Lazy Evaluation of Transactions in Database Systems

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Alexander Thomson
Google*
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Existing database systems employ an *eager* transaction processing scheme—that is, upon receiving a transaction request, the system executes all the operations entailed in running the transaction (which typically includes reading database records, executing user-specified transaction logic, and logging updates and writes) before reporting to the client that the transaction has completed.

We introduce a *lazy* transaction execution engine, in which a transaction may be considered durably completed after only partial execution, while the bulk of its operations (notably all reads from the database and all execution of transaction logic) may be deferred until an arbitrary future time, such as when a user attempts to read some element of the transaction's write-set—all without modifying the semantics of the transaction or sacrificing ACID guarantees. Lazy transactions are processed deterministically, so that the final state of the database is guaranteed to be equivalent to what the state would have been had all transactions been executed eagerly.

Our prototype of a lazy transaction execution engine improves temporal locality when executing related transactions, reduces peak provisioning requirements by deferring more non-urgent work until off-peak load times, and reduces contention footprint of concurrent transactions. However, we find that certain queries suffer increased latency, and therefore lazy database systems may not be appropriate for read-latency sensitive applications.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems–*Transaction processing*

Keywords

ACID transactions; deterministic database systems; load balancing

1. INTRODUCTION

For decades, transactional database systems have worked as follows: upon receiving a transaction request, the database system performs the reads, writes, and transactional logic associated with the transaction, and then commits (or aborts). Upon receiving a query request, the database system performs the reads associated with the query and returns these results to the user or application that made the request. In both cases, the order is fixed: the database first performs the work associated with the transaction or query, and only afterwards does the database returns the results — read results, and/or the commit/abort decision.

In this paper, we explore the design of a database system that flips this traditional model on its head. For transactions that return only a commit/abort decision, the database first returns this decision and afterwards performs the work associated with that transaction. The meaning of "commit" and "abort" still maintain the full set of ACID guarantees: if the user is told that the transaction has committed, this means that the effects of the transaction are guaranteed to be durably reflected in the state of the database, and any subsequent reads of data that this transaction wrote will include the updates made by the committed transaction.

The key observation that makes this possible is inspired by the lazy evaluation research performed by the programming language community: the actual state of the database can be allowed to differ from the state of the database that has been promised to a client — it's only if the client actually makes an explicit request to read the state of the database do promises about state changes have to be kept. In particular, writes to the database state can be deferred, and "lazily" performed upon request — when a client reads the value of the state that was written.

Therefore, when a client issues a transaction and only expects a commit/abort decision as a result, the work involved in processing the transaction can be replaced by a simple "promise" to the client that it was done. Only when the state that is affected by this transaction needs to be returned to a client does the promise have to be kept, and the work associated with the transaction performed.

However, even if a promise made to a client does not have to be immediately kept, every promise that is made must be theoretically possible to keep. Therefore, in the context of database systems, some amount of work is necessary to return the correct commit/abort decision. There are two classes of scenarios that could cause a transaction to abort: (1) transaction logic can cause a transaction to abort that is dependent on the state of the database and the particular requests made by a transaction (e.g. if the transaction will cause an integrity constraint violation), and (2) the database decides to abort a transaction for nondeterministic reasons that are totally independent of database state (e.g. a deadlock is encountered, or if the database crashes in the middle of processing the transaction).

Given the nondeterministic nature of the second class of scenarios that could cause a transaction to abort, it is impossible in traditional database systems to make promises in advance that a trans-

*This work was done while the author was at Yale.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610529 .

action will commit without running the transaction to completion and then actually committing it. Hence, lazy evaluation of transactions has not been a viable option for traditional database architectures. However, an increasing number of *deterministic* databases have been introduced in the last few years, such as Calvin [22], H-Store [20], VoltDB [23], and Hyder [2], which completely disallow nondeterministic aborts. Once these types of aborts are removed, transactions can only abort due to client-defined transaction logic. Lazy evaluation therefore becomes possible by performing enough work during transaction initiation to determine if, given the current state of the database, transaction logic will cause an abort. If not, a "commit" decision can be immediately promised to the client, with the actual transaction processing performed lazily. Furthermore, the deterministic guarantees of the database are leveraged to ensure that when the database eventually gets around to processing the transaction, it will be processed over the snapshot of database state that existed when the transaction was originally submitted (instead of the current state) so that the same database state that was used to determine whether or not the transaction will commit will still exist at the time the transaction is processed.

In this paper, we describe how deterministic database systems can be extended to allow lazy evaluation of transactions, and explore the tradeoffs involved in this lazy transactional processing. In particular we find that laziness provides the following advantages:

- **Improved overall cache/buffer pool locality.** If record X is modified several times before its value is requested by a client query, it only needs to be accessed via an IO operation and brought into memory/cache once, when the writes and final read all occur together (§2.7).
- **Temporal load balancing.** Deferring execution of transactions requested at peak load times can reduce workload skew between peak and non-peak hours, lowering resource provisioning requirements (§2.6).
- **Avoiding unnecessary work.** Transactions need not *ever* run to completion fully if their write-sets are never read. This can happen, for example, if the write set of a transaction is overwritten by a blind write (a write that is not dependent on the current value of a data item).
- **Reduced contention footprint.** Contention can be reduced by only executing high-contention operations within a transaction eagerly (the rest of the work is executed at a later time, lazily). This reduces the size of the critical section around contended data access (§2.3 and §2.4).
- **Reduced transaction execution latency.** If a client is only expecting a commit/abort decision as a result of submitting a transaction to the database system, lazy evaluation allows this decision to be returned before the execution of most transactional logic, thereby significantly reducing the transactional latency that is observable by the client.

On the other hand, lazy transactions introduce certain hazards, and in parting with traditional transaction processing dogma, they introduce new challenges:

- **Higher read latencies.** A request by a client to read a data item may incur delay while writes to that item from other transactions have to be performed prior to the read.
- **Dependencies between deferred transactions.** If many conflicting transactions have been deferred, then substantiating one data item may involve running transactions that require substantiation of other data items (§2.2).
- **Overhead of determining the write set of a transaction.** In order to know what promises have to be kept before reading a data item, any transaction that is processed lazily must mark in some way all items that it will write, so that the database can ensure that these writes will occur before the data item is read. If the user does not explicitly provide the write set of a transaction, additional work is required to determine the write set before promises can be made (§2.3).

Given this new set of tradeoffs that lazy processing of transactions introduce, it is clear that some workloads are poorly suited for lazy evaluation, while other workloads will see significantly improved throughput, latency, and provisioning characteristics if transactions are executed lazily.

2. LAZY TRANSACTIONS

In order to illustrate and motivate our approach to implementing lazy transactions, we first examine a naïve implementation of a lazy database system. For the purposes of this example, let us begin by considering only transactions that do not contain logic that can cause them to abort (this restriction will be lifted shortly).

Our naïve lazy system logs transaction requests as it receives them, and replies to each client with a commit "promise" as soon as the request is durably written to the log. However, no additional action is taken immediately to execute the transaction and apply its effects.

When a client goes to read some record(s) in the database at some later time v, this prompts the log to be played forward, so that all transactions that appear before v are executed—then the client can safely read from a snapshot at time v. This playing-forward of the log may involve executing transactions in the log serially, or it may use a locking mechanism that guarantees equivalence to serial execution in the order specified in the log—e.g. deterministic locking [21, 22] or VLL [18]—to parallelize transaction execution and increase resource utilization.

In fact, deterministic database systems such as Calvin and VoltDB already implement exactly the machinery required for this type of lazy execution—but they replay the log eagerly rather than waiting for new read requests to prompt them along. This is because very little is gained by this implementation of lazy transactions—exactly the same transactions are executed using exactly the same scheduling mechanisms, but with artificial delays inserted.

The basic problem is that the naïve system has to play forward the *entire* log up to v in order to perform *any* read at timestamp v—even if many of the transactions that were executed had no effect whatsoever on the result of the client's read. Below, we describe an approach to implementing lazy transactions using "stickies" that has considerably more useful properties.

