Nested Transaction: An Efficient Facility to Enforce the Nesting and the Partial Ordering Requirements in S-Store

Chenggang Wu
Advisor: Stan Zdonik
Reader: Uğur Çetintemel

April 28, 2015

Abstract

The goal of this thesis is to design and implement an efficient facility to enforce the nesting and the partial ordering requirements of transactions in S-Store [1], the world's first streaming OLTP engine for real-time applications. We first compare and contrast different approaches to enforce these requirements, and conclude that nested transaction stands out both in terms of data integrity guarantees and performance. We then offer details regarding the design and implementation of nested transactions in both single-partition scenarios and distributed scenarios. Finally, we discuss several optimization techniques to further improve the performance of nested transaction.

1 Introduction to S-Store

In today's big data applications, there is a growing need for managing high-velocity data streams, and answer real-time analytic queries promptly and accurately. Existing stream processing engine, such as Aurora [2] and Borealis [3], aimed to reduce the latency of query results for executing SQL-like operators on an unbounded and continuous stream of input data. However, since the developers did not consider the effect of stored state, these systems did not support ACID transactions. The lack of system-level support for transaction might put applications into inconsistent states, and provide weak guarantees for isolation and recovery semantics.

In order to support both streaming and transaction processing, we have S-Store [1]: a transactional streaming engine that can simultaneously accommodate OLTP and streaming applications. It employs a transaction model for streams that can

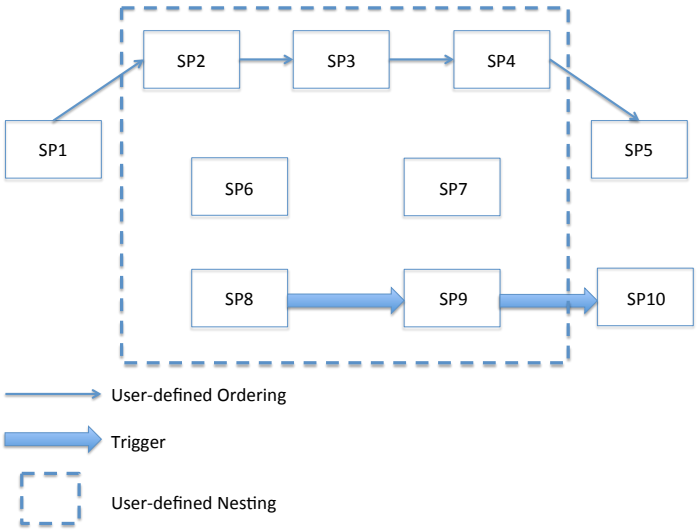

Figure 1: An Example Client Request

be integrated into traditional OLTP systems, and attempts to offer low-latency query response, and at the same time provide consistency guarantees. S-Store is built as an extension of H-Store [4], an in-memory distributed OLTP database management system, and it supports all transaction processing facilities that H-Store currently has.

A transaction template in S-Store is predefined as a stored procedure with certain input parameters. A stored procedure normally contains a mix of SQL and Java code. A transaction is initialized by instantiating an instance of the stored procedure object. There are two types of transaction in S-Store: Streaming transactions which are responsible for processing incoming data streams, and OLTP transactions which are used to support ad-hoc inquiries and display summaries of the stored states. After one streaming transaction finishes, depending on the result of this transaction, it may trigger other streaming transactions. When a batch of input tuples arrives, it will get processed via a sequence of streaming transactions, each of which is triggered by the previous transaction. The whole sequence of streaming transactions forms a dataflow graph. When a user issues a request to S-Store, the request typically contains the input data stream, a number of dataflow graphs to process the input data, and some other OLTP transactions.

An example of client request is shown in Figure 1. In this request, SP8, SP9,

and SP10 form a dataflow graph, and SP1 through SP7 represent OLTP transactions. The nesting and user-defined ordering will be explained in detail in the next section.

2 Additional Requirements

In S-Store, users are allowed to impose some additional requirements regarding how transactions should be executed. First of all, If there is a dataflow graph in the request, then all the transactions within the dataflow graph must be executed in the order specified because one transaction has to wait to be triggered by the previous transaction. Users can also specify groups of transactions that must be executed in a given order, regardless of whether or not they are part of the dataflow graph. H-Store currently enforces this constraint by serial scheduling all the transactions that have the ordering requirement. As can be seen from our example request, SP1 through SP5 have a user-defined ordering requirement.

