Yale University
Department of Computer Science

Scheduling Heuristics for Lazy Database Systems

Daniel Tahara
Yale University
daniel.tahara@yale.edu

YALEU/DCS/TR-1488
May 2014

Scheduling Heuristics for Lazy Database Systems

Daniel Tahara
Yale University
daniel.tahara@yale.edu

Abstract

Traditional relational database design specifies that transactions be executed immediately. This adheres to our intuition for how transactions should work—in order to issue a commit/abort decision, we must run the transaction logic to completion. However, there are certain classes of transactions that might benefit from deferred execution in which the commit/abort logic is executed immediately, but the non-contentious logic is executed later. We call a system adopting this approach to be a lazy database system.

Although there are a number of benefits to lazy database systems, including decreased contention through critical sections, purely demand-based lazy execution (i.e. not materializing a chain of transactions until there is an external read) suffers from very high latencies. However, if we selectively (and eagerly) materialize some transactions, we might be able to curb latency while still maintaining the contention benefits of lazy execution. This work focuses on developing such heuristics. We compare three different approaches, each of which places limits on the shape of the transaction dependency graph—in particular, depth, transitive closure, and number of parents. We show that some of these heuristics (with the correct parameter values) can outperform a purely lazy system implementation and others can offer a more tunable, incremental tradeoff between latency and throughput.

1 Introduction

A typical database workload comprises a series of reads and writes on often overlapping but possibly disjoint portions of data. In order to allow developers to make certain assumptions about their data, notably the ACID guarantees (atomicity, consistency, isolation, durability), most modern RDBMSs utilize techniques such as locking, two-phase commit protocols, and multiversion concurrency control. Together, these techniques ensure that, despite concurrent access to the data, one will never reach or remain in an intermediate state that can be considered somehow 'invalid.'

Regardless of the specific implementation, all RDBMSs rely on a transaction manager in order to effect the mechanisms necessary to ensure that transactions are executed and decide whether each given transaction has committed or needs to be aborted and rolled back. In particular, through locking and the other techniques described above, the transaction manager creates and then executes a determines a possibly interleaved execution plan that reflects some serial order of those transactions. It then, by immediately attempting to make the changes to the underlying physical store, can determine if the transactions violate any integrity or other constraints and abort the transaction if necessary. If not, the transaction succeeds and is 'committed'.

Until recently, the status quo has been that the transaction managers should execute transactions immediately. Such a perspective adheres to our intuition; in order to provide the correct commit/abort decision, the transaction must actually be executed to completion and the end state of the RDBMS verified. In typical, nondeterministic database system, this is the only method of execution because many situations such as deadlock or database crashes may cause a transaction to abort independently of the transaction logic. However, when we switch our attention to deterministic DBMSs such as [1] and [3], these nondetermistic failures are disallowed, so every transaction will be committed as long as it does not violate any integrity constraints.

Consider then, the following: rather than immediately executing transactions as they arrive in the database, what if the database delayed the execution to some later point in time as determined by some scheduling algorithm? Minus some formalisms, we have the beginnings of what we might call a lazy transaction manager, an idea which we explore in more detail below.

Define an *eager transaction manager* as one that, upon receiving a transaction (or set of potentially competing transactions), immediately creates an execution plan and applies the operations to the database. The issuers of the query are notified immediately of the result of their transactions, and the database state changes immediately.

In contrast, define a *lazy transaction manager* as one that only executes transactions when the specific data to which they refer are referenced by some other transaction; that is, when a read or write request is issued for that data. The transaction manager still provides an immediate commit/abort decision, but the guarantees associated with the decision may not be reflected in the physical state of the database state until some time in the future.

A lazy transaction system has a number of major benefits. Since transactions are executed only when one or more of the data that they modify are actually requested by another operation, we can reap tremendous gains from cache locality. To understand why, consider a chain of ten successive writes on record *r*. The system receives a read request for *r*, and then brings the appropriate database page into memory. The writes are then immediately applied to the in memory object, with no further calls to disk. The cost of a single disk read is amortized over eleven operations. This amortization occurs in contrast to the situation in a traditional, eager system, where the writes might be interleaved among other operations that cause the relevant page to be evicted from the page cache. In that case, each write must first fetch the page from disk before executing.