2.1 Stickies

We introduce our lazy execution technique with an example. Consider a transaction T that writes out a set of records $\{x, y, z\}$ whose values depend on the current values of a (possibly overlapping) set of records $\{a, b, c\}$, implemented as a stored procedure as follows:

```
T ({a, b, c}, {x, y, z}) {
  Read a.
  Read b.
  Read c.
  Perform local computation.
  Write x.
  Write y.
  Write z.
  Commit.
}
```

Note that T has two useful properties: (a) T's write-set is known at the time the transaction is invoked (since it is provided as an argument), and (b) like the transactions handled in the naïve approach described above, T will always commit if executed to completion. For ease of explanation, we will assume that all transactions have these properties for now, and describe how our lazy execution engine handles lazy transactions that do *not* have these properties in §2.3. We will also assume for now a data storage structure that supports full multiversioning. Each write inserts a new record without deleting the previous version of the record, even if the write logically clobbers the previous value.

A traditional (eager) database system executing T at time v would read the latest versions of $\{a, b, c\}$, perform whatever computation is specified by the transaction code, and write the resulting new versions $\{x_v, y_v, z_v\}$ to the storage system.

When T is processed by our lazy execution system, however, records $\{a, b, c\}$ are *not* read, nor is any of T's local computation executed—but new records *are* written out to $\{x^s_v, y^s_v, z^s_v\}$ nonetheless. Since no actual values have been computed, these records are *stickies*—temporary place-holders for the real values that provide the reader just enough information to evaluate the specific record when needed. (The superscript 's' denotes here that a record is a sticky.) For example, the sticky written to x^s_v in our example indicates to future readers of the record:

```
This record is a sticky and does not store
an actual value.  To compute this record's
value, execute transaction Tv.
```

After stickies have been created for each element of T's write-set and written out to the database's storage engine (and the transaction request T_v is appended to the transaction input log), T is considered to have been durably committed at time v. We refer to this process as *stickification*, and we refer to a transaction such as T_v that has committed by passing the stickification phase—but whose client-specified transaction logic has not yet been fully executed—as a *deferred* transaction.

2.2 Substantiating stickies

If a client subsequently looks up the record x at a timestamp $v' > v$ and finds that x^s_v is the most recent version of x, the server retrieves the transaction request T_v, executes it fully (including performing all reads and computation), and overwrites the stickies $\{x^s_v, y^s_v, z^s_v\}$ with the actual values produced, so that subsequent readers need not re-evaluate T_v again. We refer to this process as *substantiation* of x_v.

Note that T_v's read-set $\{a, b, c\}$ (specifically, the latest versions of these records that precede v) may also contain stickies written by earlier transactions. For example, when T_v attempts to read the latest version of a preceding v, it may find a sticky a^s_u inserted by an earlier transaction at time u (let's call that transaction T_u). In order to substantiate x^s_v, a^s_u must first be substantiated by looking up T_u's entry in the transaction request log and fully executing T_u.

In general, there may be many transactions on whose write-sets T_v (transitively) depends—all of which must be executed in order to finally substantiate x^s_v.

Transactional dependencies can be represented graphically, as shown in Fig 1. In Fig 1(a), $T1$ is ordered before $T2$ in the log. However, because $T1$'s read and write sets are mutually exclusive from $T2$'s, both may be executed independently. Fig 1(b) illustrates the distinction between the transaction ordering as imposed by the log and the actual data dependencies among transactions. The log ordering between transactions is shown by dotted lines. The data dependencies between transactions are depicted by solid lines.

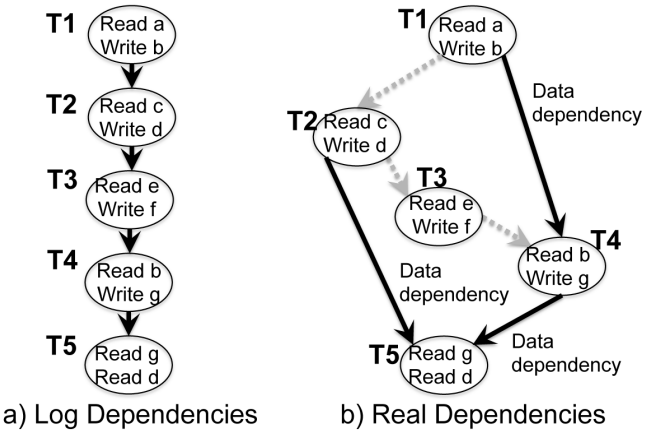

Figure 1: Transaction Ordering in the Log

Data dependencies between transactions form a *partial order*. The substantiation process needs to obey this partial order, *not* the stricter total order imposed by the stickification log.

A consequence of the partial order among transactions is that substantiation of transactions that *do not* have any data dependencies between them can be completely independent. Referring to Fig 1b again, a lazy database system could choose to substantiate the set $\{T1, T2, T4, T5\}$ today, and $T3$ at a later time.

Any partial order can be represented by a directed acyclic graph (DAG). Note that in order to substantiate a transaction T, we have to first substantiate every transaction T *transitively* depends on. For instance, consider again the data dependencies between transactions as depicted in Fig 1(b). $T5$ depends on $T4$ and $T2$, but it transitively depends on $T1$. To substantiate $T5$, we must substantiate $T2$, $T4$ *and* $T1$.

2.3 Partially Lazy Transactions

Up until now, we have considered only transactions (a) whose write-sets are known in advance and (b) that cannot abort due to specified transaction logic (e.g., integrity constraint checks). In order to handle the more general class transactions for which these properties need not hold, our execution engine actually executes transactions "partially lazily", dividing each transaction into two phases:

- a **now-phase**, executed immediately, and
- a **later-phase**, which may be deferred until some element of the transaction's write set actually needs to be substantiated.

A transaction's now-phase generally includes:

- All constraint checks and client-specified logic needed to determine commit decisions.
- Any reads from the database state that are necessary to determine a transaction's read/write set. One example of this is a transaction that reads a record via a secondary-index lookup. Without doing the lookup, it is not possible to determine which record must be read. For example, in TPC-C, all $OrderStatus$ transactions must read a customer record, and for some fraction of these transactions the record's primary key must first be determined by a secondary index lookup on the customer's name. If no secondary index is maintained on that field, a full table scan must be performed during the now phase, exactly as it would be with an eager execution mechanism.
- Inserting stickies for each element of the write-set that will be written to during the transaction's later-phase. In addition, sec-

ondary index records must be updated to reflect any changes that the transaction will make to indexed fields.

In order to provide full serializability, transactions' now-phases must be executed in a manner that guarantees equivalence to serial execution in log order. This is one of the places where the use of a deterministic system such as Calvin [22], H-Store [20], VoltDB [23], and Hyder [2] is very helpful — these systems are capable of performing transactions in parallel while still guaranteeing equivalence to a deterministic sequential execution in a specific predetermined order.

Determining a transaction's commit decision in the now-phase is not necessarily straightforward. To examine some common patterns, we examine TPC-C's $NewOrder$ transaction.

$NewOrder$ transactions make up the bulk of the TPC-C benchmark in terms of transaction numbers, work done when running the benchmark, and total contention levels between transactions. Each $NewOrder$ simulates a customer placing an order for between 5 and 15 items from an online retailer, and consists of several steps[1]:

1. **Constraint check.** Abort if a requested item has an invalid ID. The TPC-C specification states that 1% of all $NewOrder$ transactions should fail this constraint check.
2. **One high-contention Read-Modify-Write (RMW) operation.** Increment one of ten Districts' `next_order_id` counters.
3. **8-18 no-contention reads.** Read records from the Warehouse, District, Customer, and Item tables.
4. **5-15 low-contention RMW operations.** Update Stock records for each item purchased.
5. **8-18 blind writes.** Insert records into the Order, OrderLine, NewOrder, and History tables.

Since a TPC-C $NewOrder$ transaction only aborts in the event of an invalid item request, including operation 1 above in the now-phase while leaving the rest of the transaction's logic in the later-phase is sufficient to determine the commit decision. (Performing checks of this kind at the beginning of a transaction's code is a common idiom in transactional applications.)