Besides the partial ordering constraint, users can specify nestings of transactions (the nesting can include both streaming and OLTP transactions) that accomplish a single task. In this case, we should treat all transactions within the nesting as a single transaction, which means all transactions should share the same fate (either all of them commit or all of them abort), and any intermediate state should not be visible to transactions outside of the nesting. H-Store currently does not enforce this requirement, and the major focus of this thesis is to design and implement an efficient facility that provides ACID guarantees for those nestings of transactions. In Figure 1, user-defined nesting is shown by a dashed square.

3 A Simple Solution

One simple approach to solve this problem is that instead of defining transactions within the nesting as seven separate transactions, the user can merge them together and define a larger transaction that does the equivalent job as the combined effect of these seven transactions. Since now they are all in one transaction, the ACID properties are guaranteed.

However, there are two major disadvantages to this approach. First of all, whenever the users want to define new nestings using the existing stored procedures, they need to first extract the SQL and Java code from each of the stored procedures within the nesting, and merge them together to build a new stored procedure. This is fairly inefficient from the software engineering perspective, and can cause unnec-

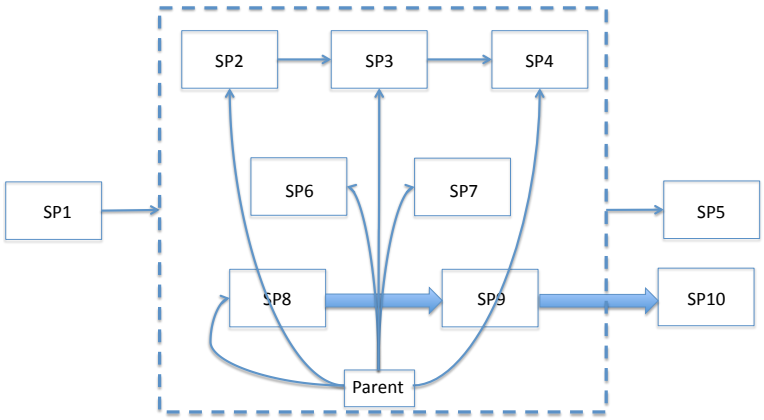

Figure 2: An Example Nested Transaction

essary code repetition. This approach also imposes a burden on the users' side as it requires them to figure out how to extract and merge stored procedures correctly.

The other disadvantage is that this approach will bring down the performance of transactions within the nesting. The reason is that there might be a partial ordering constraint for some transactions in the nesting, and other transactions can be executed in any order. Since the partial ordering is imposed by the users, a lot of distributed database management systems including H-Store are currently unable to figure out which pieces of code fragment are required to be executed in order and which pieces do not. Therefore, they employ a conservative approach that executes the code serially without any concurrency. But for those transactions that are independent of each other, they should have been executed in parallel. Hence, this approach is not ideal in terms of performance.

4 A Better Solution: Nested Transaction

Another way to address this issue is to introduce nested transaction. An example of nested transaction is shown in Figure 2. Each transaction within the nesting becomes a child transaction, and we construct a parent transaction to initiate and monitor all the child transactions. To the transactions outside of the nesting, the entire nested transaction is viewed as a single transaction. When each child finishes execution, instead of committing or aborting directly, it notifies the parent of its

commit status. After the parent receives all the responses from its children, it then commits all the child transactions if and only if all of them reported to have finished execution successfully. Otherwise the parent will abort all child transactions.

In this design, all child transactions share the same fate because the centralized commit all and abort all are done by the parent transaction. Also, the intermediate changes made by the child are not visible to other transactions until the parent commits. Therefore, this approach provides the same level of ACID guarantees as the previous approach.

However, this approach has two major advantages over the previous one. When the users define new nestings of transactions, they can simply declare a new parent transaction and pass in the name, parameter, and the partial ordering requirement of each child transaction. This makes the code less repetitive and releases the burden of mixing code fragment for different transactions from the users' side.

Additionally, since the parent is aware of the ordering requirement, it is able to execute those who have the ordering requirement (SP2, SP3, and SP4) serially and who do not (SP6 and SP7) concurrently. Compared to the previous approach where the code fragments are executed serially, this approach offers a more efficient way to process transactions that are independent of each other. Therefore, it provides better performance.