This is just one of the benefits of a lazy transaction system, others of which include decreased communication costs among elements of the RDBMS, and the ability to do temporal load balancing—i.e., even out bursts of requests to the database by deferring the execution of non-read transaction logic until a later point when the database has spare resources.

On the other hand, there are a number of disadvantages lazy transaction execution versus eager execution. In particular, as the system builds up a set of transactions not yet executed, an external read on a record affected by those transactions will suffer from a high latency while the database executes the set of transactions on which the read depends. This suggests that lazy databases might be more useful in contexts where external reads are infrequent and/or high latency reads are tolerable.

We can, however, attempt to mitigate the latency problem using scheduling heuristics that decide when to execute (a subset of) the unexecuted transactions. This paper focuses on the design of such heuristics and the performance tradeoffs involved. In Section 2, we overview the design of a lazy transaction manager, and in Section 3 we propose a few heuristics for scheduling transaction execution. We present experimental results in Section 4, and conclude in Section 5.

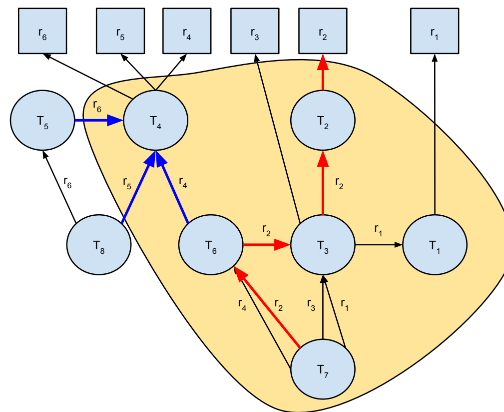


Figure 1: Example dependency graph. Transactions are numbered in log order.

2 System Design

Implementing a lazy transaction manager is incredibly non-trivial, since a lot of our intuitive reasoning about serializability and consistency breaks down when we remove the immediate temporal relations among a set of transactions. This is done in two phases, *stickification* and *substantiation*. Stickification is the part of transaction execution that occurs immediately, and in particular, evaluates the transaction to determine its record dependencies. It then generates system records to indicate which other transactions it depends on (i.e. when it accesses a value requested by a previous, potentially unexecuted transaction). In addition, the stickification phase executes any parts of the transaction that might cause the transaction to fail—specifically, integrity and other user-defined constraints or conditions—in order to immediately return a commit/abort decision.

Stickified transactions are stored for deferred execution by the substantiation phase. The transaction records implicitly create a partial ordering based on the dependencies among the stickified transactions. Using this partial ordering, we can create a directed acyclic graph (DAG) with edges representing dependencies. When the DBMS receives a read request for record *r*, then, it examines these records for all transactions that modify *r*, and then it executes those transactions to generate the present state of the tuple.

3 Scheduling Heuristics

In more detail, the stickification layer, upon receiving a new transaction, adds the transaction to a dependency graph (DAG) like the one shown in Figure 1. Under a purely lazy evaluation, this dependency graph would keep growing until the database receives a request for an external read, at which point the transaction manager would find the most recent transaction T_1 in the DAG that modifies the given record, and then execute the entire chain of transactions on which it depends. After substantiating the relevant set of transactions, the transaction manager would remove them from the DAG and return the updated value of the requested record.

Although this is a perfectly reasonable system design when the frequency of external reads is relatively high, it suffers from high latencies when the frequency of external reads is low because the size (and therefore the number of transactions that have to be substantiated) of the DAG is unbounded. If, however, we place limits on the size or shape of the DAG, we might be able to improve read latencies with some minimal cost to throughput (or even improve both). To this end, we present three potential heuristics for scheduling the substantiation of stickified transactions, chain length[1], size of transitive closure, and number of incoming dependencies. We describe each heuristic below, along with an intuition for how they might affect performance. We test our intuition with a microbenchmark in Section 4.

