Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log

Philip A. Bernstein
Microsoft Research
Redmond, WA, USA
phil.bernstein@microsoft.com

Sudipto Das
Microsoft Research
Redmond, WA, USA
sudipto.das@microsoft.com

Abstract

In optimistic concurrency control, a certifier algorithm processes a log of transaction operations to determine whether each transaction satisfies a given isolation level and therefore should commit or abort. This logging and certification of transactions is often sequential and can become a bottleneck. To improve transaction throughput, it is beneficial to parallelize or scale out the certifier and the log. One common technique for such parallelization is to partition the database. If the database is perfectly partitioned such that transactions only access data from a single partition, then both the log and the certifier can be parallelized such that each partition has its own independent log and certifier. However, for many applications, partitioning is only approximate, i.e., a transaction can access multiple partitions. Parallelization using such approximate partitioning requires synchronization between the certifiers and logs to ensure correctness. In this paper, we present the design of a parallel certifier and a partitioned log that uses minimal synchronization to obtain the benefits of parallelization using approximate partitioning. Our parallel certifier algorithm dynamically assigns constraints to each certifier. Certifiers enforce constraints using only atomic writes and reads on shared variables, thus avoiding expensive synchronization primitives such as locks. Our partitioned log uses a lightweight causal messaging protocol to ensure that transactions accessing the same partition appear in the same relative order in all logs where they both appear. We describe the techniques applied to an abstract certifier algorithm and log protocol, making them applicable to a variety of systems. We also show how both techniques can be used in Hyder, a scale-out log-structured indexed record manager.

1 Introduction

Optimistic concurrency control (OCC) is a technique to analyze transactions that access shared data to determine which transactions commit or abort [14]. Instead of delaying certain operations that might lead to an incorrect execution, OCC allows a transaction to execute its operations as soon as it issues them. After the transaction finishes, OCC determines whether the transaction commits or aborts. A certifier is the component that makes this determination. It is a sequential algorithm that analyzes descriptions of the transaction one-by-one in a given total order. Each transaction description, called an intention, is a record that describes the operations that the
transaction performed on shared data, such as read and write. One way to determine the total order of intentions is to store them in a log. In that case, the certifier analyzes intentions in the order they appear in the log.

A certifier algorithm has throughput limits imposed by the underlying hardware [7]. This limits the scalability of a system that uses it. To improve the throughput, it is worthwhile to parallelize the algorithm. One way to do this is to split the set of transactions into partitions such that for every pair of transactions from different partitions, there are no conflicts between them. Then the certifier can run independently on each partition. However, it is often infeasible to partition transactions in this way. In that case, the certifier algorithm needs to handle transactions that span more than one partition. This paper presents such an algorithm.

The log also has throughput limits imposed by the hardware. Thus, a second opportunity for improving throughput is to partition the log, such that each partition includes updates that apply to a distinct database partition. This enables the log to be distributed over independent storage devices to provide higher aggregate throughput of read and append operations to the log. However, if the partitioning is imperfect, some transactions need to appear in two or more partitions. In this case, the log partitioning must ensure that conflicting transactions appear in the same relative order in all logs where they both appear. This paper presents a way of generating a log partitioning that satisfies this property.

The goal of these two techniques—parallelizing a certifier and partitioning a log—is to increase transaction throughput. Our motivation for designing these techniques is to increase the throughput of our Hyder system, a database architecture that scales out without partitioning [8]. In Hyder, the log is the database, which is represented as a multi-version binary search tree. Each transaction $T$ executes on a snapshot of the database and generates an intention record that contains $T$’s writset and, depending on the isolation level, its readset. The intention is stored in the log. A certification algorithm, called meld [9], reads intentions from the log and sequentially processes them in log order to determine whether a transaction committed or aborted. If a transaction commits, meld does one more step beyond OCC certification, namely, it merges the transaction’s updates into the server’s locally-cached copy of the database. Since all servers receive the same log, meld makes the same commit and abort decisions for every transaction. Therefore, for any two servers, their locally-cached copies of the database are identical for any data that is stored in both of them. Since there is no synchronization between the servers apart from appending to and reading from the shared log, the system scales out. That is, throughput increases as more servers are added, until the log, network, or meld algorithm is saturated. Often, the meld algorithm is the bottleneck. This was demonstrated in [6] by experiments with a distributed implementation of Hyder on a cluster of enterprise-grade commodity servers. It is therefore important to parallelize meld to increase transaction throughput. Bernstein et al. [6] describes two approaches that use pipeline parallelism to speed up meld; it introduces two preliminary stages that reduce the work done by the final sequential meld algorithm. In this paper, we leverage database partitioning to parallelize the meld algorithm itself.

Organization: We formally define the problem in Section 2 and then present the algorithms for parallel certification (Section 3) and log partitioning (Section 4). In Section 5, we revisit the question of how to apply these parallel solutions to Hyder. Section 6 summarizes related work and Section 7 is the conclusion.

2 Problem Definition

The certifier’s analysis relies on the notion of conflicting operations. Two operations conflict if the relative order in which they execute affects the value of a shared data item or the value returned by one of them. The most common examples of conflicting operations are read and write, where a write operation on a data item conflicts with a read or write operation on the same data item. Two transactions conflict if one transaction has an operation that conflicts with at least one operation of the other transaction.

To determine whether a transaction $T$ commits or aborts, a certifier analyzes whether any of $T$’s operations conflict with operations issued by other concurrent transactions that it previously analyzed. For example, if two transactions executed concurrently and have conflicting accesses to the same data, such as independent
writes of a data item $x$ or concurrent reads and writes of $x$, then the algorithm might conclude that one of the transactions must abort. Different certifiers use different rules to reach their decision. However, all certifiers have one property in common: their decision depends in part on the relative order of conflicting transactions.

We define a database partitioning to be a set of partition names, such as $\{P_1, P_2, \ldots\}$, and an assignment of every data item in the database to one of the partitions. A database partitioning is perfect with respect to a set of transactions $T = \{T_1, T_2, \ldots\}$ if every transaction in $T$ reads and writes data in at most one partition. That is, the database partitioning induces a transaction partitioning. If a database is perfectly partitioned, then it is trivial to parallelize the certifier and partition the log: For each partition $P_i$, create a separate log $L_i$ and an independent execution $C_i$ of the certifier algorithm. All transactions that access $P_i$ append their intentions to $L_i$, and $C_i$ takes $L_i$ as its input. Since transactions in different logs do not conflict, there is no need for shared data or synchronization between the logs or between executions of the certifier on different partitions.