5 Implementation

5.1 Basic Settings

The transaction model of S-Store is built on top of H-Store. In a distributed setting, there are several H-Store sites. Each site consists of several partitions. When the user issues a transaction request, the request is received by the H-Store site and queued into a transaction queue at a certain partition. Since H-Store does not have a fine-grained locking mechanism, at each partition, transactions are executed serially by a single threaded partition executor based on their timestamp.

5.2 Defining Parent Stored Procedure

To create a normal stored procedure, the users define a sequence of operations to be executed in the *VoltProcedure*'s main function. To create a parent stored procedure, the users instead create a *VoltTable* in the *VoltProcedure*'s main function, and insert the name of each child stored procedure, its input parameter, and a flag

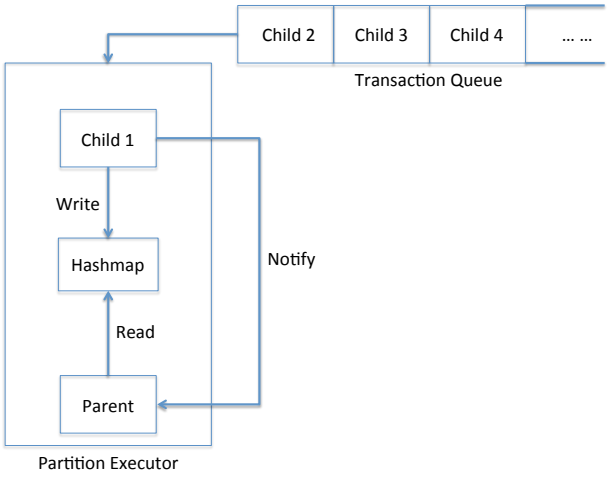

Figure 3: Single-Partition Implementation

indicating whether or not it has a partial ordering constraint as one tuple into the *VoltTable*.

5.3 Nested Single-Partition Transactions

A parent transaction is considered a single-partition transaction if all the child transactions only need to access one common partition. The implementation of nested single-partition transaction is shown in Figure 3. When a parent transaction is being executed, it first reads the *VoltTable* created by the user to extract information of the children. Then, it writes the information of each child into a bytebuffer, puts the buffer into a transaction invocation request, and sends the request to the proper H-Store site for queuing. After that, it creates another thread that keeps pulling the children for execution, and goes to sleep.

When a child finishes execution, instead of committing or aborting by itself, it first writes its transaction status into a hashmap that the partition executor has access to, and then continues to pull another child for execution.

The parent has a special data structure that keeps track of how many children has been queued, and which child has finished execution and written its information to the hashmap. After all children finish execution, the parent loops through the hashmap to see if any of its children has aborted. If there exists a child that

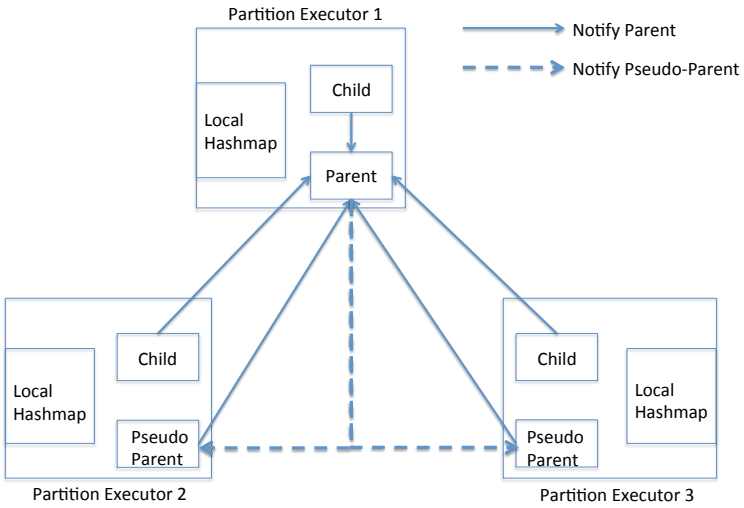

Figure 4: Multi-Partition Implementation

has aborted, the parent aborts all of its children. If not, the parent commits all of its children.