3.1 Chain Length

In order to read a record, the number of transactions that the database must substantiate is at least the depth of the subtree of the DAG containing references to the record in question. More precisely, the *chain length* is equal to the number of unsubstantiated transactions that have the given record in its read or write sets[2]. If we bound the chain length, therefore, we equivalently place an upper limit on the minimal cost of an external read. As an example, consider record r_2 in our sample Figure 1. Here, the chain corresponds to the red arrows, so the chain length of r_2 is equal to 4.

In our system, the chain length heuristic is implemented as follows. For each record in the system, the transaction manager maintains a counter (initially zero) that is updated every time a transaction depending on that record is added to the DAG. If, upon adding a new transaction, this counter exceeds the specified maximum for any of the dependent records, the transaction manager immediately schedules the transaction for substantiation and resets the relevant counters.

3.2 Size of Transitive Closure

Although chain length is a reasonable lower bound on the number of transactions that must be substantiated on an external read, the actual value is equal to the size of the *transitive closure* of the most recent transaction referencing the given record r (call this transaction $T(r)$). Specifically, the transitive closure is the number of nodes in the subtree with $T(r)$ at the root and including all descendents of $T(r)$. For r_2, the transitive closure is the set of transactions within the yellow area, giving a cardinality of 6.

For particularly linear, deep trees, bounding the size of the transitive closure has approximately the same effect as bounding chain length, i.e. bounding the worst, best-case cost of an external read. However, for wider, branching trees, this can also improve the cache locality of transaction execution by setting the maximum to a value that ensures all referenced records can fit into memory buffers.

We target the second of the two goals in our system. When a new transaction is added to the DAG, we sum the chain lengths of all dependent records (giving us an upper bound on the size of the transitive closure). If this value exceeds our maximum, we randomly substantiate one of the immediate subtrees of the new transaction. We do this, rather than substantiating the newly added transaction (and in turn, all of the immediate subtrees) because, for some appropriate maximum, the chain length heuristic could achieve a similar effect. On the other hand, this approach seeks not to limit the external read latency, but to increase the speed at which a record can be substantiated by automatically pruning the DAG and thus improving the cache locality of the substantiation phase.

3.3 Number of Incoming Dependencies

Another potential bottleneck on read latencies is the simultaneous dependency of multiple stickified transactions on some other transaction. If the system simultaneously attempts to substantiate those transactions, there may come a point where they all have to wait for the dependency (and its entire transitive closure) to execute. The system therefore pays the cost of substantiating that dependency multiple times. On the other hand, if the transaction manager can somehow anticipate that situation and substantiate the dependency before it is needed, then it might be able to significantly decrease the latency of external reads.

[1] Note this heuristic was initially proposed in [2]

[2] The read set is relevant because the transaction manager must respect the serial ordering of the transactions determined during stickification, so any internal reads (reads involved in transaction logic but not returned to the user) must be performed on the value of the record at the correct point in the overall serial set of transactions.

For each transaction in the DAG, the transaction manager maintains a counter that is incremented every time some new transaction is added and creates an edge with that transaction (i.e. the counter keeps track of the in-degree/number of parents of the transaction in the DAG). If the counter exceeds the specified maximum, then the dependency (*not* the new transaction) is queued for substantiation. Referring back to Figure 1, we see that T_4 has 3 incoming dependencies. If the limit had been set to 2, then upon adding T_8 to the DAG, the transaction manager would have enqueued T_4 for substantiation.

As mentioned, this heuristic has the benefit of decreasing the amount of 'repeated' work performed during substantiation (we count waiting for the duration of the substantiation of the dependency as 'repeated' work, since it will affect the latency of all the dependent transactions). We therefore would expect the heuristic to significantly decrease read latencies (more so than the other two heuristics). On the other hand, since this heuristic can lead to multiple transactions being enqueued for substantiation (each to different threads), it will decrease cache locality over executing the new transaction and all of its dependencies in the same thread. Thus, in exchange for the decreased latency, we would expect a drop in throughput for more aggressive (i.e. lower) limits on the the number of incoming dependencies.