A perfect partitioning is not possible in many practical situations, so this simple parallelization approach is not robust. Instead, suppose we can define a database partitioning that is approximate with respect to a set of transactions $T$, meaning that most transactions in $T$ read and write data in at most one partition. That is, some transactions in $T$ access data in two or more partitions (so the partitioning is not perfect), but most do not.

In an approximate partitioning, the transactions that access only one partition can be processed in the same way as a perfect partitioning. However, transactions that access two or more partitions make it problematic to partition the certifier. The problem is that such multi-partition transactions might conflict with transactions that are being analyzed by different executions of the certifier algorithm, which creates dependencies between these executions. For example, suppose data items $x$ and $y$ are assigned to different partitions $P_1$ and $P_2$, and suppose transaction $T_i$ writes $x$ and $y$. Then $T_i$ must be evaluated by $C_1$ to determine whether it conflicts with concurrent transactions that accessed $x$ and by $C_2$ to determine whether it conflicts with concurrent transactions that accessed $y$. These evaluations are not independent. For example, if $C_1$ determines that $T_i$ must abort, then that information is needed by $C_2$, since $C_2$ no longer has the option to commit $T_i$. When multiple transactions access different combinations of partitions, such scenarios can become quite complex.

A transaction that accesses two or more partitions also makes it problematic to partition the log, because its intentions need to be ordered in the logs relative to all conflicting transactions. Continuing with the example of transaction $T_j$ above, should its intention be logged on $L_1$, $L_2$, or some other log? Wherever it is logged, it must be ordered relative to all other transactions that have conflicting accesses to $x$ and $y$ before it is fed to the OCC algorithm. The problem we address is how to parallelize the certifier and partition the log relative to an approximate database partitioning. Our solution takes an approximate database partitioning, an OCC algorithm, and an algorithm to atomically append entries to the log as input. It has three components:

1. Given an approximate database partitioning $P = \{P_1, P_2, \ldots, P_n\}$, we define an additional logical partition $P_0$. Each transaction that accesses only one partition is assigned to the partition that it accesses. Each transaction that accesses two or more partitions is assigned to the master logical partition $P_0$.

2. We parallelize the certifier algorithm into $n + 1$ parallel executions $\{C_0, C_1, C_2, \ldots, C_n\}$, one for each partition, including the logical partition. Each single-partition transaction is processed by the certifier execution assigned to its partition. Each multi-partition transaction is processed by the logical partition’s execution of the certifier algorithm. We define synchronization constraints between the logical partition’s certifier execution and the partition-specific certifier executions so they reach consistent decisions.

3. We partition the log into $n + 1$ distinct logs $\{L_0, L_1, L_2, \ldots, L_n\}$, one associated with each partition and one associated with the logical partition. We show how to synchronize the logs so that the set of all intentions across all logs is partially ordered and every pair of conflicting transactions appears in the same relative order in all logs where they both appear. Our solution is a low-overhead sequencing scheme based on vector clocks.

Our solution works with any approximate database partitioning. Since multi-partition transactions are more
expensive than single-partition transactions, the fewer multi-partition transactions that are induced by the database partitioning, the better. The synchronization performed between parallel executions of the certifier algorithm is external to the certifier algorithm. Therefore, our solution works with any certifier algorithm. The same is true for the synchronization performed between parallel logs.

3 Parallel Certification

We now explain the design of a parallel certifier assuming a single totally-ordered log. In this section, we use the term certifier to refer to a certifier execution. A certifier can be parallelized using multiple threads within a single process, multiple processes co-located on the same machine, or multiple processes distributed across different machines; our discussion encompasses all such scenarios. Section 4, entitled “Partitioned Log,” explains how the parallel certification of this section can use a partitioned log.

3.1 Design

We dedicate one certifier $C_i$ to process intentions from single-partition transactions on partition $P_i$, and dedicate one certifier $C_0$ to process intentions from multi-partition transactions. A single scheduler $S$ processes intentions in log order, assigning each intention to one of the certifiers. The certifiers can process non-conflicting intentions in parallel. However, they must process conflicting intentions in log order.

Our design uses constraints that capture the log order. $S$ passes these constraints to each $C_i$. The certifiers validate the constraints using atomic reads and writes on shared variables, so the synchronization is efficient. Figure 1 illustrates the design of a parallel certifier showing the different variables and data structures maintained by each $C_i$, and the data structures used by $S$ to determine synchronization constraints passed to each $C_i$.

In what follows, for succinctness we frequently use the word “transaction” to mean the intention produced by the transaction. Each intention in the log has a unique location, called its log sequence number, or LSN, which reflects the relative order of intentions in the log. That is, intention $Int_i$ precedes intention $Int_k$ in the log if and only if the LSN of $Int_i$ is less than the LSN of $Int_k$.

Every certifier $C_i (\forall i \in [0, n])$ maintains a variable $\text{LastProcessedLSN}(C_i)$ that stores the LSN of the last transaction processed by $C_i$. After $C_i$ processes a transaction $T_k$, it sets $\text{LastProcessedLSN}(C_i)$ equal to $T_k$’s LSN; $C_i$ performs this update irrespective of whether $T_k$ committed or aborted. Every other certifier $C_j (\forall j \neq i)$ can atomically read $\text{LastProcessedLSN}(C_i)$ but cannot update it. In our algorithm, each $\text{LastProcessedLSN}(C_i), i \in [1, n]$, is read only by $C_0$ and $\text{LastProcessedLSN}(C_0)$ is read by all $C_i, i \in [1, n]$. Each $C_i (i \in [0, n])$ also has an associated producer-consumer queue $Q_i$ where $S$ enqueues the transactions $C_i$ needs to process (i.e., $S$ is the producer for $Q_i$). Each $C_i$ dequeues the next transaction from $Q_i$ when it completes processing its previous transaction (i.e., $C_i$ is the consumer for $Q_i$). The scheduler $S$ maintains a local structure, $\text{LastAssignedLSNMap}$, that maps each $C_i, i \in [1, n]$, to the LSN of the last single-partition transaction it assigned to $C_i$. $S$
maintains another local structure, \texttt{LastLSNAssignedToC_0.Map}, that stores a map of each partition \(P_i\) to the LSN of the last multi-partition transaction that it assigned to \(C_0\) and that accessed \(P_i\).