Uniqueness constraints are also common and can be handled specially. Suppose that $NewOrder$'s blind write into the History table required a uniqueness check on its primary key[2]. Database systems often use bloom filters for uniqueness checks, and this technique can be applied here. Whenever a record is inserted into a relation, the primary key of this record is checked and inserted into a bloom filter. Note that even unsubstantiated records for whom a sticky has been created but not yet substantiated can be included in this bloom filter since all stickies include the primary key value. In this scenario, the now-phase would first check the bloom filter for the record in question (let's call it h). If the bloom filter indicated that no previous version exists, the check passes, and h must be inserted into the bloom filter (in addition to the sticky h_v^s being inserted into the History table). If the bloom filter showed that a previous version of h may in fact exist (which might be a false positive)—or if a bloom filter were not used—the $NewOrder$ transaction then performs a *non-substantiating* read of h. Such a read looks up h in the History table, and if it finds a sticky h_u^s, it does *not* attempt to substantiate it or discern its value by recursively executing the transaction that inserted it, since the precise value is not needed to discern that the uniqueness check has failed.

[1]For brevity, we examine here a TPC-C deployment consisting of a single warehouse.

[2]This is only a hypothetical scenario—History record primary keys are guaranteed to be unique, so this is *not* actually necessary according to the TPC-C specification.

For uniqueness checks on non-primary-key columns, secondary-indexes on those columns would have to be maintained in the now-phase in order to avoid full table scans (as mentioned earlier).

A third possibility is that a transaction may violate other types of integrity constraints or have user-specified conditional logic triggering an abort. For example, a system may abort any transaction that would result in a negative stock level. In such a case even more of the transaction logic is forced to execute within the now-phase—limiting the amount of laziness that is achievable for certain classes of transactions.

2.3.1 Specifying now and later-phases

In order to maximize the benefits of lazy execution, transactions need to be divided into the now and later-phases. There are two options for doing this: user-driven or automatically-driven. In our implementation discussed below, we choose the user-driven approach. Clients specify transaction logic by registering C++ stored procedures. In each procedure, all code is executed in the now-phase up until a special `EndNowPhase()` method is called. When the stored procedure code calls `EndNowPhase()`, stickies are written out to all not-yet-updated elements of the transaction's write set, and control passes back to the calling thread at the call site of the stored procedure. Execution resumes from the same place only when a sticky written by the transaction is substantiated.

Stored procedures whose logic does not contain any call to `EndNowPhase()` execute entirely eagerly and insert no stickies; it is safe for workloads to mix eager and lazy transactions.

Depending on client-provided annotations to determine how much of each transaction to defer to a lazy phase has a clear cost: it imposes an additional burden on database system users, who must reason carefully about data dependencies to safely use lazy transaction evaluation. To ameliorate this, it is possible to introduce automated dependency analysis tools (similar to those commonly used in compilers) that could statically detect the earliest place in the stored procedure logic where it would be safe to call `EndNowPhase()`. For example, in the case of the TPC-C $NewOrder$ stored procedure, it could be statically determined that no operations after the constraint check could lead to an abort decision or modify the write set. By carefully choosing what work to defer, clients can also reduce lock contention as a bottleneck in high-contention workloads, as we discuss in the following section.

2.4 Transaction Contention in Lazy Database Systems

A transaction's *contention footprint* is the duration of time that it limits the total concurrency achievable in the system. If transactions are executed serially in a single thread, each one has a contention footprint of its entire active duration, since no other transaction may execute until it completes. In systems that use locking for concurrency control, contention footprint corresponds to the length of time that a transaction holds locks, thereby preventing conflicting transactions from executing. In systems that use optimistic concurrency control, a transaction's contention footprint is the time period from when it starts executing until the critical section of its validation phase—the period during which other transactions may have performed writes that then cause it to fail validation.

The contention footprint of a transaction executing in a lazy database system is more complex, since transactions are not executed all at once. In general, total system throughput can be limited by either now-phase contention or later-phase contention.

Two transactions' now-phases conflict with one another if the two transactions would have conflicted had they been executed eagerly. However, when blocking on another transaction's now-phase,

it is only necessary to block until that transaction's now-phase (including stickification) completes. The blocked transaction can begin executing its now-phase before the first transaction's later-phase. Thus, although the likelihood of transactions conflicting is no different, the total contention between now-phases footprint is reduced compared to eager execution.

Later-phase "contention" manifests as dependencies when substantiating stickies: when multiple transactions' later-phases require a single sticky to be substantiated, the amount of parallelism that can be achieved is reduced, since the worker threads processing those later-phases block on that one substantiation operation.This blocking behavior mirrors that which would be observed when executing the same transactions eagerly using a lock-based concurrency control scheme—readers block getting read locks until writers release their exclusive write locks.

Note that unlike traditional locking protocols, and even MVCC, later-phase contention does not reflect write-write conflicts, but only read-write conflicts, since it is only necessary to substantiate existing stickies when reading a record, not when overwriting it with a new value[3].

Furthermore, later-phase dependencies between transactions only appear for read-write conflicts on records on which the earlier transaction inserted a sticky. If the earlier transaction wrote out a record's actual value (not a sticky) during its now-phase, then later readers of that record need not block on the transaction's later-phase, since the value does not need to be substantiated.

This last observation introduces a subtle but powerful opportunity to reduce transactions' contention footprints by pushing high-contention RMW operations into the now-phase and leaving low-contention reads and writes in the later-phase. For example, while TPC-C $NewOrder$'s constraint check is the only step that *must* execute during the now-phase, incrementing the high-contention `District.next_order_id` counter could also be done during the now-phase, while leaving the remaining operations (around 13 contention-free reads, around 10 low-contention RMWs, and around 13 blind writes) in the later-phase. This allows both the now-phase contention and the later-phase contention to be much lower than contention levels observed when executing $NewOrder$ transactions eagerly. Now-phases still conflict with high probability, but are very short, consisting of only the constraint check and incrementing a counter, so blocked transactions need not block for long before running their now-phases. Later-phases do not need to do the RMW operation on any Districts' `next_order_id` counters, and so each transaction only depends on earlier transactions that updated conflicting sets of Stock records, resulting in a much lower contention rate.

2.5 Foundations of Laziness

Our work builds on the theoretical foundations and early implementations of lazy programming languages [24, 10, 6, 11]. In particular, this early work defines a *pure* expression as one whose evaluation depends neither on any external behavior (such as a globally mutable variable being modified by another thread) nor performs any externally visible action (such as printing output or sending messages over a network). A pure expression always evaluates to the same value, regardless of whether eager or lazy evaluation is used. A *function* is considered pure if, when applied to a pure expression, the resulting expression is also pure.

Existing transaction processing systems constrain themselves to eager transaction execution due to an implicit assumption that unpredictable events during transaction evaluation may cause transactions to abort. In other words, transactions are presumed *not* to be pure functions from one database state to the next.

The key observation underlying this work is that database transactions *can* often be formulated as pure functions on database state, introducing the possibility of lazy evaluation. However, this is only possible if deterministic database techniques are used to ensure non-deterministic aborts do not happen [21, 22, 20]. These systems accomplish this by ordering all transactions into a single serial order before executing them, and writing out this order to a log on stable storage (or across the network). They then use a deadlock-free concurrency control protocol that guarantees equivalence to this serial order that had been defined in advance. A node failure cannot cause a transaction to abort; instead the ordered transaction log is replayed upon a failure (from the most recent checkpoint) to get the database into the same state that it was in at the time of the failure and finishes all in-process transactions from there.

2.6 Reducing Peak Provisioning Requirements

Partitioning data across multiple machines is currently the most popular method of servicing high transaction throughput. Repartitioning data on-the-fly to make use of a varying number of machines is challenging, so most practical systems provision a large number of machines, in order to deal with peak traffic. However, these extra machines are not fully utilized most of the time.

A lazy database system can deal with bursty traffic more elegantly. A lazy database can choose to limit the rate of substantiation while dedicating more resources to stickification. When traffic subsides, it can begin substantiating transactions at a higher rate. Since the stickification of a transaction in a lazy database is much less expensive than evaluating a transaction in a conventional system, a lazy database deals with an increased rate of traffic without resorting to adding more machines to the system.

It should be noted, however, that this bursty traffic must be mostly transactions that return only commit/abort decisions. If there are many read queries in this burst of traffic, then stickification is not able to get much farther ahead than substantiation, and lazy execution does not help with peak provisioning.

