事务首先被中止。然后给应用程序猿发出错误描述，程序猿可以纠正它。然后，SEC（或应用程序）重新运行这个事务并且继续处理这个saga。

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.

幸运的是，在手动修复事务时，saga并不持有任何数据库资源（比如锁）。因此，已经时间很长的saga将需要更长的时间，但实际不会对其他事务的性能产生重大影响。

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.

依靠人工干预绝对不是一个优雅的解决方案，但它是一个实用的解决方案。剩下的替代方案是将saga作为一个长期的事务来运行。当此LLT遇到错误时，它将完全中止，可能会浪费更多的精力。 此外，仍然必须手动纠正错误并重新提交LLT。 唯一的好处是在修复期间，LLT将是系统未知的。 在这个saga的情况下，在安装修复的事务之前，saga将继续在系统中挂起。

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. 在现有DBMS之上实现SAGAS

7. IMPLEMENTING SAGAS ON TOP OF AN EXISTING DBMS

在我们讨论saga管理器时，我们假设SEC是DBMS的一部分，可以直接访问日志。但是，在某些情况下，是可能可以在不支持sagas的现有DBMS上运行sagas的。只要数据库能够存储大型非结构化对象（即代码和存储点），但是，它涉及到给应用程序猿更多的责任，并可能损害性能。

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.

基本上有两件事要做。首先, 嵌入在应用程序代码中的saga命令成为子程序调用 (相对于系统调用)。(子例程与应用程序代码一起加载。)每个子程序都存储在数据库中sec 将存储在日志中的所有信息。例如, begin-saga 子程序将在活动 sagas 的数据库表中输入该saga的标识。Save-point子程序将导致应用程序将其状态 (或其状态的关键部分) 保存在类似的数据库表中。同样, end-transaction子程序在执行结束交易系统调用之前进入其他一些表, 即结束事务的标识及其补偿事务 (由 TEC 处理)

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 data-base 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).

在数据库中存储saga信息的命令（存储点除外）必须始终在事务中执行，否则信息可能会在崩溃中丢失。因此，saga子程序必须跟踪saga当前是否正在执行事务。如果开始事务设置了一个标志然后被结束事务重置，那么依靠事务是一种很容易实现方式。如果未设置标志，则禁止所有数据库的存储操作。请注意, 子程序方法仅在应用程序代码从不自行进行系统调用的情况下才有效。例如, 如果事务因系统调用end-transaction (而不是子例程调用) 而终止, 则不会记录补偿信息, 也不会重置事务标志。

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.

其次，必须有一个特殊的程序来实现SEC的其他职能。这个程序，这个 saga daemon（SD）将始终处于活跃状态。它在崩溃后会因操作系统而重启。崩溃后，它将扫描saga表，找出挂起状态的sagas。这个扫描将通过提交一个数据库事务来执行。只有在事务恢复完成后，TEC才会执行此事务。 因此，SD将读取一致性的数据。一旦SD知道了待处理的saga的状态，它就会发起必要的补偿或正常的事务，就像SEC恢复后所做的那样。必须注意在SD提交其数据库查询之前，不要干扰sagas崩溃后的正常启动。

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.

在TEC中止事务（例如，由于死锁或用户引起的中止）后，它可能只是中止这个事务的处理。在一个传统的系统，这可能是好的，但在saga中它导致saga其他工作没有完成。如果再这种情况发生时，TEC无法向SD发出信号，则SD必须定期扫描saga表，以发现这种情况。如果发现了，将立即采取纠正措施。

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.

正在运行的saga也可以直接向SD发起请求。例如，若要执行中止saga，这个中止saga子程序将请求发送到SD，然后（如果有必要）执行中止事务。

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. 并行SAGAS

8. PARALLEL SAGAS

我们在saga中执行顺序事务的模型可以扩展到包括并行事务。在saga天然支持并发事务在应用程序中是很有用的。例如，在处理采购订单时，最好同时生成装运订单并更新应收账款。

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.

我们假设saga进程（父进程）可以创建新的进程（子进程），他们可以并行，请求类似于UNIX中的分岔请求。该系统还可以提供join功能，以组合saga中的流程。

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.