Each certifier \(C_i\) needs to behave as if it were processing all single-partition and multi-partition transactions that access \(P_i\) in log order. This requires that certifiers satisfy the following synchronization constraint:

**Parallel Certification Constraint:** Before certifying a transaction \(T\) that accessed partition \(P_i\), all transactions that precede \(T\) in the log and accessed \(P_i\) must have been certified.

This condition is trivially satisfied by a sequential certifier. Threads in a parallel certifier must synchronize to ensure that the condition holds. For each transaction \(T\), \(S\) determines which certifiers \(C_i\) will process \(T\). \(S\) uses its two local data structures, \texttt{LastAssignedLSNMap} and \texttt{LastLSNAssignedToC_0.Map}, to determine and provide each such \(C_i\) with the synchronization constraints it must satisfy before \(C_i\) can process \(T\). Note that this constraint is conservative since this strict ordering is essential only for conflicting transactions. However, in the absence of finer-grained tracking of conflicts, this conservative constraint guarantees correctness.

### 3.2 Synchronizing the Certifier Threads

Let \(T_i\) denote the transaction that \(S\) is currently processing. We now describe how \(S\) generates the synchronization constraints for \(T_i\). Once \(S\) determines the constraints, \(S\) enqueues the transaction and the constraints to the queue corresponding to the certifier.

**Single-partition transactions:** If \(T_i\) accessed a single partition \(P_i\), then \(T_i\) is assigned to the single-partition certifier \(C_i\). \(C_i\) must synchronize with \(C_0\) before processing \(T_i\) to ensure that the parallel certification constraint is satisfied. Let \(T_i\) be the last transaction that \(S\) assigned to \(C_0\), that is, \texttt{LastLSNAssignedToC_0.Map}(\(P_i\)) = \(k\). \(S\) passes the synchronization constraint \texttt{LastProcessedLSN}(\(C_0\)) \(\geq\) \(k\) to \(C_i\) along with \(T_i\). The constraint tells \(C_i\) that it can process \(T_i\) only after \(C_0\) has finished processing \(T_k\). When \(C_i\) starts processing \(T_i\’s\) intention, it accesses the variable \texttt{LastProcessedLSN}(\(C_0\)). If the constraint is satisfied, \(C_i\) can start processing \(T_i\). If the constraint is not satisfied, then \(C_i\) either polls the variable \texttt{LastProcessedLSN}(\(C_0\)) until the constraint is satisfied or uses an event mechanism to be notified when \texttt{LastProcessedLSN}(\(C_0\)) \(\geq\) \(k\).

**Multi-partition transactions:** If \(T_i\) accessed multiple partitions \(\{P_{i1}, P_{i2}, \ldots\}\), then \(S\) assigns \(T_i\) to \(C_0\). \(C_0\) must synchronize with the certifiers \(\{C_{i1}, C_{i2}, \ldots\}\) of all partitions \(\{P_{i1}, P_{i2}, \ldots\}\) accessed by \(T_i\). Let \(T_{k_j}\) be the last transaction assigned to \(P_j\) \(\in\) \(\{P_{i1}, P_{i2}, \ldots\}\), that is, \texttt{LastAssignedLSNMap}(\(C_j\)) = \(k_j\). \(S\) passes the following synchronization constraint to \(C_0\):

\[
\bigwedge_{j: P_j \in \{P_{i1}, P_{i2}, \ldots\}} \text{LastProcessedLSN}(C_j) \geq k_j.
\]

The constraint tells \(C_0\) that it can process \(T_i\) only after all \(C_j\) in \(\{C_{i1}, C_{i2}, \ldots\}\) have finished processing their corresponding \(T_{k_j}\’s\), which are the last transactions that precede \(T_i\) and accessed a partition that \(T_i\) accessed. When \(C_0\) starts processing \(T_i\’s\) intention, it reads the variables \texttt{LastProcessedLSN}(\(C_j\)) \(\forall j : P_j \in \{P_{i1}, P_{i2}, \ldots\}\). If the constraint is satisfied, \(C_0\) can start processing \(T_i\). Otherwise, \(C_0\) either polls the variables \texttt{LastProcessedLSN}(\(C_j\)) \(\forall j : P_j \in \{P_{i1}, P_{i2}, \ldots\}\) until the constraint is satisfied or uses an event mechanism to be notified when the constraint is satisfied.

Notice that for all \(j\) such that \(P_j \in \{P_{i1}, P_{i2}, \ldots\}\), the value of the variable \texttt{LastProcessedLSN}(\(C_j\)) increases monotonically over time. Thus, once the constraint \texttt{LastProcessedLSN}(\(C_j\)) \(\geq\) \(k_j\) becomes true, it will be true forever. Therefore, \(C_0\) can read each variable \texttt{LastProcessedLSN}(\(C_j\)) independently, with no synchronization. For example, it does not need to read all of the variables \texttt{LastProcessedLSN}(\(C_j\)) within a critical section.
3.3 An Example

Consider a database with three partitions $P_1, P_2, P_3$. Let $C_1, C_2, C_3$ be the parallel certifiers assigned to $P_1, P_2, P_3$ respectively, and let $C_0$ be the certifier responsible for multi-partition transactions. In this example, we consider the following sequence of transactions:

$$T_1^{[P_2]}, T_2^{[P_2]}, T_3^{[P_2]}, T_4^{[P_2]}, T_5^{[P_2]}, T_6^{[P_2]}, T_7^{[P_2]}, T_8^{[P_2]}, T_9^{[P_2]}$$

A transaction is represented in the form $T_i^{[P_j]}$ where $i$ is the transaction’s unique identifier and $[P_j]$ is the set of partitions that $T_i$ accesses. In this example, we use the transaction’s identifier $i$ also as its LSN. That is, we assume $T_1$ appears in position 1 in the log, $T_2$ in position 2, and so on.

$S$ processes the transactions (i.e., intentions) in log order. For each transaction, it determines which certifiers will process the intention and determines the synchronization constraint it needs to pass to the certifiers to enforce the parallel certification constraint. The sequence of figures 2–8 illustrate the parallel certifier in action while it is processing the above sequence of transactions, showing how the certifiers synchronize. In each figure, we emphasize the transaction(s) at the tail of the log being processed by $S$; time progresses from top to bottom. The LastProcessedLSN at the top of the figure shows the variable’s value for each certifier before it has started processing the recently-arrived transactions, i.e., the values after processing the transactions from the previous figure in the sequence. The vertical arrows beside each vertical line shows the processing time of each intention at a certifier. The values updated as a result of processing an intention are highlighted in red. To avoid cluttering the figure, we show minimal information about the previous transactions.