4 Experiments

In order to evaluate the performance of the various substantiation heuristics, we ran the microbenchmark developed in [2] for different system configurations and thresholds. Specifically, the benchmark comprises transactions over a 1 million record (1024 bytes each) dataset, where each transaction's read and write sets include 10 total records. The read and write sets are determined by first selecting one of the 10 records uniformly at random, and then selecting the remaining 9 from a normal distribution with the first record as the mean and a standard deviation of 30. This sampling strategy is designed to mimic circumstances in which data accesses are correlated, since transactions overlapping on one record will likely overlap on others. Lastly, the transactions include external reads at the rate of 1 in 1000, which is reasonable in the write-intensive environments targeted by lazy database systems[3].

The underlying physical system has 64 GB of main memory and 10 physical cores. For these experiments, we allocated 8 cores to the database. In the case of lazy evaluation, 1 of these cores was allocated to stickification and the other 7 to substantiation, whereas for the standard, 'eager' system, the 8 cores were uniform in purpose.

We implemented the heuristics as part of the stickification portion of the transaction manager. In the purely lazy implementation, the stickification layer first executes a portion of each transaction so it can determine whether or not the transaction commits or aborts. Then, assuming the transaction 'committed', the stickification layer adds the transaction (along with its remaining, unexecuted logic) to the dependency graph. Additionally, it maintains a set of tables for each relation in the database indicating for each record what transaction last modified that record. This auxiliary data structure is necessary because on a read request, the system needs the ability to check whether or not the record's value actually reflects the current state of the database system, or in other words, whether or not there are transactions affecting the record that have not yet been substantiated.

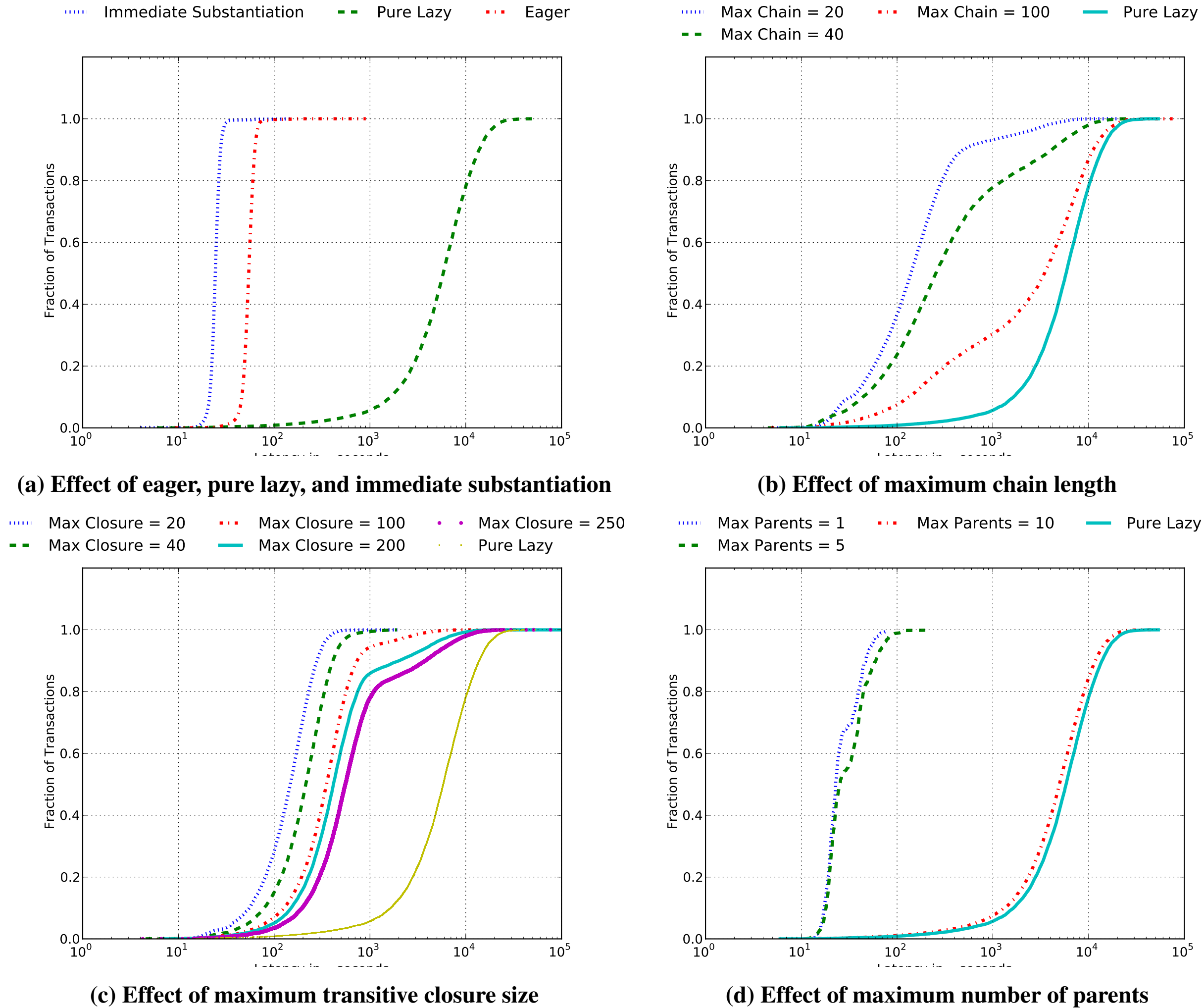
As outlined in Section 3, our heuristics require a few additional fields stored as part of the nodes of the dependency graph and the auxiliary tables. For the case of chain length, we added a counter to the elements of the auxiliary tables that indicate how many unsubstantiated transactions read from or write to the given record. When a new transaction is added to the dependency graph, this counter value is incremented if the transaction affects the record, and any time a transaction (and its transitive closure) are substantiated (from an external read or because a record's chain length exceeds the specified maximum), the value of the counter is reset to its correct value (usually zero) for all affected records.