5.4 Nested Multi-Partition Transactions

A parent transaction is considered a multi-partition transaction if its child transactions need to access different partitions. For simplicity, we assume that the data is well-partitioned such that each child is going to be a single-partition transaction. The implementation of nested multi-partition transaction is shown in Figure 4. The multi-partition case is more complicated than the single-partition case for two reasons. First of all, since a child may access different partitions, we need a protocol that allows communication between the parent and the child across partitions. Additionally, since a transaction can only be committed or aborted by the local partition executor, when the parent and the child are executing at different partitions, the centralized commit/abort mechanism in the single-partition scenarios no longer applies.

In order to resolve the above challenges, we give the following implementation: Whenever a child arrives at the remote partition, it will first gets pulled and executed by the remote partition executor, and then writes its transaction status into a hashmap that the remote partition executor has access to. Then, it queues a special transaction, *NotifyParent*, to the parent partition informing of its execu-

tion status. Since this child transaction is the first child that arrives at the remote partition, it will now act as a pseudo-parent transaction. It then creates another thread that keeps pulling other children that get queued at the remote partition for execution, and then goes to sleep.

When the subsequent child at the remote partition finishes execution, it first writes its transaction status into a hashmap that the remote partition executor has access to, and then queues *NotifyParent* to the parent partition informing of its execution status. After that, it continues to pull another child for execution.

When the parent transaction receives notifications from all of its children, it checks if any of them has aborted. Based on the result, the parent sends special transactions, *NotifyPseudoParent*, to all remote partitions that are involved to let them commit/abort all of its children that get executed on the remote partitions. At each remote partition, after the pseudo-parent receives the special transaction sent by the parent, it loops through the hashmap to commit/abort transactions as instructed.

By doing so, the child executing at a remote partition can notify the parent by sending *NotifyParent*, and the parent can notify a pseudo-parent at the remote partition to commit/abort its child by sending *NotifyPseudoParent*. For each group of children that get executed at a remote partition, they will get committed/aborted by the pseudo-parent (the first child that arrived at the remote partition) instead of the parent.

5.5 Triggered Child Transactions

In a traditional nested transaction facility, the parent transaction knows the total number of child transactions before the execution. In S-Store, triggers are used to enable push-based, data-driven processing needed by the streaming transactions. After a streaming transaction finishes, it may or may not trigger other streaming transactions depending on the output stream. Therefore, in S-Store the parent is unable to know the number of children beforehand because of the triggering mechanism. This makes it difficult to track the execution status of the triggered transactions.

To resolve this issue, we made the following changes: When a child finishes execution, before sending *NotifyParent*, it checks to see if it has triggered any transactions and record the number of transactions triggered. It then includes this information in the *NotifyParent* and sends it to the parent. When a triggered transaction finishes execution, it includes the information that it is a triggered

transaction in the *NotifyParent* and sends it to the parent.

For the parent, we add a counter for it to keep track of the total number of transactions triggered. When the parent receives the message from the triggering child, it increments the counter as instructed. When the parent receives the message from the triggered child, it decrements the counter as instructed. The parent starts commit/abort all of its children if and only if it has received *NotifyParent* from all the children that are queued by itself as well as all the children that are triggered by other child transactions. Note that the latter can be checked using the counter. When the counter becomes zero, it means that all the triggered transactions have gotten back to the parent.

5.6 Multi-User Environment

The above model works when we have a single user who queues transaction requests in a serial order. However, in reality there will be multiple users queuing several transaction requests concurrently. In this scenario, when the parent transaction pulls the next transaction from the queue, it has to make sure that the transaction that gets pulled is its own child. Also, we need an appropriate deadlock prevention mechanism in the presence of nested transactions.

To do so, we assign globally unique timestamps to each non-nested transaction that gets queued by the users (This is already supported by H-Store). For each nested transaction, instead of assigning one timestamp, we assign a unique range of timestamps to the parent. When the parent queues its children, it will distribute the timestamp within the unique range to its children. As described in section 5.5, since some children can trigger other streaming transactions, the parent is unable to know the exact number of children beforehand. However, since the length of the predefined dataflow graph is fixed, the parent knows the upper bound of how many children can be triggered, and can therefore requests the range based on the bound. An example of timestamp allocation is shown in Figure 5.