Figure 2 shows a single-partition transaction $T_1$ accessing $P_2$. The numbers (1)–(6) identify points in the execution. At (1), $S$ determines the synchronization constraint it must pass to $C_2$, namely, that $C_0$ must have at least finished processing the last multi-partition transaction that accessed $P_2$. $S$ reads this value in LastLSNAssignedToC0Map($P_2$). Since $S$ has not processed any multi-partition transaction before $T_1$, the constraint is LastProcessedLSN($C_0$) $\geq 0$. At (1), $S$ updates LastAssignedLSNMap($C_2$) = 1 to reflect its assignment of $T_1$ to $C_2$. At (2), $S$ assigns $T_1$ to $C_2$, and then moves to the next transaction in the log. At (3), $C_2$ reads LastProcessedLSN($C_0$) as 0 and hence determines at (4) that the constraint is satisfied. Therefore, at (5) $C_2$ starts processing $T_1$. After $C_2$ completes processing $T_1$, at (6) it updates LastProcessedLSN($C_2$) to 1.

Figure 3 shows the processing of the next three single-partition transactions—$T_2, T_3, T_4$—using steps similar to those in Figure 2. As shown in Figure 4, whenever possible, the certifiers process the transactions in parallel. In the state shown in Figure 3, at (2) $C_1$ is still processing $T_2$, at (3) $C_2$ completed processing $T_3$ and updated its variable LastProcessedLSN($C_2$) to 3, and at (4) $C_3$ completed processing $T_4$ and updated its variable LastProcessedLSN($C_3$) to 4.

Figure 4 shows the processing of the first multi-partition transaction, $T_5$, which accesses partitions $P_1$ and $P_2$. $S$ assigns $T_5$ to $C_0$. At (5), $S$ specifies the required synchronization constraint, which ensures that
Figure 3: $S$ processes transactions in log order and updates its local structures. Each certifier processes the transactions that $S$ assigns to it.

Figure 4: For multi-partition transactions, $S$ determines the synchronization constraints and assigns the transaction to $C_0$.

$T_5$ is processed after $T_2$ (the last single-partition transaction accessing $P_1$) and $T_3$ (the last single-partition transaction accessing $P_2$). $S$ reads LastAssignedLSNMap($P_1$) and LastAssignedLSNMap($P_2$) to determine the LSNs of the last single-partition transactions for $P_1$ and $P_2$, respectively. The synchronization constraint shown at (1) corresponds to this requirement, i.e., $\text{LastProcessedLSN}(C_1) \geq 2$ and $\text{LastProcessedLSN}(C_2) \geq 3$. $S$ passes the constraint to $C_0$ along with $T_5$. Then, at (1), $S$ updates $\text{LastLSNAssignedTo}C_0\text{Map}(P_i) = 5$ and $\text{LastLSNAssignedTo}C_0\text{Map}(P_j) = 5$ to reflect that $T_5$ is the last multi-partition transaction accessing $P_1$ and $P_2$. Any subsequent single-partition transaction accessing $P_1$ or $P_2$ must now follow the processing of $T_5$. At (2) and (3) $C_0$ reads $\text{LastProcessedLSN}(C_2)$ and $\text{LastProcessedLSN}(C_1)$ respectively to evaluate the constraint. At this point in time, $C_1$ is still processing $T_2$ and hence at (4) the constraint evaluates to false. Therefore, even though $C_2$ has finished processing $T_3$, $C_0$ waits for $C_1$ to finish processing $T_2$. This occurs at (5), where it updates $\text{LastProcessedLSN}(C_1)$ to 2. Now, at (5) $C_1$ notifies $C_0$ about this update. So $C_0$ checks its constraint again and sees that it is satisfied. Therefore, at (7) it starts processing $T_5$.

Figure 5 shows processing of the next transaction $T_6$, a single-partition transaction that accesses $P_2$. Since both $T_5$ and $T_6$ access $P_2$, $C_2$ can process $T_6$ only after $C_0$ has finished processing $T_5$. Similar to other single-partition transactions, $S$ constructs this constraint by looking up $\text{LastLSNAssignedTo}C_0\text{Map}(P_2)$ which is 5. Therefore, at (1) $S$ passes the constraint $\text{LastProcessedLSN}(C_0) \geq 5$ to $C_2$ along with $T_6$, and at (1) sets $\text{LastLSNAssignedTo}C_0\text{Map}(P_2) = 6$. At (2) $C_2$ reads $\text{LastProcessedLSN}(C_0) = 0$. So its evaluation of the constraint at (3) yields false. $C_0$ finishes processing $T_5$ at (4) and sets $\text{LastProcessedLSN}(C_0) = 5$. At (5), $C_0$ notifies $C_2$ that it updated $\text{LastProcessedLSN}(C_0)$, so $C_2$ checks the constraint again and finds it true. Therefore, at (6) it starts processing $T_6$.

While $C_2$ is waiting for $C_0$, other certifiers can process subsequent transactions if the constraints allow
Figure 5: Synchronization constraints to order single-partition transactions after a multi-partition transaction.

Figure 6: Benefits of parallelization for single-partition transactions. \(C_3\) can start processing \(T_7\) while \(T_6\) is waiting for \(T_5\) to complete on \(C_0\).

it. Figure 6 illustrates this scenario where the next transaction in the log, \(T_7\), is a single-partition transaction accessing \(P_3\). Since no multi-partition transaction preceding \(T_7\) has accessed \(P_3\), at (0) the constraint passed to \(C_3\) is LastProcessedLSN(\(C_0\)) \(\geq 0\). The constraint is trivially satisfied, which \(C_3\) observes at (3). Therefore, while \(C_2\) is waiting, at (4) \(C_3\) starts processing \(T_7\) in parallel with \(C_0\)'s processing of \(T_5\) and \(C_2\)'s processing of \(T_6\), thus demonstrating the benefit of parallelizing the certifiers.