For the limit on the size of the transitive closure, we estimate the quantity when a new transaction is added to the dependency graph. Specifically, we sum the chain lengths of each of the dependencies to get an upper bound on the size of the transitive closure for the new transaction (it is an upper bound because some of the dependencies could be modified by the same transaction). If the value exceeds the specified maximum, one of the dependencies is selected at random, and it (rather than the parent) is scheduled for execution.

Lastly, the number of incoming dependencies for a transaction (i.e. the number of parents) is maintained with an attribute of the nodes of the dependency graph. Each time a new edge is added to the graph, the terminal node has its counter incremented. If this value exceeds the limit, then the transaction represented by the terminal node is substantiated.

Although these heuristics can be applied in combination, our experiments evaluate them independently in order to better demonstrate the dynamics of a lazy system as well as the tradeoffs involved with any pre-emptive (i.e. non demand-based) substantiation. Our results are detailed below.

[3] For comparison, the TPCC benchmark has an external read frequency of about 1 in 20



(a) Effect of eager, pure lazy, and immediate substantiation

(b) Effect of maximum chain length

(c) Effect of maximum transitive closure size

(d) Effect of maximum number of parents

Figure 2: External read latencies for microbenchmark. X-axes are in microseconds

4.1 Latency

The first set of benchmarks, shown in Figure 2a, evaluated the relative performance characteristics of eager and lazy database systems, with the latter being split into the cases of pure laziness (substantiating only on external read) and immediate substantiation (which retains the benefits of shortening a transaction's critical path (by splitting execution into a 'Now Phase' and 'Later Phase'; see [2]) but is otherwise similar to an eager system in that the stickified transactions are substantiated right away.

The results, as we see in the figure, fall directly in line with the system designs. The pure lazy system has the highest read latencies because, in absence of external reads, the number of unsubstantiated transactions is unbounded (as we will see, however, this system has the best throughput of the 3). On the other hand, the lazy system with immediate substantiation outperforms the eager system due to the characteristic described above—contention in a lazy system is significantly lower than an eager system, so the system is better able to utilize idle resources.