2.7 Improving Cache (Buffer Pool) Locality

Consider the case of any two transactions that have a data dependency. We would expect better performance if they were substantiated together, than if their substantiations were separated by a large period of time. Substantiating a transaction involves bringing its records into a processor's cache (and also to the buffer pool for non-main memory systems). If the transactions were substantiated together, then the records the second transaction *shares* with the first will be cache (buffer pool) resident. Thus, evaluating such transactions together yields better cache (and buffer pool) locality. A lazy database system can delay substantiating transactions until the size of a set of *data dependent* transactions is large enough to take advantage of cache locality over large sets of transactions. Since conventional database systems evaluate transactions immediately, they cannot exploit such data sharing between transactions if the difference between the times the transactions enter the system is sufficiently large. In other words, eager databases systems can only expect records to be read from cache if there is temporal *and* spatial locality. Lazy database systems artificially create temporal locality and do not require it to exist naturally in the workload.

2.8 Logging and Recovery

As mentioned in §2.3, in order to provide full serializability, transactions' now-phases are executed in a manner that *determin-*

[3]Note that if a transaction does a RMW operation on a value, it does conflict with earlier writers of that value due to the read part of the RMW operation.

istically guarantees equivalence to serial execution in transaction log order. Given a transaction log, there is only one possible final state of the database system. Therefore, as long as the transaction log is persisted to stable storage, the database system can recover by replaying the log, a concept introduced by other deterministic systems [22, 14]. In order to avoid replaying the entire history of all transactions, checkpointing is necessary. Checkpoints must include a complete snapshot of the database as of a particular point in the log where all operations for all transactions before this point are reflected in the database state in the checkpoint, and no operations from future transactions are reflected in the checkpoint — i.e. the checkpoint must be taken at a "point of consistency" with respect to the transaction log. In the case of a lazy database system, a point of consistency cannot be reached unless there are no unsubstantiated transactions present in the system. Although quiescing the database system and finishing the execution of all unsubstantiated transactions before new transactions enter the system is one way to achieve a point of consistency, we use Calvin's mechanism of creating a virtual point of consistency without quiescing the system, and creating a checkpoint over a period of time as transactions that were unsubstantiated at the time of the virtual point of consistency get substantiated [22].

3. IMPLEMENTATION

We have taken a clean slate approach to building our lazy database prototype. We have implemented our prototype based on the idea of separating the execution of transactions into two phases; a stickification phase, and a substantiation phase (§2). Corresponding to the two transaction execution phases, our system architecture is divided into two layers; a stickification layer, and a substantiation layer. The stickification layer is responsible for returning a commit/abort decision to clients and determining the *dependencies* between transactions; once it determines that a transaction can commit, the stickification layer analyzes the transaction's read set, and determines which prior transactions it depends on. The substantiation layer is responsible for executing transactions; it uses the dependency information determined by the stickification layer to batch the execution of a set of dependent transactions. The rest of this section discusses the implementation of the stickification and substantiation layers in detail.

3.1 Stickification Layer

The stickification layer is the component that receives external input. Processing a transaction involves three steps: first, determining the transaction's commit/abort decision and finding its dependencies, second, maintaining heuristics to process transactions, and third, handing transactions to the substantiation layer.

3.1.1 Dependency Maintenance

When the stickification layer takes a transaction, T, from its input queue, it first executes the transaction's now-phase in order to determine the transaction's commit/abort decision. If the transaction commits, the stickification layer must determine the prior transactions T depends on. It determines T's dependencies based on T's read set; for each record in T's read set, T depends on the last transaction to have the record in its write set. We keep track of the last transaction to have written each record in the system using auxiliary tables mapping primary keys to a 128-bit pair of two 64-bit values: first, a 64-bit transaction identifier corresponding to the record's last writer, and second, a 64-bit counter (§3.1.2). For each record in T's read set, we look up the auxiliary table and record a reference to the record's last writer within T. Finally, for each record in T's write set, we update the auxiliary table to reflect T as the record's last writer.

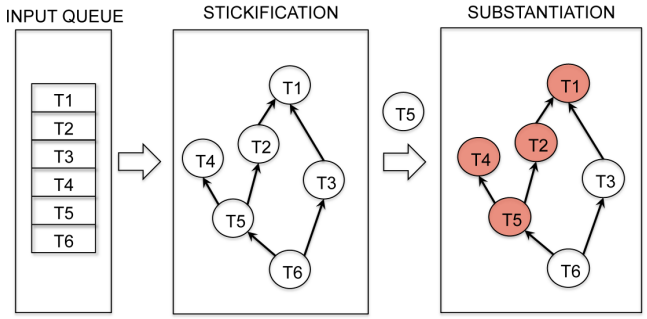

Figure 2: Work flow of Lazy Transaction Execution. White circles correspond to Stickified transactions. Red circles correspond to Substantiated transactions.

In addition to writing the auxiliary table, the stickification thread uses an array local to T to keep track of T's dependencies. As a consequence of tracking dependencies within transactions, the stickification thread maintains an implicit *dependency graph* of transactions. Fig 2 shows the work flow of transaction processing in a lazy database system. It shows the stickification layer processing transactions from an input queue, analyzing the transactions, and maintaining the dependency graph. The transactions that make up the graph correspond to as yet *unevaluated* transactions. In order to evaluate a particular transaction T, we have to first recursively evaluate all the transactions T depends on. For instance, if we wish to evaluate transaction T_5 in Fig 2, we need to have evaluated T_1, T_2 and T_4.

3.1.2 Heuristics

If the dependency graph is allowed to grow arbitrarily, then the latency of an external read (a read that must be returned to a database user) is adversely affected. External read latency increases with the size of the dependency graph because the external read might have a very long chain of transactions it depends on, and this chain of transactions must be executed *before* the external read can be executed. As a consequence, an external read incurs the latency cost of executing every transaction it depends on.

To ensure that the dependency graph does not grow unreasonably large, we keep track of the total number of unexecuted transactions that access each record. We store this information in the second counter field of the value in the auxiliary last-writer table (§3.1.1). Whenever a transaction looks up the value keyed by a particular record in the last-writer table, we update the counter field of the value. When we update the counter, we check to see if it exceeds a certain user-defined threshold; if it does, we hand the current transaction to the substantiation layer and reset the counter to 0. Intuitively, the larger the value of the threshold, the longer the chain of transactions that access a particular record.

3.2 Substantiation Layer

The substantiation layer takes transactions that are handed to it by the stickification layer and executes them to completion. Our implementation of the substantiation layer consists of multiple worker threads evaluating transactions in parallel. The execution of a transaction on the worker thread proceeds in two steps:

1. **Recursive Dependency Evaluation.** Before executing a transaction's logic, we first need to ensure that its dependencies have been evaluated. The stickification layer ensures that every transaction maintains a *reference* to each of its dependencies (§3.1.1).

The worker thread looks up each of a transaction's dependencies and checks to see if the dependency has been evaluated; if it has not, then the dependency itself must be recursively evaluated. Fig 2 shows the set of transactions that are executed when the substantiation layer is handed transaction T_5. The worker thread must first recursively evaluate T_5's dependencies before it can evaluate T_5.

Two different substantiation layer worker threads working in parallel on different transactions may have an overlapping dependency graph. In order to ensure that transactions within the overlapping subgraph are not processed more than once, each transaction structure is augmented with a single bit which serves as a spinlock. The spinlock protects internal transaction state by ensuring that only one worker thread may substantiate the transaction.

2. **Transaction Logic Evaluation.** Once all of a transaction's dependencies have been evaluated, the thread can proceed with evaluating the transaction's logic; the worker thread reads the records in the transaction's read set, and writes out the records in the transaction's write set.

Executing dependent transactions immediately one after another in the two steps outlined above allows the worker thread to *amortize* the cost of bringing a particular record into on-chip cache across all the transactions that access the record. For instance in Fig 2, when the worker thread executes T_5, it can re-use the records brought into cache when it executed T_1, T_2, and T_4 (assuming all their records together fit in cache).

4. EXPERIMENTAL EVALUATION

Our prototype lazy database consists of a single-threaded stickification layer and a multi-threaded substantiation layer (§3). As a comparison point, we implemented a system which uses a traditional two-phase locking concurrency control mechanism. Our two-phase locking prototype is built by replacing our lazy database prototype's concurrency control module.