并发saga的逆向的崩溃恢复类似于有序的sagas。在并发saga中的每个处理，事务都按照相反的顺序进行补偿（或撤销），就像使用有序saga一样。此外，当父进程执行了一些事务后创建了子进程，那么必须将子进程的补偿完成后，才能执行父进程的补偿。 (请注意, 只有有序执行的事务限制了补偿的顺序。如果 T1T2 已在并行进程中执行, T2 读取了 T1 写入的数据, 补偿了 T1并不强迫我们先补偿 T2。)

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 T1 and T2 have executed in parallel processes and T2 has read data written by T1, compensating for T1 does not force us to compensate for T2 first.)

不同于逆向崩溃恢复，逆向恢复一个失败的并发的saga更为复杂，因为saga可能有多个程序组成，所有这些都必须中止。~ 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. ~ 当其中一个saga程序请求abort-saga，这个SEC姜夔杀死所有涉及的过程中的saga。然后, 它将中止所有等待的事务并补偿所有已提交的事务。

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.

因为有可能存在“不一致”的保存点，正向恢复更加复杂。为了说明这一点，请思考图8.1中的saga。每个框表示一个进程；每个框都有进程需要有序执行的事务和存储点（SP）。下面的那个进程是在T1提交后fork的。假设T3和T5是当前正在执行的事务，并且保存点在T1和T5之前执行的。

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 T1 committed. Suppose that T3 and T5 are the currently executing transactions and that save-points were executed before T1 and T5.

---------------------------------

| T0-->sp-->T1-->T2-->T3 |

---------------|-----------------

| |--------------------------|

|----> T4 --> sp --> T5 |

|--------------------------|

Figure 8.1

在此时系统失败。上面的进程将会在T1前面重启。因此，第二个进程所做的保存点是没用的。它取决于T1的补偿的执行情况。

At this point the system fails. The top process will have to be restarted before T1. Therefore, the save-point made by the second process is not useful. It depends on the execution of T1 which is being compensated for.

此问题称为级联回滚。在流程通过消息 [rand78a] 进行通信的情况下, 对问题进行了分析。在那里, 可以分析存储点依赖关系, 以获得一组一致的存储点 (如果存在)。然后, 一致的集可用于重新启动进程。对于并行 sagas, 情况更加简单, 因为存储点依赖关系仅通过分叉和联接以及进程中的事务和存储点顺序产生。

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.

为了达成一套一致的存储点，SEC必须被告知程序的fork和join。这个信息必须存储在日志中，并在恢复时进行分析。SEC会在saga的每个程序中选择最新的存储点，这样更早的事务就不会补偿。（如果事务在存储点之前执行，但在存储点被加载后，这个事务必须要补偿）。如果程序中没有这样的存储点，则必须回滚整个程序。对于具有存储点的程序，可以进行必要的逆向恢复并重启整个程序。

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. 设计SAGAS

9. DESIGNING SAGAS

我们描述的saga处理机制只有在应用程序猿将他们的LLT编写为saga时才有用。因此，随之而来的问题：程序猿如何知道一个指定的LLT是否可以被安全的分解为一系列有序的事务？程序猿如何选择断点？写补偿事务有多苦难？在本节中，我们将讨论其中的一些问题。

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.

要辨别出潜在的子事务，必须寻求正在执行工作的自然划分边界。在许多情况中，LLT模型是一系列现实世界的动作，其中每个动作都saga子事务的候选人。举个栗子：当大学生毕业时，必须采取若干行动，才能颁发文凭：图书馆必须检查没有未归还的书籍，必须检查住房账单和学费都核对通过，学生的新地址必须被记录下来；显然，这些现实世界中的每个操作都可以成为一个事务模型。

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.

在其他情况下，数据库本身作为自然的相对独立的组件，并可以将每个组件上的操作分组为saga的事务。举个栗子，考虑下大型操作系统的源码。通常，操作系统及其程序可以分为调度程序、内存管理器、中断处理程序等组件。一个LLT是向操作系统添加一个跟踪工具，这个跟踪工具可以分解到每个组件上，每一个作为有一个事务。同样，如果员工数据可以按工厂位置来切分，那么给员工发放生活费补贴的LLT也可以按照工厂进行拆分。

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.

为LLT设计补偿事务是一个非常普遍的难题。（例如，如果事务触发一枚导弹，则可能无法撤销此操作）。然而，对于许多实际应用来说，它可能和编写事务本身一样简单（或困难）。事实上Gray在[Gray81a]中指出，在应用程序中事务通常有相应配套的补偿事务。特别是类似真实世界的可以撤销的事务模型，例如预定一个出租车或者商场购物下单。在这种情况下，编写补偿事务或普通事务非常相似：程序员必须编写执行操作的代码并保证数据库一致性约束。

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.