Figure 7 illustrates that if the synchronization constraints allow, even a multi-partition transaction can be processed in parallel with other single-partition transactions without any waits. Transaction \(T_8\) accesses \(P_1\) and \(P_3\). At (0), based on LastAssignedLSNMap, \(S\) generates a constraint of LastProcessedLSN(\(C_1\)) \(\geq 2\) \& LastProcessedLSN(\(C_3\)) \(\geq 7\) and passes it along with \(T_8\) to \(C_0\). By the time \(C_0\) starts evaluating its constraint, both \(C_1\) and \(C_3\) have completed processing the transactions of interest to \(C_0\). Therefore, at (2) and (3) \(C_0\) reads LastProcessedLSN(\(C_1\)) = 2 and LastProcessedLSN(\(C_3\)) = 7. So at (4) \(C_0\) finds that the constraint LastProcessedLSN(\(C_1\)) \(\geq 2\) \& LastProcessedLSN(\(C_3\)) \(\geq 7\) is satisfied. Thus, it can immediately start processing \(T_8\) at (5), even though \(C_2\) is still processing \(T_6\). This is another example demonstrating the benefits of parallelism.

As shown in Figure 8, \(S\) processes the next transaction, \(T_9\), which accesses only one partition, \(P_2\). Although \(T_8\) is still active at \(C_0\) and hence blocking further activity on \(C_1\) and \(C_3\), by this time \(T_7\) has finished running at \(C_2\). Therefore, when \(S\) assigns \(T_9\) to \(C_2\) at (0), \(C_2\)'s constraint is already satisfied at (3), so \(C_2\) can immediately start processing \(T_9\) at (4), in parallel with \(C_0\)'s processing of \(T_8\). Later, \(T_8\) finishes at (5) and \(T_9\) finishes at (6), thereby completing the execution.
3.4 Discussion

Correctness requires that for each partition $P_i$, all transactions that access $P_i$ are certified in log order. There are two cases, single-partition and multi-partition transactions.

- The constraint on a single-partition transaction $T_i$ ensures that $T_i$ is certified after all multi-partition transactions that precede it in the log and that accessed $P_i$. Synchronization conditions on multi-partition transactions ensure that $T_i$ is certified before all multi-partition transactions that follow it in the log and that accessed $P_i$.

- The constraint on a multi-partition transaction $T_i$ ensures that $T_i$ is certified after all single-partition transactions that precede it in the log and that accessed partitions $\{P_{i_1}, P_{i_2}, \ldots\}$ that $T_i$ accessed. Synchronization conditions on single-partition transactions ensure that for each $P_j \in \{P_{i_1}, P_{i_2}, \ldots\}$, $T_i$ is certified before all single-partition transactions that follow it in the log and that accessed $P_j$.

Note that transactions that modify a given partition $P_i$ will be certified by $C_i$ or $C_0$ (but not both), depending on whether it is single-partition or multi-partition.

The extent of parallelism achieved by the proposed parallel certifier depends on designing a partitioning that ensures most transactions access a single partition and that spreads transaction workload uniformly across the partitions. With a perfect partitioning, each certifier can have a dedicated core. So with $n$ partitions, a parallel certifier will run up to $n$ times faster than a single sequential certifier.
Each of the variables that is used in a synchronization constraint—LastAssignedLSNMap, LastProcessedLSN, and LastLSNAssignedToC₀Map—is updatable by only one certifier. Therefore, there are no race conditions on these variables that require synchronization between certifiers. The only synchronization points are the constraints on individual certifiers which can be validated with atomic read operations.

3.5 Finer-Grained Conflict Testing

The parallelized certifier algorithm generates constraints under the assumption that certification of two transactions that access the same partition must be synchronized. This is a conservative assumption, in that two transactions that access the same partition might access the same data item in non-conflicting modes, or might access different data items in the partition, which implies the transactions do not conflict. Therefore, the synchronization overhead can be improved by finer-grained conflict testing. For example, in LastAssignedLSNMap, instead of storing one value for each partition that identifies the LSN of the transaction assigned to the partition, it could store two values: the LSN of the last transaction that read the partition and was assigned to the partition and the LSN of the last transaction that wrote the partition and was assigned to the partition. A similar distinction could be made for the other variables. Then, S could generate a constraint that would avoid requiring that a multi-partition transaction that only read partition \( P_i \) be delayed by an earlier single-partition transaction that only read partition \( P_i \), and vice versa. Of course, the constraint would still need to ensure that a transaction that wrote \( P_i \) is delayed by earlier transactions that read or wrote \( P_i \), and vice versa.

This finer-grained conflict testing would not completely do away with synchronization between \( C_0 \) and \( C_i \), even when a synchronization constraint is immediately satisfied. Synchronization would still be needed to ensure that only one of \( C_0 \) and \( C_i \) is active on a partition \( P_i \) at any given time, since conflict-testing within a partition is single-threaded. Aside from that synchronization, and the use of finer-grained constraints, the rest of the algorithm for parallelizing certification remains the same.

4 Partitioned Log

Partitioning the database also allows partitioning the log, provided ordering constraints between intentions in different logs are preserved. The log protocol is executed by each server that processes transactions. Alternatively, it could be embodied in a log server, which receives requests to append intentions from servers that run transactions.

4.1 Design

In our design, there is one log \( L_i \) dedicated to every partition \( P_i (\forall i \in [1, n]) \), which stores intentions for single-partition transactions accessing \( P_i \). There is also a log \( L_0 \), which stores the intentions of multi-partition transactions. If a transaction \( T_i \) accesses only \( P_i \), its intention is appended to \( L_i \) without communicating with any other log. If \( T_i \) accessed multiple partitions \( \{ P_i \} \), its intention is appended to \( L_0 \) followed by communication with all logs \( \{ L_i \} \) corresponding to \( \{ P_i \} \). The log protocol must ensure the following constraint for correctness:

**Partitioned Log Constraint:** There is a total order between transactions accessing the same partitions, which is preserved in all logs where both transactions appear.

Figure 9 provides an overview of the log sequence numbers used in the partitioned log design. A technique similar to vector clocks is used for sequence-number generation [11, 17]. Each log \( L_i \) for \( i \in [1, n] \) maintains the single-partition LSN of \( L_i \), denoted SP-LSN(\( L_i \)), which is the LSN of the last single-partition log record appended to \( L_i \). To order single-partition transactions with respect to multi-partition transactions, every log also maintains the multi-partition LSN of \( L_i \), denoted MP-LSN(\( L_i \)), which is the LSN of the last multi-partition
transaction that accessed $P_i$ and is known to $L_i$. The sequence number of each record $R_k$ in log $L_i$ for $i \in [1, n]$ is expressed as a pair of the form $[\text{MP-LSN}(L_i), \text{SP-LSN}(L_i)]$ which identifies the last multi-partition and single-partition log records that were appended to $L_i$, including $R_k$ itself. The sequence number of each record $R_k$ in log $L_0$ is of the form $[\text{MP-LSN}(L_0), 0]$, i.e., the second position is always zero. All logs start with sequence number $[0, 0]$.