For each partition, we also changed the implementation of the transaction queue from a FIFO queue to a priority queue with the priority being the transaction timestamp. By doing so, when the parent pulls the next transaction from the queue, it first checks if the timestamp of the transaction is within the range that it is responsible for. If it is, then it means that the transaction is one of its children, and it will get pulled and executed. If the timestamp is bigger than the range, then it means that this transaction is not one of its children, and should get executed after the current nested transaction finishes. In this case, the parent will keep pulling the queue until it gets one of its children. This is possible because

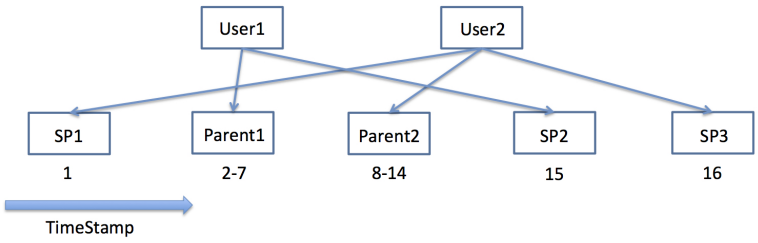

Figure 5: Multi-User Timestamp Allocation

all children will have a smaller timestamp, and will eventually get queued at the front of the priority queue. However, if the timestamp of a transaction pulled by the parent is smaller than the range, it means that this transaction is not one of its children, and should have been executed earlier. To avoid deadlock, the parent aborts the transaction, assigns a new (bigger) timestamp, and restarts it. This is a modified version of the Wound-Wait deadlock prevention algorithm.

5.7 Recovery

Since S-Store is an in-memory streaming engine, in the face of failure, all the data in the main-memory is lost and S-Store must rely on checkpoint and command-logging in order to recover to a legal state. With nested transactions, S-Store must recover to a state such that either all child transactions within a parent take effect or none take effect. To achieve this, only the parent is included in the command-log, and any child that has to be initiated by the parent is not included. In this way, while recovering from failure, either the whole nested transaction gets replayed, or the system pretends as if it never happened. Therefore, it guarantees to bring the system back to a legal state where no intermediate state of the nested transaction is revealed.

5.8 Optimization

In S-Store, we can do some optimization to improve the performance of nested transactions. First of all, within the parent, some children have a partial ordering requirement and some do not. For those who do not have a ordering requirement, instead of queuing one and waiting for the response, we can queue them altogether. If some of them can be executed in different partitions, we can then execute them in parallel to improve the performance.

Also, for those children who have a partial ordering requirement, instead of checking their transaction status after all of them finish execution, we can check the sta-

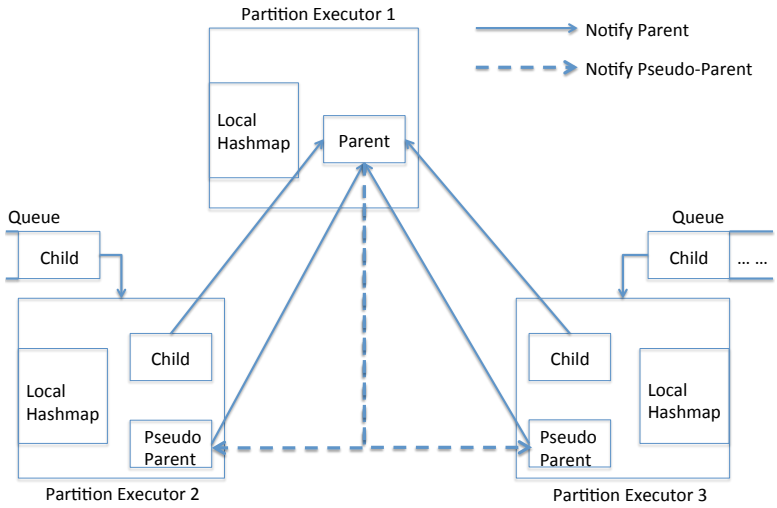

Figure 6: A Suboptimal Parent Scheduling

tus every time after the parent get a response from the previously queued child. If the previous child aborts, then the parent can abort immediately without queuing subsequent transactions. By short circuiting these transactions, we can improve the performance of nested transactions.

If we have some additional knowledge about the child transactions, such as which partition it is going to access, then we can schedule the parent transaction wisely to minimize the inter-partition communication overhead and to release the resources for certain partitions. The example in Figure 6 shows a suboptimal scheduling of parent. If we have no information about which partition the children are going to access, the S-Store scheduler will schedule the parent onto a random partition. In the example, the parent is scheduled onto partition 1. However, there might be cases that none of its children is going to access this partition. Therefore, the parent not only has to do unnecessary communication across partitions, but also blocks partition 1 so that other transactions that need to access the partition have to wait until the entire nested transaction finishes, which is extremely inefficient. On the other hand, if we know which partition its children are going to access, we can schedule the parent onto partition 2 or 3 to minimize inter-partition communication overhead and at the same time release the resource for partition 1.