虽然困难但上述这些仍是可以实现的，例如寄信或者打印支票，寄信完了想撤回可以再寄一封说明，支票印完了想反悔可以赶紧向银行发个信息让他们终止操作。当然最好不要有这种，单如果按常规方式处理代价很高的话，那么开发者就往往不得不采用这种方式。

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 stop-payment 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.

当然，正向补偿可以避免这种（见第五部分），因此如果这种补偿机制很难写的话，考虑把应用适配成能够采用纯正向补偿的，这样就可以避开类似的中止操作了。

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.

正如我们所讨论的，DB的结构在saga设计中扮演着关键角色，因此最好不要孤立地研究每种LLT，而是在设计数据库时就把如何适配LLT和saga考虑进去。如果DB能被设计成一套松散链接的组件（加上一点儿简单的组件内一致性约束），那么LLT应该就能自然而然地以子事务形式被组织在一起。

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.

另一项对于转换LLT而言可能比较有用的手段是将LLT所涉及到的临时数据存储在数据库内部。为了说明这点，考虑一个包含三个子事务T1,T2,T3的LLT，记为L。在T1中，L执行一些操作，然后从数据库中存储的账户条目中取出一些钱。这一金额暂时存储在临时本地变量中，直到T3才将这部分钱写入到另一个（或几个）账户里。T1完成后，数据库处于不一致状态：一部分钱“消失”了，即无法在数据库中查到这部分金额。因此，L不能被视为saga运行。如果这样的话，在T1至T3之间运行、需要查询全部金额的操作（例如审计）就无法查询到全部资金了。如果L作为普通事务执行，那么审计操作就需要排队等L结束。这虽保证了一致性但却损害了性能。

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 T1, T2, and T3. In Ti, 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 T3 the funds are placed in some other account(s). After Ti 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 Ti and T3 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.

然而，如果不是将这部分钱存到L本地，而是存到数据库中，那么数据库就一致了，其他事务也可以同时运行。为实现这点，我们必须将【临时存储】概念引入到数据库中（例如为处于中间状态或用作”保险“目的的资金加入一种关系）另外，需要查询到全部金额的事务必须知道这个新的存储。因此最好能在设计数据库的时候就把这个临时存储考虑进去，而不是事后再往里加。

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.

如果L没有T2事务的前提下，在数据库中写入少的这部分金额会变得非常方便，但这种情况下L会在T1执行后释放掉对临时存储的锁，然后等到T3时才重新再去申请它们。这会给L增加一些额外开销，但作为汇报，需要查看全部金额的事务得以在T1之后更快地运行，这有点儿像让冗长事务为小事务让道。为此必须暂时释放对资源的占有。

Even if L had no T2 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 Ti, only to immediately request them again in T3. 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 T1. This is analogous to having a person with a huge photocoping 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.

我们认为上述的”钱“和LLT L的总体陈述是成立的。数据库和LLT应被设计得能在一个子事务向另一个子事务传递最少的本地存储数据。这项技术与结构化数据库相结合，就可以将LLT作为saga写入了。

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.

1. 结论

10.CONCLUSIONS

我们展示了saga的概念：一个长时间存活大事务能够被分割成若干小事务且执行时被视为一个整体。其概念和实现均相对简单且实用。我们认为saga的实现机制能够用相对较小的代价实现出来（以数据库的一部分或者以附加设施的形式）。这一机制可以被用于处理大量可被视为SAGA的LLT，并显著提高处理性能。

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.

//TODO

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. NO0014-85-C-0456 and NOO014-85-K-0465 and by the National Science Foundation

under Cooperative Agreement No. DCR-842094. 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

References

Ande81a.

Anderson, T and P. A. Lee, Fault Tolerance, Principles and Practice, Prentice-Hall

International, London, 1981

17-

Date81a.

Date,C. J., An Introduction to Database Systems,(Brd 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,June1983

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 Sys-

tem 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 Sci-

ence 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

-18-

Design, Computing Surveys, vol 10, no. 2, pp.123-165, ACM, June, 1978.

Ullm82a.

Ullman, Jeffrey D, Principles of Database Systems,(2nd Edition), Computer Sci-

ence Press, Rockville, MD, 1982.