The order of two sequence numbers is decided by first comparing MP-LSN($i$) and then SP-LSN($i$). That is, $\text{MP-LSN}_m(L_i) \prec \text{MP-LSN}_m(L_j)$ if either $\text{MP-LSN}_m(L_i) < \text{MP-LSN}_m(L_j)$, or $\text{MP-LSN}_m(L_i) = \text{MP-LSN}_m(L_j)$ and $\text{SP-LSN}_m(L_i) < \text{SP-LSN}_m(L_j)$. This technique totally orders intentions in the same log (i.e., if $i = j$), while partially ordering intentions of two different logs (i.e., if $i \neq j$). If the ordering between two intentions is not defined, then they are treated as concurrent.

Notice that LSNs in different logs are incomparable, because their SP-LSN’s are independently assigned. The assignment of sequence numbers is explained in the description of the log protocol.

### 4.2 Log Protocol

**Single-partition transactions**: Given transaction $T_i$, if $T_i$ accessed a single partition $P_i$, then $T_i$’s intention is appended only to $L_i$. SP-LSN($L_i$) is incremented and the LSN of $T_i$’s intention is set to $[\text{mp-lsn}, \text{SP-LSN}(L_i)]$, where mp-lsn is the latest value of MP-LSN($L_0$) that $L_i$ has received from $L_0$.

**Multi-partition transactions**: If $T_i$ accessed multiple partitions $\{P_{i1}, P_{i2}, \ldots\}$, then $T_i$’s intention is appended to log $L_0$ and the multi-partition LSN of $L_0$, MP-LSN($L_0$), is incremented. After these actions finish, MP-LSN($L_0$) is sent to all logs $\{L_{i1}, L_{i2}, \ldots\}$ corresponding to $\{P_{i1}, P_{i2}, \ldots\}$, which completes $T_i$’s append.

This approach of log-sequencing enforces a causal order between the log entries. That is, two log entries have a defined order only if they accessed the same partition.

Each log $L_i(\forall i \in [1, n])$ maintains MP-LSN($L_i$) as the largest value of MP-LSN($L_0$) it has received from $L_0$ so far. However, each $L_i$ does not need to store its MP-LSN($L_i$) persistently. If $L_i$ fails and then recovers, it can obtain the latest value of MP-LSN($L_0$) by examining $L_0$’s tail. It is tempting to think that this examination of $L_0$’s tail can be avoided by having $L_i$ log each value of MP-LSN($L_0$) that it receives. While this does potentially enable $L_i$ to recover further without accessing $L_0$’s tail, it does not avoid that examination entirely. To see why, suppose the last transaction that accessed $P_i$ before $L_i$ failed was a multi-partition transaction that succeeded in appending its intention to $L_0$; but $L_i$ did not receive the MP-LSN($L_0$) for that transaction before $L_i$ failed. In that case, after $L_i$ recovers, it still needs to receive that value of MP-LSN($L_0$), which it can do only by examining $L_0$’s tail. If $L_0$ has also failed, then after recovery, $L_i$ can continue with its highest known value of MP-LSN($L_0$) without waiting for $L_0$ to recover. As a result, a multi-partition transaction might be ordered in $L_i$ at a later position than where it would have been ordered if the failure did not happen.

Alternatively, for each multi-partition transaction, $L_0$ could run two-phase commit with the logs corresponding to the partitions that the transaction accessed. That is, it could send MP-LSN($L_0$) to those logs and wait for
acknowledgments from all of them before logging the transaction at \( L_0 \). However, like any use of two-phase commit, this protocol has the possibility of blocking if a failure occurs between phases one and two.

To avoid this blocking, in our design, when \( L_0 \) recovers, it communicates with every \( L_i \) to pass the latest value of MP-LSN(\( L_0 \)). When one of the \( L_i \)’s recovers, it reads the tail of \( L_0 \). This recovery protocol ensures that MP-LSN(\( L_0 \)) propagates to all single-partition logs.

4.3 An Example

Let us assume that a database has three partitions \( P_1, P_2, P_3 \). Let \( L_1, L_2, L_3 \) be the logs assigned to \( P_1, P_2, P_3 \) respectively, and \( L_0 \) be the log for multi-partition transactions. Consider the following sequence of transactions:

\[
T_1^{[P_2]}, T_2^{[P_2]}, T_3^{[P_2]}, T_4^{[P_3]}, T_5^{[P_1, P_2]}, T_6^{[P_2]}, T_7^{[P_3]}, T_8^{[P_1, P_3]}, T_9^{[P_2]}
\]

As earlier, a transaction is represented in the form \( T_i^{[P_j]} \) where \( i \) is a unique transaction identifier; note that this identifier does not induce an ordering between the transactions. The superscript on \( T_i \) identifies the partitions that \( T_i \) accesses. We use \( T_i \) to refer to both a transaction and its intention. In figures 10–14, the vertical line at the extreme left shows the order in which the append requests arrive; time progresses from top to bottom. The LSN at the top of each figure shows each log’s LSN before it has appended the recently-arrived transactions, i.e., the values after processing the transactions from the previous figure in the sequence. The black circles on each vertical line for a log shows the append of the transaction and the updated values of the LSN. A multi-partition transaction is shown using a triangle and receipt of a new multi-partition LSN at the single partition logs is shown with the dashed triangle. The values updated as a result of processing an intention are highlighted in red.

Figure 10 shows four single-partition transactions \( T_1, T_2, T_3, T_4 \) that are appended to the logs corresponding to the partitions that the transactions accessed; the numbers (1)–(4) identify points in the execution. When appending a transaction, the log’s SP-LSN is incremented. For instance, in Figure 10, \( T_1 \) is appended to \( L_2 \) at (1) which changes \( L_2 \)’s LSN from \([0, 0]\) to \([0, 1]\). Similarly at (2)–(4), the intentions for \( T_2, T_3, T_4 \) are appended and the SP-LSN of the appropriate log is incremented. Appends of single-partition transactions do not need synchronization between the logs and can proceed in parallel; an order is induced only between transactions appended to the same log. For instance, \( T_1 \) and \( T_3 \) both access partition \( P_3 \) and hence are appended to \( L_2 \) with \( T_1 \) (at (1)) preceding \( T_3 \) (at (3)); however, the relative order of \( T_1, T_2, \) and \( T_4 \) is undefined.