Our experimental evaluation is conducted on a 10 core Intel Xeon E7-8850 processor using 64GB of memory. Our operating system is Linux 3.9.2. We dedicate 8 out of the 10 cores to the transaction processing engine for both 2PL and the lazy system in each of our experiments; both systems use the *same* number of cores. In the 2PL system, each of these 8 cores is utilized by a worker thread. In the lazy system, we dedicate 1 core to the stickification layer, and 7 cores to the substantiation layer. Of the two cores that remain on the system, one is used to drive the database input, while the other is used to measure performance. The measurements for all of our throughput experiments are averaged over 10 runs, the variance across runs in each of our experiments is negligible. Each line in our CDF plots shows a distribution over at least 500,000 points.

This section is organized as follows. §4.1 describes a set of microbenchmarks designed to evaluate the basic tradeoffs of lazy transaction processing relative to conventional eager processing. §4.2 evaluates the benefit of deferring transaction execution in the presence of load spikes. In §4.3 we explore the benefits of eliminating the need for evaluating transactions in the presence of blind writes. Finally, §4.4 evaluates the benefits of laziness in a high-contention multithreaded environment using the TPC-C benchmark.

4.1 Microbenchmarks

We begin our experimental evaluation using a simple microbenchmark involving transactions that perform read-modify-write operations on 10 distinct records The database consists of a single table with 1,000,000 records. Each record has a size of 1024 bytes, and

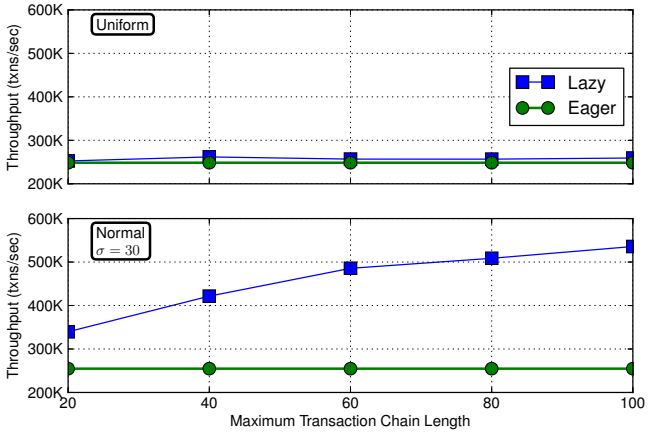

Figure 3: Microbenchmark Throughput

is indexed by a 64-bit primary key. We pick unique records in each transaction's read-write sets from two different distributions:

1. **Uniform.** We pick 10 unique records among the 1,000,000 uniformly at random.
2. **Normal.** We first select a single record among the 1,000,000 records uniformly at random. The remaining 9 are selected according to a Gaussian distribution around the first record using a standard deviation of 30. The effect of selecting records in this manner is that if two transactions have conflicting read-write sets, then they often conflict on several records. This workload is designed to model applications for which there are correlations in data access; users that buy item X tend to also buy Y in an online shopping scenario, or friends (followers) of X tend to also be friends (followers) of Y in a social networking platform.

4.1.1 Throughput

We first compare the transactional throughput of the lazy and eager systems. Throughput is defined as the rate at which transactions are fully processed. For the lazy scheme, only transactions that have finished both the stickification and substantiation phases (i.e., are completely finished) count towards throughput. The lazy execution scheme is parameterized by the length of the longest chain of unsubstantiated transactions that are allowed to exist before they get automatically substantiated (§3).

Fig 3 shows how the throughput of lazy transaction processing varies with the bound on the longest chain of unexecuted transactions. We plot two graphs, one for the Uniform workload and the second for the Normal workload.

For the Normal workload, we see that the lazy system gets significantly better throughput than the eager system. This is due to the improved cache locality of the lazy system — when substantiating a chain of transactions, the later transactions in the chain find many of the records they access already in cache, having been brought into cache by earlier transactions in the chain. The cost of the initial access of a record (to bring it into cache) is amortized over the number of subsequent accesses in the chain. Therefore, the throughput difference between the lazy system and eager system increases as the maximum chain size increases.

In contrast, for the Uniform workload, we see that the throughput of the lazy and eager systems is very similar; and furthermore, the chain bound has no effect on the throughput of the lazy system. This is because in the Normal workload, when two transactions overlap in their data access, they tend to overlap on multiple records. Therefore, substantiating a chain of transactions will lead to cache benefits for multiple records. However, in the Uniform

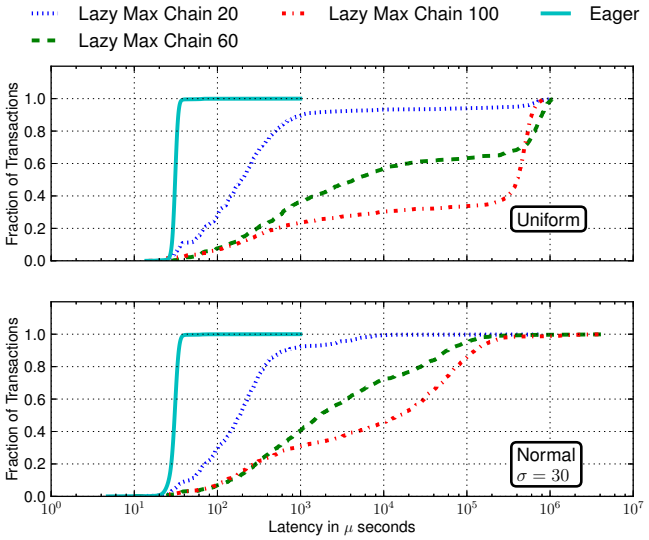

Figure 4: External Read Latency CDF

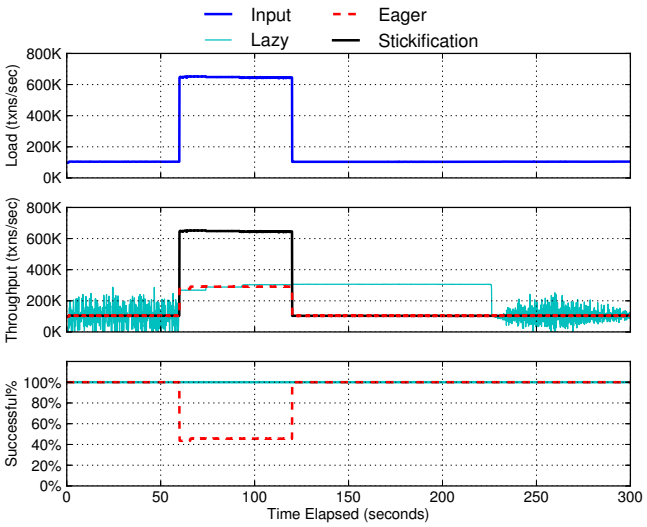

Figure 5: Peak Load

workload, two transactions almost never overlap on more than one record. Therefore, the only cache benefits from substantiating a chain of transactions together is just for the one shared record in the chain. This cache benefit is almost completely offset by the additional overhead of maintaining the dependency graph.

4.1.2 End-to-End Latency of Queries

Fig 4 shows the latency incurred when the lazy and eager systems receive a query whose evaluation cannot be delayed, because it contains a read request that must be returned to the user. In particular, the query reads a single record from the database. We measure end-to-end latency from the time it begins execution to the time the query result is generated. We generate such "external read" queries at a rate of 1 in every 1000 transactions. In order to execute the logic of an external read query in the lazy system, we have to first execute all unevaluated dependencies of the records that the query reads (§3.2). Thus, the latencies in this experiment include overhead that is involved in maintaining and traversing the dependency graph of transactions, as well as the latency of executing an entire batch of transactions.

As shown in Fig 4, the end-to-end query latency of the lazy system improves with lower limits on transaction chain length. Intuitively, this occurs because a lower limit on transaction chain length forces batches of transactions to be evaluated before they get too large. Meanwhile, the eager system *always* outperforms the lazy system, because when the eager system takes the query off its input queue, it can immediately begin executing it. In contrast, the lazy system first needs to execute the query's unevaluated dependencies before it can begin processing the query. For the Normal workload, which has better cache locality, the cost of evaluating these dependency chains is smaller, which results in a better latency relative to the uniform workload. However, the eager system, which doesn't have any unevaluated dependencies at all, still yields smaller latencies than the lazy system on the Normal workload.