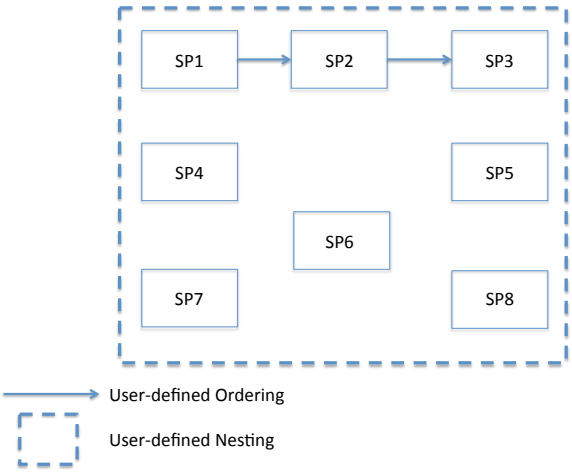

Figure 7: The Client Request for the Experiment

6 Experiment

The goal of the experiment is to compare the performance between the simple approach and the nested transaction approach.

6.1 Setup

In our experiment, S-Store is set up on a machine with a 2.8GHz dual-core Intel i7 CPU and 6GB of RAM running Linux Mint 17. The client request used for the experiment is shown in Figure 7. The functionality of each stored procedure is to insert a certain number of tuples at a partition. Note that SP1, SP2, and SP3 have a user-defined ordering requirement, and other stored procedures can be executed in any order. In our experiment, we fix the number of partitions in S-Store to be five, and vary the latency per each stored procedure by varying the number of tuples inserted by each stored procedure.

6.2 Results and Analysis

The result of our experiment is shown in Figure 8. As a baseline comparison, we also include the case where all the transactions are initiated separately by the user. The experiment shows that when the latency per each stored procedure is low, the simple approach performs slightly better than the nested transaction approach. As the latency increases, the nested transaction approach starts to outperform the

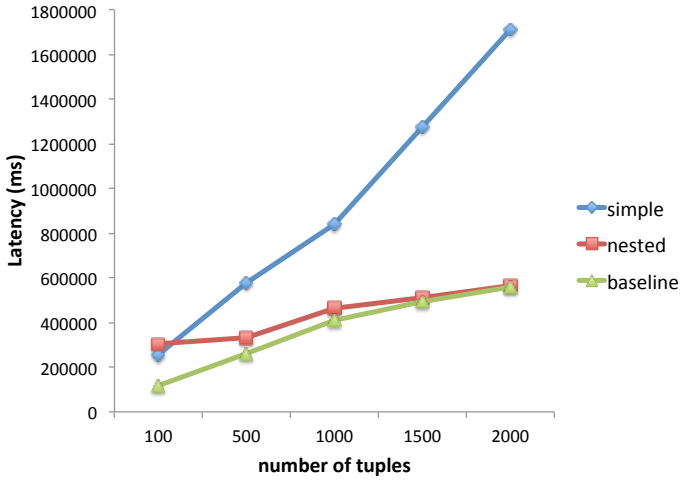

Figure 8: Performance Comparison Result

simple approach. When the number of tuples inserted by each stored procedure grows to 2000, the nested transaction approach can achieve up to 4x lower average latency over the simple approach. The reason is that in the low-latency case, the bottleneck for nested transaction lies in the additional communication overhead between parent and its children. However, as the latency per stored procedure increases, the additional communication overhead becomes negligible. Since the parent transaction is able to execute stored procedures without the ordering requirement (SP4 through SP8) in parallel, it manages to outperform the simple approach. In general, the baseline approach performs the best because transactions are scheduled optimally by the user. We also notice that the performance of the nested transaction approach is similar to the baseline approach, which again shows that our parent can schedule its children in an efficient way.