Multi-partition transaction results in loose synchronization between the logs to induce an ordering among transactions appended to different logs. Figure 11 shows an example of a multi-partition transaction \( T_5 \) that accessed \( P_1 \) and \( P_2 \). When \( T_5 \)’s intention is appended to \( L_0 \) (at (1)), MP-LSN(\( L_0 \)) is incremented to 1. In step (2), the new value MP-LSN(\( L_0 \)) = 1 is sent to \( L_1 \) and \( L_2 \). On receipt of this new LSN (step (3)), \( L_1 \) and \( L_2 \)
Figure 11: A multi-partition transaction is appended to $L_0$ and MP-LSN($L_0$) is passed to the logs of the partitions accessed by the transaction.

Figure 12: Single-partition transactions that follow a multi-partition transaction persistently store the new value of MP-LSN($L_i$) in $L_i$.

update their corresponding MP-LSN, i.e., $L_1$’s LSN is updated to [1, 1] and $L_2$’s LSN is updated to [1, 2]. As an optimization, this updated LSN is not persistently stored in $L_1$ or $L_2$. If either log fails, this latest value can be obtained from $L_0$ that stores it persistently.

Any subsequent single-partition transaction appended to either $L_1$ or $L_2$ will be ordered after $T_5$, thus establishing a partial order with transactions appended to $L_0$. As shown in Figure 12, $T_6$ is a single-partition transaction accessing $P_2$ which when appended to $L_2$ (at ①) establishes the order $T_3 < T_5 < T_6$. As a side-effect of appending $T_6$’s intention, MP-LSN($L_2$) is persistently stored as well. $T_7$, another single-partition transaction accessing $P_3$, is appended to $L_3$ at ②. It is concurrent with all transactions except $T_4$, which was appended to $L_3$ before $T_7$.

Figure 13 shows the processing of another multi-partition transaction $T_8$ which accesses partitions $P_1$ and $P_3$. Similar to the steps shown in Figure 11, $T_8$ is appended to $L_0$ (at ①) and MP-LSN($L_0$) is updated. The new value of MP-LSN($L_0$) is passed to $L_1$ and $L_3$ (at ②) after which the logs update their corresponding MP-LSN (at ③). $T_8$ induces an order between multi-partition transactions appended to $L_0$ and subsequent transactions accessing $P_1$ and $P_3$. The partitioned log design continues processing transactions as described, establishing a partial order between transactions as and when needed. Figure 14 shows the append of the next single-partition transaction $T_9$ appended to $L_2$ (at ①).
4.4 Concurrent Appends to $L_0$

To ensure that multi-partition transactions have a consistent order across all logs, a new intention is appended to $L_0$ only after the previous append to $L_0$ has completed, i.e., the new value of MP-LSN($L_0$) has propagated to all single-partition logs corresponding to the partitions accessed by the transaction. This sequential appending of transactions to $L_0$ might increase the latency of multi-partition transactions. A simple extension can allow parallel appends to $L_0$ simply by requiring that each log partition retains only the largest MP-LSN($L_0$) that it has received so far. If a log $L_i$ receives values of MP-LSN($L_0$) out of order, it simply ignores the stale value that arrives late. For example, suppose a multi-partition transaction $T_i$ is appended to $L_0$ followed by another multi-partition transaction $T_j$, which have MP-LSN($L_0$) = 1 and MP-LSN($L_0$) = 2, respectively. Suppose log $L_i$ receives MP-LSN($L_0$) = 2 and later receives MP-LSN($L_0$) = 1. In this case, $L_i$ ignores the assignment MP-LSN($L_0$) = 1, since it is a late-arriving stale value.

4.5 Discussion

With a sequential certification algorithm, the logs can be merged by each compute server. A multi-partition transaction $T_i$ is sequenced immediately before the first single-partition transaction $T_j$ that accessed a partition that $T_i$ accessed and was appended with $T_i$’s MP-LSN($L_0$). To ensure all intentions are ordered, each LSN is augmented with a third component, which is its partition ID, so that two LSNs with the same multi-partition and single-partition LSN are ordered by their partition ID.
Figure 15: Partitioning a database in Hyder. Subfigure (a) shows partitions as independent trees. Subfigure (b) shows a single database tree divided into partitions with inter-partition links maintained lazily.

With the parallel certifier, the scheduler $S$ adds constraints when assigning intentions to the certifiers. Certifiers $C_i (i \in [1, \ldots, n])$ will process single-partition transactions appended to $L_i$ and $C_0$ will process multi-partition transactions appended to $L_0$. For $C_i (i \in [1, \ldots, n])$ processing a single-partition transaction with LSN $[\text{MP-LSN}_k(L_i), \text{SP-LSN}_k(L_i)]$, the certification constraint for $C_i$ is $\text{LastProcessedLSN}(C_i) \geq [\text{MP-LSN}_k(L_i), 0]$. This constraint ensures that the single-partition transaction is certified only after $C_0$ has certified the multi-partition transaction with $\text{MP-LSN}_k(L_i)$. For $C_0$ processing multi-partition transaction $T$ that accessed partitions $\{P_i\}$ and has LSN $[\text{MP-LSN}_k(L_0), 0]$, the scheduling constraint is $\wedge_{\{j \in P_i\}} \text{LastProcessedLSN}(C_j) \geq X_j$, where $X_j$ is the LSN of the last single-partition transaction accessing $P_j$ that appeared in $L_j$ before $T$. This constraint ensures that the multi-partition transaction $T$ is certified only after all single-partition transactions that are ordered before $T$ have been certified. These constraints can be deduced from the data structures that the scheduling thread $S$ maintains, as described in Section 3.1.