The main conclusion we draw from from this experiment is that when external read queries are rare, the latency to execute such queries can sometimes be much slower in lazy systems than in eager systems, due to the need to process unsubstantiated transactions before beginning query execution (note that the latency graphs used a log scale on the x-axis). In our experimental evaluation of TPC-C, we show the differences between lazy and eager systems when external read queries are more frequent (§4.4.1).

4.2 Temporal Load Balancing

In this section, we experiment with a lazy database system's ability to deal with bursty traffic. We set up the experiment in the same manner as those in the previous section. We experiment with the Uniform workload so that the lazy and eager databases will have the same baseline throughput, and changes to the baseline as a result of the load burst will be easier to observe. The lazy database's bound on maximum length of a chain of transactions is set to 100. During the course of the experiment, we sample the number of stickified transactions and the number of transactions that have been executed to completion. We sample these statistics every 100 milliseconds, and our plots show how these statistics vary over the course of time.

The experiment lasts for 300 seconds. During the time interval of 0-60 seconds, we warm up the lazy and eager databases with a load of about 100,000 transactions per second. At time t=60s, we simulate a load spike by suddenly increasing the input load to 660,000 transactions per second. We maintain this load during the time interval of 60-120 seconds. At time t=120s, we decrease the load down to 100,000 transactions per second, and maintain it during the time interval of 120-300s. The experiment ends at t=300s.

Fig 5 shows the results of the experiment. The topmost graph shows how the input load varies over the course of the experiment. The second graph shows the throughput of the systems as a function of time. For the lazy system, we measure two different types of throughput: (1) throughput observed by the user in terms transactions that have been committed (but in reality have only passed the stickification step) and (2) actual throughput of the system in terms of transactions that complete execution of both the stickification and and substantiations phases. We label the former throughput "stickification" in the figure, and the latter "lazy".

During the time interval of 0-60 seconds (when the offered load is 100,000 transactions per second), we see that both databases are able to keep up with the input load. The lazy database's stickification throughput (i.e. the throughput observed by the user) and the eager database's execution throughput both mirror that of the input load. In contrast, the "actual" transactional throughput of the lazy database is bursty. The reason for this behavior is that the lazy database does not substantiate transactions as soon as they enter the system. Instead, it accumulates batches of transactions until one of two scenarios occurs: either a chain of dependent transactions gets too long and needs to be pruned (so as to adhere to the bound on the maximum length of a chain), or it receives a transaction/query that

must be immediately evaluated (and its dependencies must therefore also be immediately evaluated).

During the time interval of 60-120 seconds, we increase the offered load to about 650,000 transactions per second. The throughput of the eager database increases to its maximum, but it is not enough to keep up with the offered load. As a consequence, the eager database begins dropping transactions as its internal queues begin to fill up. This can been seen in the bottommost graph, which shows the percentage of transactions that were able to be successfully processed. We see that the eager database can only successfully handle about 45% of the offered load. The second graph indicates that the substantiation throughput is *identical* to that of the eager system. However, despite substantiation being unable to keep up with the load, the lazy system does not drop transactions; it is able to handle 100% of the offered load. This is because the *stickification* layer is able to keep up with the offered load.

During the time interval of 120-300 seconds, we lower the offered load back to 100,000 transactions per second. Fig 5 shows that the eager system's throughput mirrors that of the offered load; it is now able to keep up with the offered load. In contrast, the throughput of the lazy system remains higher than the offered load. This occurs because the transactions that were processed by the stickification layer but not processed by the substantiation layer result in a backlog. This backlog of transactions to be substantiated is processed concurrently with incoming transactions. The substantiation layer finishes processing the backlog at around t=240 seconds. Once the backlog is processed, the throughput of the lazy system once again becomes bursty.

This experiment demonstrates that in a situation where the substantiation layer is not able to keep up with the load, substantiation can gracefully fall behind stickification and wait until the load burst is complete before catching up (this is why the "actual" throughput of the lazy system remains temporarily high after the load burst is complete). Unlike the eager system, the lazy system is able to *defer* the processing of transactions during resource constrained execution to a time when there are more resources available.

4.3 Blind Writes

Until this point, we have only experimented with transactions where all writes to a record are preceded by a read to that same record. However, some workloads contain "blind writes" to data (writes that are not preceded by reads to the same records). Blind writes are particularly beneficial to lazy systems, since by delaying writes to an item, these writes may never have to be performed if they are rendered unnecessary by a blind write.

Before getting into a discussion about our experiment, we first describe a scenario where blind writes may occur in practice. Consider the case of a customer using an online shopping portal. The shopping portal's database consists of two base tables – first, an inventory table of items a user can buy online, and second, a shopping cart table, each of whose records corresponds to a particular user's shopping cart. Every customer with an account on the shopping portal has a *private* shopping cart. The state of the shopping cart reflects the items on the shopping portal's catalog that the user is interested in purchasing. During a typical session on the shopping portal, she browses the portal's online catalog for items of interest (the online catalog may be a view of the inventory table). If she finds an interesting item, she adds it to her shopping cart, which causes a read of the online catalog to find the primary key of the item in order to add it to the shopping cart. Eventually a *check-out* operation is performed on her shopping cart. We define a check-out operation as one in which one of the two following possibilities occurs:

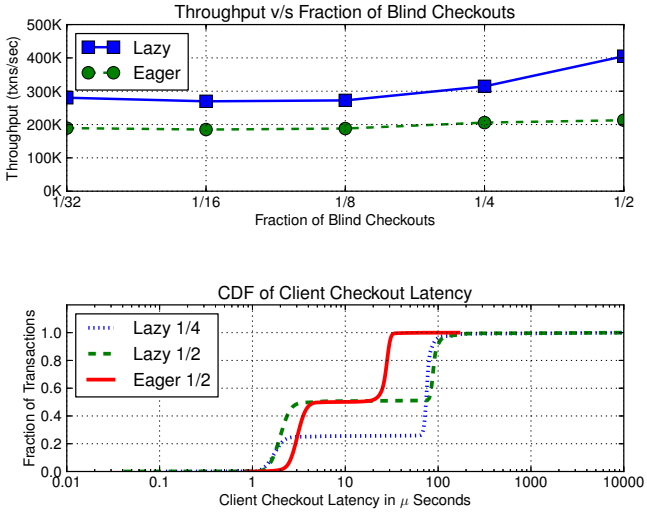

Figure 6: Blind Writes

- **Type 1.** The user may decide that she wants to buy the items in her shopping cart. In this case, the shopping portal's database back-end must read all the items in the user's shopping cart, and update the inventory table.
- **Type 2.** The user may decide that she does not want to buy any of the items she just added to her shopping cart. Instead she clears the contents of the cart. Alternatively, her session times out, and her shopping cart is cleared automatically. (We still use the term "checkout" for these two cases, even though many Websites would not call a cart clearing operation a "checkout".) The clearing of a shopping cart is a blind-write – the clear operation resets the state of the cart *without* performing any reads. The transactions that added items to the cart are no longer relevant because their effects were "clobbered" by the clear operation.

Our blind-write experiment is motivated by the scenario described above. The workload for our experiment is as follows: each client adds 20 items to its shopping cart and then proceeds to check-out according to one of the two cases described above. Adding an item to the shopping cart involves reading a particular record in the inventory table, and writing the shopping cart record. For simplicity, in the lazy database, we require check-out transactions to be evaluated immediately (even though in many cases a commit/abort decision may be all that is necessary to be returned to the user, and check-outs could therefore be executed more lazily).

The topmost graph in Fig 6 shows the throughput of the lazy and eager databases as we vary the fraction of blind writes. The throughput of the lazy database increases as we increase blind-write fraction because transactions whose results are clobbered by a blind-write do not need to be executed at all. However, these transactions still count towards throughput because they are "committed". The throughput of the eager database does not vary with blind-write fraction, since it is unable to benefit from not doing work that will eventually be rendered unnecessary.

The plot at the bottom of Fig 6 shows a CDF of the end-to-end latency of executing a check-out transaction. In the case of the lazy database, we plot the CDF of check-out latency for two different fractions of blind-writes (1/4 and 1/2), while in the case of the eager database, we plot just for the fraction of 1/2. For the lazy database, the end-to-end latency varies depending on the type of check-out. If the check-out is a blind-write, the lazy database must only evaluate a single transaction (which clears the state of the shopping cart). On the other hand, if the check-out is not a blind-write, it incurs

a higher latency penalty because the database must first execute all the transactions the check-out depends on (adding items to the shopping cart). Therefore, the latency distribution is bimodal. As we vary the fraction of blind-writes, the fraction of check-outs with low latency is precisely the same as the fraction of blind-writes.