The second set of benchmarks uses the purely lazy system as a reference point in order to evaluate the impact of the scheduling heuristics introduced in Section 3. Examining each in turn, we see that all three of chain length (Figure 2b), transitive closure size (Figure 2c), and incoming dependencies (parents) (Figure 2d) improve read latencies by 2–3 orders of magnitude when compared to a purely lazy system with no heuristics. The performance of each heuristic does vary based on the value of the parameter because they are closely tied to the properties of the workload. For example, when external reads are frequent and each transaction affects a large number of records, a high maximum chain length would induce very few substantiations, whereas an equivalent maxi-

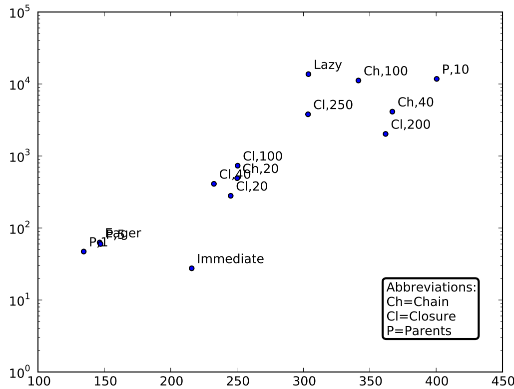


Figure 3: Throughput (thousand txns/sec) vs. 90th-percentile latency

mum transitive closure size would induce more (because the DAG would tend to be wider than it is deep).

An interesting extension of the system would be to include an online learning algorithm for these heuristics subject to some objective function controlled by the user (how aggressively to minimize latency versus throughput; we explore this next). Furthermore, the heuristics can easily be utilized in tandem, because each affect a different property of the DAG.

4.2 Performance Tradeoffs

Despite the promising decrease in external read latency as a result of implementing the substantiation strategies listed above, the strategies do come with potential throughput tradeoffs depending on how aggressive the transaction manager decides to be in automatically substantiating transactions. The extent of these tradeoffs are shown in Figure 3, which compares the transaction throughput with the 90th-percentile read latency under the various system configurations.

We first note that, as expected given our discussion above, the eager and immediate substantiation systems have two of the lowest read latencies but also two of the lowest throughputs. Furthermore, the pure lazy system has a significantly higher latency (refer to Figure 2a), but this comes in exchange for about a 200% increase in throughput.

Those three reference points in mind, and the fact that any point A below and to the right of another point B is strictly better, we discuss the tradeoffs introduced by the substantiation strategies. For aggressive (i.e. low) maxima for chain length and transitive closure size, we see about a 1–2 order of magnitude improvement in read latencies in exchange for a decrease in throughput of about 50%. On the other hand, there do exist values for chain length and closure size (e.g. 40 and 200, respectively) that strictly improve the performance of the lazy system. In general, however, there is an inverse relationship between throughput and latency. An eager system generally achieves lower latencies but also lower throughputs, whereas the lazy systems generally achieve higher throughputs at the cost of read latency. Thus, depending on the context and use case, both types of systems have their benefits.

5 Conclusion

In this paper, we have overviewed the concept of a lazy database system and its benefits over a standard, eager database system. We have observed that the stickification of records for later substantiation opens the possibility for introducing a number of strategies for scheduling those substantiations, whether they are demand-driven (external reads) or based on some internal set of statistics. We then introduced three such heuristics, chain length, size of the transitive closure, and number of incoming dependencies, and showed that with the right parameters, each heuristic can improve read latencies by an order of magnitude over a pure lazy system. We further explored the tradeoff between latency and throughput in a lazy database system, concluding that there are system configurations that are strictly dominant to a pure lazy system, but that in general, a lazy system achieves higher throughput at some cost to latency. versus an eager system.

6 Acknowledgements

I would like to thank Daniel Abadi for his feedback and support throughout the project and Jose Faleiro for sharing his benchmarking code. This work was sponsored by the NSF under grant IIS-1249722.

References

[1] VoltDb: Technical Overview. `http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf`.

[2] Jose Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of SIGMOD*, 2014.

[3] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of SIGMOD*, 2012.