Consider for example the sequence of transactions in Section 4.3 and the LSNs assigned as shown in Figure 14. $T_6$ is a single partition transaction with LSN $[1, 3]$ ordered after multi-partition transaction $T_5$ with LSN $[1, 0]$. $T_5$’s position in $L_2$ is between $T_3$ and $T_6$. The constraint passed to $C_2$ which certifies $T_6$ is $\text{LastProcessedLSN}(C_0) \geq [1, 0]$. This constraint ensures that $C_2$ certifies $T_6$ only after $C_0$ has certified $T_5$. Now consider the certification of multi-partition transaction $T_8$ which accessed partitions $P_1$ and $P_3$. $C_0$’s constraint is $\text{LastProcessedLSN}(C_1) \geq [0, 1] \wedge \text{LastProcessedLSN}(C_3) \geq [0, 2]$. This ensures that $C_0$ certifies $T_8$ only after $C_1$ has certified $T_2$ and $C_3$ has certified $T_7$.

To argue about correctness, we need to show that the partitioned log behaves the same as a non-partitioned log. For sequential certification, the partitioned log is merged into a single non-partitioned log, so the result follows immediately. For parallel certification, for each log $L_i (i \neq 0)$, the constraints ensure that each multi-partition transaction is synchronized between $L_0$ and $L_i$ in exactly the same way as in the single-log case.

If most of the transactions access only a single partition and there is enough network capacity, this partitioned log design provides a nearly linear increase in log throughput as a function of the number of partitions. The performance impact of multi-partition transactions is not expected to be very high.

5 Partitioning in Hyder – An application scenario

As we explained in Section 1, Hyder is a system that uses OCC and a log-structured database that is shared by all servers. Given an approximate partitioning of the database, the parallel certification and partitioned log algorithms described in this paper can be directly applied to Hyder. Each parallel certifier would run Hyder’s OCC algorithm, called meld, and each log partition would be an ordinary Hyder log storing updates to that partition. Each log stores the after-image of the binary search tree created by transactions updating the corresponding partition. Multi-partition transactions result in a single intention record that stores the after-image of all partitions, though this multi-partition intention can be split so that a separate intention is created for every partition.
The application of approximate partitioning to Hyder assumes that the partitions are independent trees as shown in Figure 15(a). Directory information is maintained that describes which data is stored in each partition. During transaction execution, the executer tracks the partitions accessed by the transaction. This information is included in the transaction’s intention, which is used by the scheduler to parallelize certification and by the log partitioning algorithm.

In addition to the standard Hyder design where all compute nodes run transactions (on all partitions), it is possible for a given compute node to serve only a subset of the partitions. However, this increases the cost of multi-partition transaction execution and meld.

A design with a partitioned tree, as shown in Figure 15(b), is also possible, though at the cost of increased complexity. Cross-partition links are maintained as logical links, to allow single-partition transactions to proceed without synchronization and to minimize the synchronization required to maintain the database tree. For instance, in Figure 15(b), the link between partitions $P_1$ and $P_3$ is specified as a link from node $F$ to the root $K$ of $P_3$. Since single-partition transactions on $P_3$ modify $P_3$’s root, traversing this link from $F$ requires a lookup of the root of partition $P_3$. This link is updated during meld of a multi-partition transaction accessing $P_1$ and $P_3$ and results in adding an ephemeral node replacing $F$ if $F$’s left subtree was updated concurrently with the multi-partition transaction. The generation of ephemeral nodes is explained in [9].

6 Related Work

Optimistic concurrency control (OCC) was introduced by Kung and Robinson in [14]. Its benefits and tradeoffs have been extensively explored in [1, 2, 12, 16, 18, 20]. Many variations and applications of OCC have been published. For example, Tashkent uses a centralized OCC validator over distributed data [10]. An OCC algorithm for an in-memory database is described in [15]. None of these works discuss ways to partition the algorithm.

An early timestamp-based concurrency control algorithm that uses partitioning of data and transactions is described in [5]. More recent examples of systems that partition data to improve scalability are in [3, 13, 19, 21].

The only other partitioned OCC algorithm we know of is for the Tango system [4]. In Tango, after a server runs a multi-partition transaction $T$ and appends $T$’s log record, it rolls forward the log to determine $T$’s commit/abort decision and then writes that decision to the log. The certifier of each partition uses that logged decision to decide how to act on log records from multi-partition transactions. This enables the certifier to update its version state of data, so it can perform OCC validation of single-partition transactions. That is, each certifier $C_i$ reads the sequence of single-partition and multi-partition log records that read or updated $P_i$. When $C_i$ encounters a multi-partition log record, it waits until it sees a decision record for that transaction in the log. This synchronization point is essentially the same as that of $C_i$ waiting for $C_0$ in our approach. However, the mechanism is different in two ways: the synchronization information is passed through the log, rather than through shared variables; and every server that runs a multi-partition transaction also performs the log roll-forward to determine the transaction’s decision (although this could be done by a centralized server, like $C_0$). The experiments in [4] show good scalability with a moderate fraction of cross-partition transactions. It remains as future work to implement the algorithm proposed here and compare it to Tango’s.

In Tango, all partitions append log records to a single sequential log. Therefore, the partitioned log constraint is trivially enforced. By contrast, our design offers explicit synchronization between log records that access the same partition. This enables them to be written to different logs, which in aggregate can have higher bandwidth than a single log, like Tango’s.

Another approach to parallelizing meld is described in [6]. It uses a pipelined design that parallelizes meld onto multiple threads. One stage of the pipeline preprocesses each intention $I$ by testing for conflicts with committed transactions before the final meld step. It also “refreshes” $I$ by replacing stale data in $I$ by committed updates. The other stage combines adjacent intentions in the log, also before the final meld step. Each of these stages reduces the work required by the final meld step.
7 Concluding Remarks

In this paper, we explained a design to leverage approximate partitioning of a database to parallelize the certifier of an optimistic concurrency control algorithm and its accompanying log. The key idea is to dedicate a certifier and a log to each partition so that independent non-conflicting transactions accessing only a single partition can be processed in parallel while ensuring transactions accessing the same partition are processed in a sequence. Since partitioning of the database, and hence the transactions, need not be perfect, i.e., a transaction can access multiple partitions, our design processes these multi-partition transactions using a dedicated multi-partition certifier and log. The efficiency of the design stems from using lightweight synchronization mechanisms—the parallel certifiers synchronize using constraints while the partitioned log synchronizes using asynchronous causal messaging. The design abstracts out the details of the certifier and the logging protocol, making it applicable to a wide variety of systems. We also discussed the application of the design in Hyder, a scale-out log-structured transactional record manager. Our design allows Hyder to leverage approximate partitioning to further improve the system’s throughput.

References