The latency of a blind-write check-out in the lazy system is comparable to the latency of a check-out in the eager system. This is because both systems perform a similar amount of work — they both evaluate a single transaction. Since the blind-write check-out transaction (clearing the cart) is more lightweight than the non-blind check-out (making a purchase), the distribution of transaction execution latency in the eager system is also bimodal. However, the latency of executing a non-blind checkout (purchase) in the eager system is far less expensive than in the lazy system (the x-axis in the second graph is log-scale) since it has already processed all of the transactions that add items to the shopping cart.

In the scenario mentioned above, the blind-writing "checkout" transaction does not perform any reads. As a consequence, it does not need to wait for the result of any other transaction, and can execute immediately. A lazy database system may not be able to immediately execute transactions in workloads where blind-writing transactions have non-empty read sets. However, even in such a scenario, a lazy database will still never need to process transactions whose effects are clobbered by a blind-write.

4.4 TPC-C

Our final set of experiments are designed to evaluate the performance of lazy transaction evaluation on a known benchmark: TPC-C. The two differences we found in TPC-C relative to the other experiments we ran in this paper are that (1) External read queries appear more frequently (the $StockLevel$, and $OrderStatus$ queries) and (2) TPC-C contains many contended data accesses. In particular, the two transactions that make up the bulk of TPC-C's workload mix, $NewOrder$ (45%) and $Payment$ (43%), write records that are highly contended. Every $NewOrder$ transaction updates a District record, and every $Payment$ transaction updates a District record and its corresponding Warehouse record (each District record contains a foreign key, corresponding to a particular Warehouse's primary key). As a consequence, $NewOrder$ and $Payment$ transactions submitted to the database system will conflict with concurrently executing transactions involving the same District and Warehouse. In a traditional multithreaded database system, such conflicts inhibit scalability; the number of concurrently executing transactions is limited by the number of Warehouses in the system (the number of Districts per Warehouse is limited to 10).

Lazy transaction processing allows for an elegant solution to this scalability problem. Instead of executing writes to highly-contended records in worker threads, highly-contended records can be written solely by the stickification thread(s).

§2.3 explained that the stickification layer can *partially execute* part of a transaction immediately. We referred to the stickification layer's immediate partial execution of a transaction as a *now-phase*. In §2.4 we explained that the contention footprint of a transaction can be decreased by moving highly contended accesses to the now-phase, which, for the case of TPC-C, involves moving updates to the district and warehouse records (in the $NewOrder$ and $Payment$ transactions) to the now-phase.

4.4.1 TPC-C Commit Latencies

We now closely analyze the latency distribution of two transactions that are running in the context of the full TPC-C transaction mix: $NewOrder$ and $StockLevel$. We choose these two because they are representative of two fundamental types of transactions:

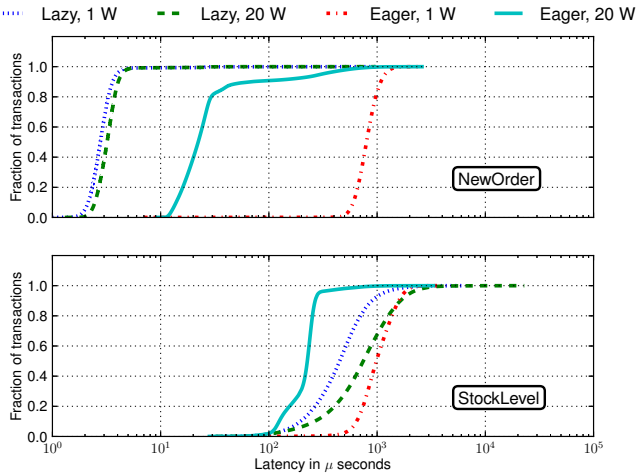

Figure 7: Client-observed latency for NewOrder and StockLevel transactions.

those that only have to immediately return a commit or abort decision ($NewOrder$), and those that need to immediately return the value of database state (an external read) to the user ($StockLevel$).

We plot the client-observed latency of $NewOrder$ and $StockLevel$ transactions while varying the contention in the system. Contention in the TPC-C benchmark is inversely proportional to the number of warehouses in the system. We therefore run each of our systems against two TPC-C warehouse configurations; the first with 1 warehouse (which has high contention), the second with 20 warehouses (which has low contention).

The top of Fig 7 shows the CDF of the client-observed latency of executing $NewOrder$ transactions. Lazy transactions clearly have at least an order of magnitude better latencies than eager transactions. This is because the client-observed latency is only the time it takes to receive the commit or abort decision. Since the lazy system does not have to process the entire transaction before returning the decision, it achieves much lower latencies. Meanwhile, the eager system must process the entire transaction before returning the decision to the client.

Furthermore, we see that the latency of eager transactions is significantly impacted by data contention. When there is only 1 warehouse, contention is very high, and most $NewOrder$ transactions must wait in lock queues before they can acquire locks and proceed. At 20 warehouses (low contention), these queuing delays are not present.

Since the lazy system also reduces contention by shrinking the contention footprint of the $NewOrder$ and $District$ transactions, its latencies are not affected by the number of warehouses. Lazy transactions thus have two advantages for transactions that only return commit/abort decisions: (1) they can return to the client without processing the whole transaction and (2) they reduce queuing delays due to contention.

The graph at the bottom of Fig 7 shows the CDF of client-observed latency to execute $StockLevel$ transactions. Since $StockLevel$ is an external read query, the lazy system no longer has the advantage of being able to return early, and therefore no longer outperforms the eager system by an order of magnitude. However, in contrast to the external read latencies for the microbenchmark presented in §4.1.2, the latencies for the lazy system are comparable to the eager system. To understand why this is the case, recall from §4.1.2 that the reason why lazy systems have high latencies for external reads is that they first need to execute the entire transitive closure of dependencies. In TPC-C, the transitive closure is generally much

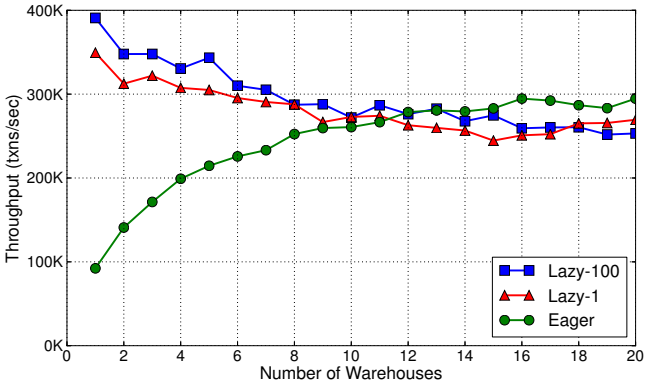

Figure 8: TPC-C throughput varying number of warehouses

smaller than in the microbenchmark because TPC-C contains more frequent external-read queries. In particular, the two external-read queries in TPC-C, $StockLevel$ and $OrderStatus$, each make up approximately 5% of the workload (recall that the microbenchmarks had one external read per 1000 transactions). These frequent external reads keep the dependency graph from getting large, which reduces the latency of other external reads (keeping them close in cost to reads in the eager system). On the other hand, the smaller dependency graph results in smaller transaction batches and reduced cache locality relative to the microbenchmarks (which we will discuss in more detail for the throughput experiments in the next section).

Unlike the eager system, the latency of the lazy system is *greater* for the 20 warehouse (low contention) case than the 1 warehouse (high contention) case. This is because with more warehouses, the stickification layer experiences larger costs for maintaining the dependency graph, which we explain in detail in the next section.

4.4.2 TPC-C Throughput

In order to analyze the effect of lazy transactions on TPC-C throughput, we run two configurations of the lazy system. The first is the same configuration of the lazy system we have been using in the experiments up until this point, with a bound on the maximum chain length of stickified transactions of 100 (henceforth called "lazy-100"). The second sets the maximum chain length to 1 (henceforth called "lazy-1"). Setting the maximum length chain to 1 forces each transaction's later-phase to be executed as soon as its now-phase completes. This ensures that there is no batching of transactions; when the maximum length is 1, the system is still able to achieve the contention benefits of lazy decomposition of transactions into the now-phase and later-phase, but all cache benefits of laziness are eliminated. By comparing the lazy-100 scheme with the lazy-1 scheme, we are able to distinguish between the cache locality and contention benefits of lazy transactions on TPC-C.