7 Future Direction

7.1 Distributed Child Transactions

In this thesis, we assume that the data is well-partitioned such that each child transaction is a single-partition transaction. While this is a valid assumption, it would be interesting if we could extend the nested transaction facility to the most general case, where each child can also be a distributed transaction. An example of this scenario is shown in Figure 9. As can be seen from the figure, in this scenario we have two layers: one layer between the parent and its children, and the other layer between the child and its sub-transactions. It would be interesting

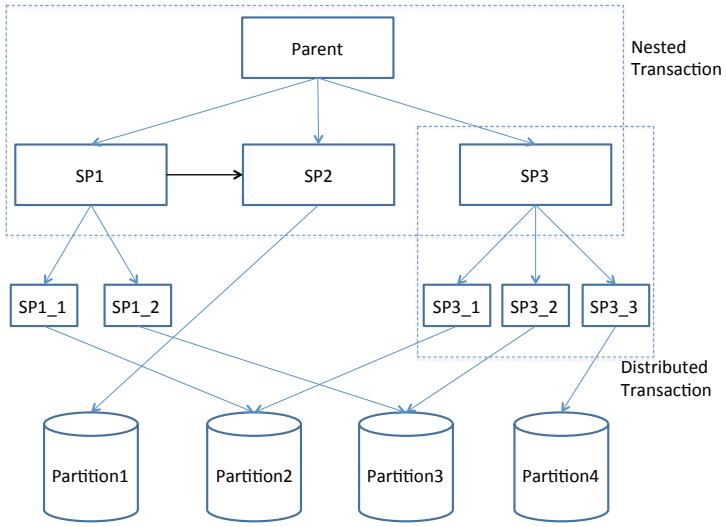

Figure 9: Distributed Child Transactions

to investigate under the existence of nested transactions, what additional work should be done to facilitate the distributed transaction for each child transaction.

7.2 A More Powerful Parent

The current nested transaction facility in S-Store wraps its children and treat them as an atomic unit. In general, however, nested transaction can be more powerful. The parent might be able to do some computation on its own, and depending on the computation result and the responses from its children, decide to abort some of its children while committing the others. Although the current S-Store does not require such functionality, it might be useful in the future.

8 Related Work

Nested transaction has been studied in the past. In [5], a nested transaction mechanism is implemented in LOCUS, an integrated distributed operating system. In their implementation, the users are able to write transactions as modules that can be composed freely. This allows the users to construct new transactions by composing existing transactions. However, instead of letting the children share the same fate, they allow each child to fail independently of others. In case that a child fails, a new back-up child will be invoked by the parent in order to accomplish a similar task.

In [6], the authors argue that although lower level processes should be executed in strict isolation, they do not have to be strictly atomic. This is because within a process, there are some subprocesses that are critical, and others that are not. To achieve weak atomicity for these processes, the authors build a nested transaction facility where the parent can commit part of its children while aborting others based on the importance of each child.

Our version of nested transaction is implemented in a transactional streaming engine, where certain groups of transactions have a predefined ordering requirement. Our implementation enables the parent to figure out which children have to be executed serially and which can be executed concurrently.

In the original H-Store, the users are allowed to add the nesting functionality at the application's level. However, compared to our approach, it not only puts the burden on the users' side, but also requires more round-trips between the client and the partition engine, thereby bringing down the performance. Therefore, it is more efficient to build the nested transaction facility inside the system's layer.

9 Conclusion

The goal of this thesis is to design and implement an efficient facility that enforces the nesting and the partial ordering requirements in S-Store. After comparing a simple approach which combines all transactions into a single larger transaction against the nested transaction approach, we conclude that both approaches offer ACID guarantees among the nesting of transactions, but the nested transaction approach is more performant, user-friendly, and offers better code quality. We then provide detail about the implementation of nested single-partition transaction and nested multi-partition transaction, as well as how to deal with multiple users, triggered transactions, and system failures. Finally, we discuss several optimization strategies that would increase the performance of nested transaction, and suggest some possible future directions to explore.

References

[1] J. Meehan et al.
S-Store: Streaming Meets Transaction Processing. arXiv:1503.01143.

[2] D. Abadi et al.
Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal,* 12(2), 2003.

[3] D. Abadi et al.
The Design of the Borealis Stream Processing Engine. In *CIDR,* 2005.

[4] R. Kallman et al.
H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):14961499, 2008.

[5] E. Muller et al.
A Nested Transaction Mechanism for LOCUS. Proceedings of ACM-SIGOPS Conference, Bretton Woods, NH, October 1983.

[6] E. Boertjes et al.
An Architecture for Nested Transaction Support on Standard Database Systems. Proceedings of 9th international conference on Database and Expert System Applications., 1998; pp. 448-459