Fig 8 shows the results of our experiment. When the system runs with 1 warehouse, we see that both lazy-1 and lazy-100 achieve a substantially higher throughput than the eager system, while the difference between the two lazy systems is more modest. Recall that lazy-100 is able to achieve both the cache benefits and contention benefits of lazy processing, while lazy-1 is only able to achieve the contention benefits. Therfore, the difference between these two lines can be attributed to the cache effects of lazy execution. The rest of the difference of the lazy systems relative to the eager system is due to the contention benefits of lazy execution. We therefore conclude that lazy transaction execution benefits both from improved cache locality and reduced contention, but the benefits of reduced contention are larger for TPC-C. The reason why the cache benefits are not large in this case is explained in the previous section — the large number of external read queries reduces the size of the dependency graph, limiting the amount batching that the lazy system is able to perform.

As the number of warehouses increases, the contention in the system decreases. As a result of the reduced contention, we see that the throughput of the eager system steadily increases until about 12 warehouses, after which contention is no longer the bottleneck, and throughput stabilizes.

In contrast to the eager system's increase in throughput, the throughput of lazy-1 and lazy-100 *decreases* as we add more warehouses. This decrease in throughput occurs because the throughput of both lazy-1 and lazy-100 is limited by the throughput of the stickification layer. The stickification layer performs two tasks for every transaction that enters the system; first, it processes the transaction's now-phase, and second, it maintains a dependency graph of transactions to be processed by the substantiation layer. While the now-phase is short for TPC-C (only reads or writes to records in the Warehouse and District tables are performed), the overhead of dependency graph maintenance is much higher. Maintaining the dependency graph of transactions involves tracking the last transaction to write a particular record by maintaining an *inverted index* from each record to its last writer (§3.1.1). Since the inverted index tracks the last writer of *every* record in the database, the size of the inverted index increases as we increase the number of records in the database. As the size of the inverted index increases, a smaller percentage of it remains in cache, and the stickification layer must may a higher cost to update it.

The fact that the stickification layer becomes the lazy system's primary bottleneck as the size of the database scales is specific to our current implementation, and not fundamental to lazy execution. As mentioned above, our implementation of the stickification layer consists of just a single thread executing every transaction's now-phase and maintaining the dependency graph. Multithreading the stickification phase is an important avenue for future work.

5. RELATED WORK

Although database system researchers often use the term "lazy" specifically to describe database replication strategies [9, 3, 16], often in the context of eventually-consistent replication schemes, we use it in this paper in a completely orthogonal context. We talk about laziness in the context of programming language theory, where *lazy evaluation* is an evaluation strategy in which the evaluation of an expression is delayed until the value of the expression is actually needed. This is in contrast to *eager* or *strict* evaluation, in which an expression is evaluated as soon as it is bound to a variable. Most programming languages use eager evaluation by default, while a few choose lazy evaluation [12]. Our work therefore builds on the theoretical foundations and early implementations of lazy programming languages [24, 10, 6, 11]. The key contribution of our work is the application of these lazy execution techniques to transaction execution in database systems.

Buneman et al. were the first to apply programming language style lazy evaluation techniques to database systems [4]. This research introduced a functional query language called FQL through which users could express queries against a database that facilitates lazy execution of read-only queries so that subexpressions, such as nested queries, are not evaluated repeatedly, and if their results are not ever needed due to the semantics of a query, these subexpressions never have to be evaluated. Morton et al. [15] also propose lazy evaluation of read-only queries in a slightly different sense; if the complete result set is not immediately required by a

downstream client, only the subset of results that will be used immediately need to be calculated. Note that read-only queries do not change database state, and therefore it is a very different problem to apply lazy evaluation techniques in these contexts. In contrast, our techniques evaluate entire transactions lazily.

As part of our lazy execution design, we break transactions up into a now-phase and later-phase, which allows a commit decision to be returned before a transaction is fully executed (which is critical for lazy execution), and also has the potential to reduce contention by reducing the size of critical sections protecting highly contended data items. There have been several related proposals to decompose transactions into pieces, either to improve contention [19, 7, 1], or to facilitate consistent replication [25, 13]. However, these previous proposals either reduce serializability guarantees as a result of the decomposition [7, 1], or require knowledge of the complete set of transactions that will be run against the system in order to do a static analysis of these transactions that will determine how transactions can be decomposed without violating serializability [19, 13, 25]. The key advantage of our lazy decomposition approach, that distinguishes it from all of these prior approaches, is that it maintains serializability guarantees *while not requiring any of the decomposed pieces to be globally commutative.* In particular, actions in both the now-phase and later-phase may be dependent on or conflict with any other transaction. The lazy execution framework tracks and deals with these dependencies to maintain a guarantee of serializability. This greatly facilitates the decomposition process and extends the applicability of the mechanism.

To maximize the contention benefits of decomposition of transactions into now and later-phases, the now-phase should be short and contain the most highly contended operations. Several recent papers have introduced predictive and/or static mechanisms to anticipate accesses and dependencies of transactions [17, 5]. This work can be used help automate the process of transaction decomposition.

Prior systems have proposed buffering and batching techniques to improve cache locality and facilitate the sharing of intermediate results of transactions. Zhou et al. [26] propose buffering the output of operators in a pull-based pipelined query execution model. They propose a coarse-grained interleaving of the execution of consecutive operators so as to improve instruction cache locality of each operator. However, their technique addresses the problem of instruction cache thrashing in the specific case of a single threaded pipeline of operators. Giannikis et al. [8] propose compiling a global query plan for large batches of concurrent queries and updates. Using a global query plan allows for intermediate results to be shared across transactions in a batch. In contrast to this prior work, lazy systems have more flexibility in creating batches, since they are able to create batches containing transactions that have been submitted to the system over a temporally diverse set of time.

6. CONCLUSION

While lazy evaluation has been applied in the past in programming languages, it is interesting and perhaps surprising to note that laziness in the context of database systems has a largely different set of advantages and applications than in programming languages. In particular our experimental implementation shows that lazy evaluation of transactions in database systems can improve cache locality, temporally load balance a workload during spikes of transactional activity, simplify concurrency control, and reduce latency for transactions that return only a commit or abort decision. Not all workloads are well-suited for lazy evaluation, as some queries are delayed as long chains of dependencies are evaluated, but our experimental results show that there is an interesting class of workloads for which lazy evaluation is able to improve throughput and latency.

Acknowledgments This work was sponsored by the NSF under grants IIS-1249722 and IIS-1422205, and by a Sloan Research Fellowship.

7. REFERENCES

[1] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24:673–698, 1999.

[2] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[3] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. *SIGMOD*, 1999.

[4] P. Buneman, R. E. Frankel, and R. Nikhil. An implementation technique for database query languages. *ACM Trans. Database Syst.*, 7(2):164–186, June 1982.

[5] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Speeding up database applications with pyxis. In *SIGMOD*, 2013.

[6] D. Friedman and D. Wise. *Cons should not evaluate its arguments*. Automata, Languages and Programming. 1976.

[7] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of SIGMOD*, pages 249–259, 1987.

[8] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, Feb. 2012.

[9] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[10] P. Henderson and J. H. Morris, Jr. A lazy evaluator. POPL, 1976.

[11] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[12] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *SIGPLAN*, 2007.

[13] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. of OSDI*, 2012.

[14] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of ICDE*, 2014.

[15] K. Morton, M. Balazinska, D. Grossman, and C. Olston. The case for being lazy: how to leverage lazy evaluation in mapreduce. ScienceCloud, 2011.

[16] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. VLDB*, pages 126–137, 1999.

[17] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *PVLDB*, 5(2):85–96, 2012.

[18] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. PVLDB, 2013.

[19] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, Sept. 1995.

[20] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. of VLDB*, 2007.

[21] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *Proc. of VLDB*, 2010.

[22] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[23] VoltDB. Website. voltdb.com.

[24] C. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971.

[25] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of SOSP*, 2013.

[26] J. Zhou and K. A. Ross. Buffering databse operations for enhanced instruction cache performance. SIGMOD, 2004.